Solution-based

Heuristic Algorithms

for Combinatorial Optimisation

Lecture notes

Prof. Roberto Cordone

Edoardo Marangoni

University of Milan
Department of Computer Science
September 26, 2024

Contents

Foreword

I wish to thank Edoardo Marangoni for the remarkable effort of collecting these lecture notes, which concern the teaching on "Heuristic algorithms" of the Computer Science course. They have been collected in the academic year 2020/21, based on the slides and the videorecordings of the course. I have personally added the sections on the laboratory sessions and the exercises, and I am gradually revising the content of the other sections. The readers are encouraged to point out mistakes or ambiguities.

Roberto Cordone

Part I

Introduction to problems and heuristics

CHAPTER 1

Introduction

This chapter explains the goals of the course on "Heuristic algorithms", starting with a discussion on the word *heuristic*, which has several meanings, depending on the field in which it is used. Then, we surview the problems addressed in the course, characterise the family of algorithms that will be described and classify them. In fact, the course does not consider *all* heuristic algorithms for *all* problems, but only **solution-based heuristics** for **combinatorial optimisation problems**.

The main take away of this course is that heuristic algorithms are **not recipes for specific problems**. Therefore, this course is not just a parade of specific algorithms used for specific issues. The theme of this course can be summarised by saying that *any* heuristic can be used on *any* problem, obviously with different performance and outcomes. The point is to consider general ideas that can be applied to a wide range of problems. So, we will first discuss common heuristic frameworks to design algorithms and then tailor them to specific problems, considering the features of each problem to determine whether they can work on it or not. Finally, we will implement some algorithms in laboratory sessions and quantitatively evaluate their performance (see the Appendices).

1.1 Heuristics

The term **heuristic** comes from the greek *eurisko*, meaning *I find*. This word is derived from the very famous story of Archimedes, who had the problem of finding out whether a golden crown the tyrant of Syracuse had been given was actually made of gold or not. The Greeks knew that gold is heavy, so if one knows the weight and the volume of the crown, their ratio tells whether it is solid gold or only plated, or an alloy. But, while measuring the weight is easy, measuring the volume of an object shaped like a crown is hard. Archimedes realized that if you put something into water, the water displaced has exactly the same volume as the object, and that water can be put into shapes whose volume is easy to measure. Eureka!

1.1.1 History

The word heuristic was coined in the 19th century, so no Greek ever used it. The Greeks, however, talked about something similar to heuristics: a mathematician called Pappus of Alexandria, living around the 4th century AD, published a book titled "*Analyòmenos*", meaning *treasure of analysis*, in which he discussed how to build a mathematical proof, starting from the hypotheses and coming to the thesis, or solve a problem, starting from the data and coming to the result. In the 17th century, mathematicians like Descartes and Leibniz discussed the *ars inveniendi* -

which means *art of finding*. Finding what? - one may ask: mathematical truth, of course. In the 19th century, Bernard Bolzano discussed the *Erfindungskunst*, once again the art of finding, which treated the most common strategies to build mathematical proofs. Shortly after, in a completely different context, philosophers, psychologists and economists started considering what they called heuristics, : heuristics were now pratical and simple rules that do not aim at an optimal result but at a *satisficing* one. Finally, in 1945, György Pólya published a book entitled "How to solve it" using the term "heuristic" with its original mathematical meaning.

1.1.2 What is a heuristic algorithm?

Considering the field of neuropsychology, philosopy, law and so on, the meaning of *heuristic* is the exact opposite of *algorithm*, as the latter is a formal procedure composed of deterministic steps leading to the solution of a problem, while the former is a *creative* solution. In other words, the algorithm has some correctness proof while the heuristic is drawn from common sense. So what does it mean to have a **heuristic algorithm**?

A heuristic algorithm is a procedure run by a computer and composed by well defined steps (so, it is an algorithm), but it does not have a correctness proof, so it is not guaranteed to be correct. This sound strange and useless, one may say. On the contrary, it can be quite useful provided that it *costs* much less in space and time than a possible exact alternative and that it *frequently* yields something *close* to the correct solution. The *closeness* is defined following a definition of *distance*, so it becomes necessary to introduce a **metric** in the solution space, in order to determine if a solution is satisfactory or not. As well, it becomes necessary to introduce a **probabilistic distribution** in the solution space, that expresses how often the algorithm yields a satisfactory solution.

Going back to the idea of proofs, problems and so forth, we can observe that an exact algorithm is always accompanied by a proof of its correctness and, often, the proof matches the algorithm going through all of its steps and showing that, starting from the data, the steps of the algorithm guarantee certain mathematical properties, until in the end some new mathematical property is introduced, and the combination of all these properties guarantees the correct solution. The relation between the algorithm and its proof is therefore very tight. The process of building the proof typically starts from heuristic general ideas, that point in the direction of the correct solution and are tested and combined until they form a proof. Heuristic algorithms fail to reach a proof. Hence, the basic ideas used in order to find and justify the algorithm remain a practical support, but they are not strong enough to constitute a proof.

1.2 Different types of problems

Let us take a step further and define the problems that are going to be addressed. The course focuses only on heuristic algorithms that apply to **Combinatorial Optimisation** problems and only on **solution-based** algorithms (as opposed to **model-based** ones). Even though we limit the kind of problems and algorithms, it is still a pretty wide field.

A problem is generically a question on some mathematical system. These problems can be classified on the basis of the nature of their solution: **decision problems** have a boolean solution (true or false), for **search problems** the solution is a mathematical object belonging to the given system, for **optimisation problems** the solution is a number, namely the value of a suitable function estimating the quality of mathematical objects drawn from a suitable collection in the given system. An example is the problem of finding the minimum time path to go from home to the town center: the mathematical system is the map of the town; the objects of the collection

are paths from the origin to the destintion. Each of them has a time length and the problem is to find the path with the shortest total time. Optimisation problems can be combined with search problems, by searching not only the value, but also the corresponding solution: these are the problems considered in the course. Other families of problems are **counting problems** (find the number of objects in the collection) and **enumeration problems** (list the objects in the collection).

1.2.1 Optimisation-search problems

An optimisation problem can be represented as

$$\operatorname*{opt}_{x \in X} f(x)$$

where x is a solution (a description of a mathematical object), the **feasible region** X is the collection of objects (defined by satisfying suitable conditions) and $f : X \to \mathbb{R}$ is the **objective function**, that quantitatively measures the quality of each object in the collection. Symbol opt usually stands for min or max.

The problem consists in determing the **optimal value** f^* of the objective function in the feasible region

$$f^* = \operatorname*{opt}_{x \in X} f(x)$$

Optimisation-search problems consist in determing at least one **optimal solution**:

$$x^* \in X^* = \arg\operatorname*{opt}_{x \in X} f(x) = \left\{ x^* \in X : f(x^*) = \operatorname*{opt}_{x \in X} f(x) \right\}$$

where the arg function provides the whole set of solutions that optimise the objective.

People are interested in optimisation-search problems because several application fields require objects, or structures, characterised by very high or very low values of a suitable evaluation function. Exact optimisation is nice, but usually very costly from a computational point of view. Therefore, heuristics can be the more viable alternative. Moreover, in some fields the evaluation function is just an approximation of what really goes on in the physical system, and therefore investing time in exact optimisation can be not really required. In this course we will assume anyway the point of view of the applications in which an optimal solution is really preferable, and therefore we will try and improve the objective function as much as possible.

A relevant point is that many search problems or decision problems can be transformed into optimisation-search problems, in particular when they require to find objects that satisfy interesting condition or to determine whether they exist. The ides is to relax some of those conditions until it becomes easy to find objects that satisfy the other ones, and then minimise the violation of the relaxed conditions. For example, consider the famous *four-colour problem*: is it possible to colour a map with exactly four colors, so that any two bordering states have different colours? Seen as a decision problem, its solution is true or false. Seen as a search problem, its solution is a mathematical object (a coloured map). Both can be reduced to an optimisation problem asking for a colouring of the map with a minimum number of colours. Starting with more than four colours (possibly even one different colour for each state), one can measure the violations to the threshold of four colours as the difference between the actual number of colours and 4, and minimise the violation by reducing the number of colours used. Once an optimal colouring is found, if it has zero violations both the search and decision problem are solved positively; if it has more, they are solved negatively, proving that four colours are insufficient.

1.2.2 Combinatorial Optimisation

A problem is a Combinatorial Optimisation problem when the feasible region X is a **finite set**, so the problem has a finite number of solution. This is a restrictive assumption, but there are lots of problems which have infinitely many solution that can be somehow reduced to a finite subset of potentially optimal solution. An example is linear programming, where at least one of the optimal solution is a *basic* solution, so one just needs to consider the basic solutions. In addition, lots of ideas developed for Combinatorial Optimisation problems can be extended to other problems with infinite solutions.

An alternative definition of Combinatorial Optimisation is the following: they are problems whose solutions are *combinations* (subsets with irrelevant ordering) of a finite ground set. That is also why these problems are known as *combinatorial*. So, the second definition presents the feasible region as $X \subseteq 2^B$ for a suitable finite ground set B. We recall that, if B is a given finite set, 2^B is the *power set* of B, which contains every subset of B, from the empty set $\emptyset$, to the singletons, pairs, triplets, to the whole set itself. The feasible region X is a subset of this set, so every solution (element of X) is a subset of 2^B.

1.3 Different types of heuristics

1.3.1 Solution-based heuristics

Solution-based heuristics are characterised by the operations that one can make on subsets of a finite set. If one wants to find a subset of a set, one can mainly do three things. The first one is build the subset starting from the empty set $\emptyset$ and iteratively adding elements until a certain condition is found. This is a form of **constructive heuristic**: starting from an empty set, construct a solution adding elements. The idea can be reverted by starting with the full set and removing elements from it. This gives rise to a **destructive heuristic**. The second thing one can do is to exchange elements of a current subset obtained somehow. In a sense, constructive and destructive heuristics also exchange elements, but the subsets in that case are strictly growing or reducing. In **exchange heuristics**, the elements of the subsets are exchanged in a general way, with no imposed limitation on the direction of the exchange. Finally, the third family of heuristics consists of **recombination heuristics**, where subsets are created combining parts of several given subsets, instead of modifying a single set.

In general, designers of heuristic algorithms can use one of these techniques, but also freely combine them, adapting to the specific features of the problem.

1.3.2 Metaheuristics

There are two important mechanisms that can be exploited in the design of an algorithm: **randomisation** and **memory**. These two mechanisms are perfectly orthogonal to the classification seen before, meaning that any kind of algorithm can use randomisation or not, and use memory or not. The full set of options of heuristics, therefore, counts 3 families and $2 * 2 = 4$ possibilities for each, that is 12 overall. the heuristic algorithms that use randomisation or memory, or both, are called **metaheuristics**, from the Greek *meta*, that stands for *beyond*.

Randomisation consists in performing operations that are not completely determined by the data and the results of the previous operations, but partly random. This apparently conflicts with the definition of algorithms as performing deterministic operations, but the contradiction

is solved by the fact that randomised algorithms are actually deterministic, but require an additional information in input, besides the data (typically an integer number, known as **random seed**), that is used by a `pseudorandom number generator` to build a sequence of integer numbers, which the algorithm uses to decide which operations to perform. Different random seed imply different sequences, and different results for the same data. Repeating the execution with the same random seed, on the contrary, guarantees to obtain the same solution. The random seed is totally arbitrary and can be provided by the user directly or indirectly, using an entropy source (the computer's current time, keyboard activity, radioactive decay...).

Memory consists in performing operations that depend not only on the data, but also on the solutions previously obtained by the same algorithm. To use a trendy name, it can be therefore denoted as a form of *machine learning*.

1.4 Caveats

When designing a heuristic, it is adviseable to be aware of some risks, that should be avoided.

- choosing an approach for other reasons than its intrinsic value (there are communities in which a certain algorithm is considered good in every situation: one has to prove that an algorithm is good);
- design an algorithm based on analogies with physical and natural phenomena (they can be good ideas, but only after a proof is given);
- using heuristics instead of exact algorithms with an acceptable complexity (this is plain stupid);
- performing "number crunching", complex computations on unreliable data, relying on hardware power ("SUV attitude");
- using components and parameters without justify their introduction;
- overfitting, that is adapting components and parameters to the specific dataset used in the experiments.

It is fundamental to free oneself from prejudice and evaluate the performance of an algorithm in a purely scientific way, distinguishing the contribution of each component to the final result.

CHAPTER 2

Combinatorial optimization problems

A Combinatorial Optimisation problem requires to minimise or maximise an objective function $f(x)$, defined as $f : X \to \mathbb{R}$, over the feasible region X, that is a subset of the power set of B ($X \subseteq 2^B$), with B being a finite ground set; therefore, $x \in X \subseteq 2^B$ that is $x \subseteq B$, or, x is a *combination* of elements of B.

This section presents a list of Combinatorial Optimisation problems in order to give an idea of what they are, how to recognize one and to know how to deal with it. The first basic points is to identify the ground set B, as it is the foundation on which an algorithm to solve the problem is built: constructive/destructive, exchange and recombinaton heuristics all operate on subsets of the ground set. We shall start with **set problems**, revolving around sets, and then consider **logic function problems**, **numerical matrix problems** and **graph problems**.

In the following chapters, when we will consider general heuristic approaches, it will be useful to **map the abstract ideas onto practical problems**, and therefore it will be useful to have a wide list of potential problems. Applying each approach to different problems allows to understand it better. Moreover, the same abstract ideas can be good or bad depending on the *structure* of the specific problem to which they are applied. Some ideas can even be restricted to work only on some problems. Finally, some problems that are apparently different have hidden structures that the algorithms can exploit, and it is therefore useful to see examples and learn how to exploit such relations. So, we are trying to apply abstract ideas to many problems, to understand what they have in common and what is different.

2.1 Weighted set problems

2.1.1 The knapsack problem

Given a knapsack of limited capacity, one wants to fill it with different objects, each of which has a value and a weight. The objects do not fit altogether inside, so a choice has to be made. The objective is clearly to find the subset of **maximum value** that fits in the knapsack. The problem is mathematically defined by the set E of elementary objects, the function $v : E \to \mathbb{N}$ describing the volume of each object, the number $V \in \mathbb{N}$ describing the capacity of the knapsack and, finally, the function $\phi : E \to \mathbb{N}$ describing the value of each object.

See Section 2.7.1 for a deeper discussion of this problem. Its ground set can be trivially defined as the set of objects, $B \equiv E$; the feasible region includes all the subsets of objects whose total volume does not exceed the capacity of the knapsack:

$$X = \left\{ x \subseteq B : \sum_{j \in x} v(j) \leq V \right\}$$

The objective is to maximise the total value of the chosen objects, that is

$$\max_{x \in X} f(x) = \sum_{j \in x} \phi(j)$$

E	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1

Table 2.1: An example of the Knapsack problem

Table 2.1 gives an example of possible data for this problem. Given the case represented in the table and a capacity $V = 8$, consider the two subsets of objects $x' = \{c,d,e\} \in X$, which has $f(x') = 13$, and $x'' = \{a,c,d\} \notin X$, which yields $f(x'') = 16$. One can see that x'' is not a **feasible** solution as it is not in X. Strictly speaking, it is not a solution, but it is often defined as an unfeasible solution.

2.1.2 Maximum diversity problem

The Maximum Diversity Problem (MDP) is defined on a metric space, that is a space with a notion of *distance*. It considers a set of points P, a positive integer number $k \in \{1, \cdots, |P|\}$ that represents the cardinality of the subset to select, and a function $d : P \times P \to \mathbb{N}$ that provides the distance between any two points.

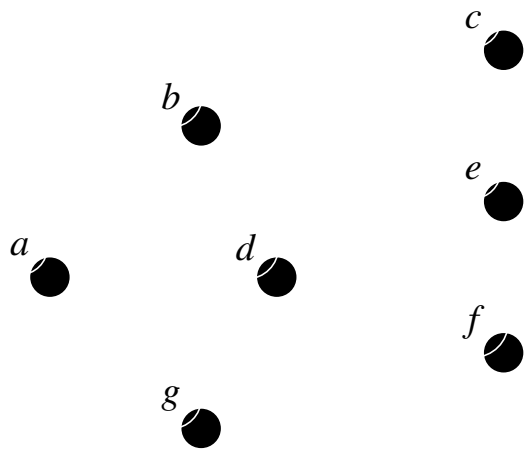


Figure 2.1: An instance of the Maximum Diversity Problem

The aim is to find a subset of exactly k points, such that the sum of all the pairwise distances between the points in the subset is maximised. It is a Combinatorial Optimisation problem because the number of combinations of k points from the given n is finite. The ground set is the set of all points $B \equiv P$ (see also Section 2.7.2). The feasible region includes all subsets of k points:

$$X = \left\{ x \subseteq B : |x| = k \right\}$$

and the objective function maximises the sum of distances:

$$\max_{x \in X} \; f(x) = \sum_{(i,j):i,j \in x} d(\langle i, j \rangle)$$

Figure 2.2 represents two solutions with $k = 3$.

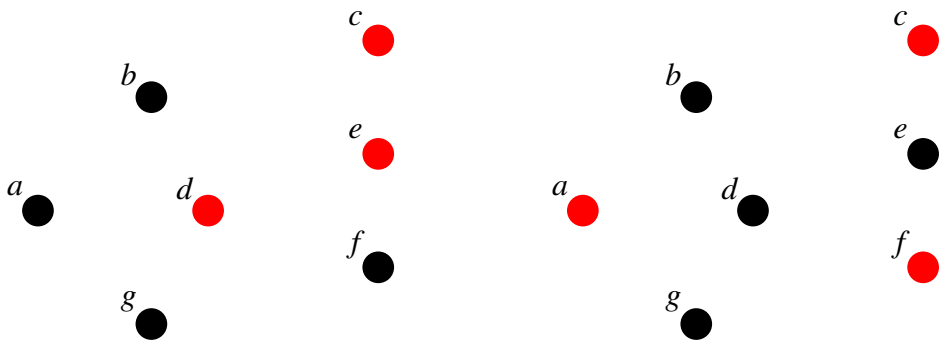


Figure 2.2: Two possible solutions for the MDP instance of Figure 2.1: $x = \{c,d,e\}$ and $x' = \{a,c,f\}$.

2.1.3 Interlude I: the objective function

The objective function is defined as a function from the feasible region to the natural numbers:

$$f : X \to \mathbb{N}$$

In general, it can be complex to compute this function. In the worst case, each solution could have its own value, and the only way to compute it could be to scan the list of values and find the correct one. Of course, this is not the case of the previous two problems.

The KP has an **additive objective function**: it is a sum of the values that an auxiliary function, defined on the ground set, assumes on the elements of the solution:

$$\phi : B \to \mathbb{N} \text{ induces } f(x) : X \to \mathbb{N} \text{ to be defined } f(x) = \sum_{j \in x} \phi(j)$$

This is interesting, as it means that one needs to store only the values of the auxiliary function ϕ instead of those of the objective function, that is $|B|$ values instead of $|X|$. The same holds for the MDP and its auxiliary function d. The difference is that the objective function here is quadratic, that is a sum of values of an auxiliary function defined on the pairs of elements of the solution. Computing it requires to sum sum $(k*(k-1))/2$ numbers (the distance from each point to each other point). As $k \leq n = |B|$, the computation is still of polynomial complexity.

What happens to the objective function if a solution is modified? Is it necessary to recompute its value from scratch, or can you update it, more efficiently? For the KP, and for all additive objective functions, if a solution is modified adding new elements and removing some original ones, the value of the objective function can be obtained simply summing the values of the auxiliary function for the new elements and subtracting those of the removed ones. For example, in the KP instance considered above $f(\{c,d,e\}) = 13$; if we remove e and add f, then $f(\{c,d,f\}) = 13 - \phi(e) + \phi(f) = 10$. This takes constant time if the number of added and removed elements is constant.

The objective function of the MDP must be treated differently: we can avoid to recompute it from scratch, but the terms that must be summed and subtracted are not a constant number, even if the number of removed and added elements are constant. The complexity is linear in k. We shall see that it is actually possible to update the value of the objective in constant time, but it takes a more refined approach.

Another important remark is that the objective function for the KP and the MDP problem are defined on the whole of 2^B, instead of just on X. This is not really necessary, but we shall see that it can be helpful.

In summary, important questions on a problem are the cost to compute the objective function of a solution from scratch, the cost to update it after small modifications, the actual domain on which it is defined, and the family to which it belongs (additive, quadratic, etc...).

2.2 Partitioning set problems

Now we consider two problems in which a given set of objects must be divided into disjoint subsets, obtaining a partition that satisfies suitable conditions.

2.2.1 Bin packing problem

In the Bin Packing Problem (BPP), a set of objects with a volume is given and each of them must be put in a container of fixed capacity, drawn from a set of containers C. The objective is to use the least number of containers, never exceeding the capacities. Formally, a set E of elements (or object) is given, and a function $v : E \to \mathbb{N}$ gives the volume of each object; a set C of containers is given and a number $V \in \mathbb{N}$ represent the volume of the containers (the same for all of them).

How to describe this problem as a Combinatorial Optimisation problem?
How can a solution be described as a subset of a suited ground set B?

We can define the ground set (see also Section 2.7.3) as

$$B = E \times C$$

that includes all the (element,container) pairs. A solution, in fact, is a subset of such pairs: take the first object and put it in a certain container, take the second and put it in a certain container, and so on. A solution consists in a list of such pairs that assign all the elements in E to suitable containers. Let us define $B_e = \{(i, j) \in B : i = e\}$ as the set of all ordered pairs in the ground set that refer to the object e, and $B^c = \{(i, j) \in B : j = c\}$ as the set of all ordered pairs in the ground set that refer to the container c. The feasible region includes all partitions of the objects among the containers that do not exceed the capacity of any container:

$$X = \left\{ x \subseteq B : (\forall e \in E | x \cap B_e| = 1) \wedge (\forall c \in C \sum_{(e,c) \in x \cap B^c} v(e) \leq V) \right\}$$

In words, the first part of the condition is a partition constraint (every element must appear exactly once in the solution) and the second is a volume constraint (the sum of the volumes of all elements in a container must not exceed the volume of the container, for all containers).

The objective is to minimise the number of containers used:

$$\min_{x \in X} f(x) = |\{c \in C : x \cap B^c \neq \emptyset\}|$$

Figure 2.3 gives an example of a BPP instance and solution.

2.2.2 Parallel machine scheduling problem

The Parallel Machine Scheduling Problem (PMSP) requires to divide a set of tasks among a set of machines, so that the total completion time (i. e., the time at which all machines have stopped working) is minimised. Each task has a time length, or duration d: the problem is to assign tasks to machines so that the maximum time a machine employs to solve all of its tasks

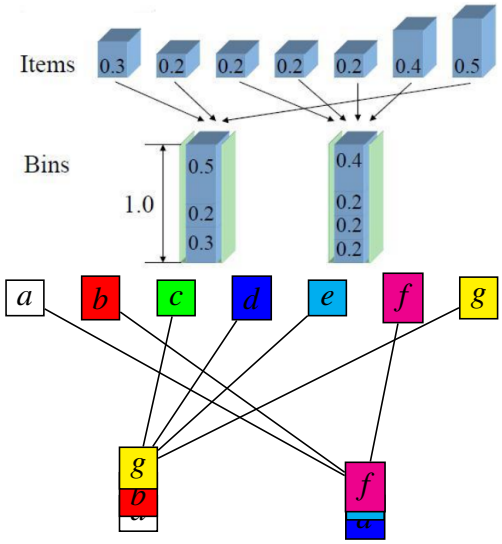


Figure 2.3: An instance of the BPP

is minimised. It is important to underline that the *sequence* in which the tasks are executed by each machine is not important: only the completion time matters, and this depends on the sum of the durations of the tasks assigned to each machine. Formally, a set T of tasks and a set M machines are given, and each task has a value d ($d : T \to \mathbb{N}$) describing its duration.

As before, the ground set is $B = T \times M$, including all possible $(task, machine)$ pairs (see also Section 2.7.4). The feasible region includes all partitions of tasks among machines

$$X = \left\{ x \subseteq B : \forall t \in T | x \cap B_t | = 1 \right\}$$

and the objective function is

$$\min_{x \in X} f(x) = \max_{m \in M} \sum_{t:(t,m) \in x} d(t)$$

In words, the solution is a partition, since every task must appear exactly once in it. The objective minimises the maximum value with respect to the machines of the total execution times.

2.2.3 Interlude II: again, the objective function

The last two problems show that the ground set B is not always one of the sets given by the problem. It may also be some combination of such sets. Considering the questions introduced in the last interlude, the two objective functions are not additive: they are not sums on the elements of the solution. Even considering the number of containers used as a sum of units (one for each container), this is not a sum on the (element,container) pairs included in the solution. Even considering that the completion time includes sums of durations, they are partial sums only one of which (the maximum) is considered. Both objectives are not trivial to compute, though there is still a polynomial algorithm that calculates their values.

Once again, the objective function is defined not only on feasible solutions, but also on general subsets of the ground set.

When a feasible solution is modified, the impact on the objective value is variable. In some cases, nothing happens (moving a task from a machine to another sometimes does not change the overall maximum completion time). In other cases, the objective function changes as much as the duration of the moved task (this happens when the task is moved to the machine with the

overall maximum execution time). In yet other cases, the objective function has an intermediate change (this happens when the task moves from the machine with the maximum execution time to another, or when it moves to a machine that had a time close, but not equal, to the maximum.

The question whether these values can be updated without recomputing them from scratch will be considered later: the answer is not trivial, but very important for the overall efficiency of the algorithm.

Another interesting point is that the objective function is **flat**, meaning that several solutions have the same value. As one might imagine, this can be a problem when the algorithm has to choose which modification to perform in order to improve a solution.

2.3 Logic function problems

2.3.1 Max-SAT problem

The Max-SAT problem, where SAT stands for *satisfiability*, is very well known in computer science. It concerns a set V of logical variables x_j with a boolean value (true or false). By **literal** we denote a very simple logical function consisting of an affirmed or negated variable

$$l_j(x) \in \{x_j, \neg x_j\} \text{ with } j = 1, \ldots, n$$

Literals are combined in a logical sum (with non exclusive ORs operators) to create **disjunctions**, called *logical clauses*

$$C_i(x) = l_{i,1} \vee \cdots \vee l_{i,n_i} \text{ with } i = 1, \ldots, m$$

In turn, the clauses are combined in a logical product (with AND operators) to build a *conjunctive normal form* (CNF):

$$CNF(x) = (l_{1,1} \vee \cdots \vee l_{1,n_i}) \wedge \cdots \wedge (l_{n,1} \vee \cdots \vee l_{n,n_i})$$

Statisfying a logical function means finding a truth assignment, that is an assignment of logical values to the logical variables so that the overall function assumes the value true (1). The CNF is called *normal* because a theorem proves that any logical function can be transformed into a logically equivalent CNF, a form that assumes the same values as the original function for any truth assignment.

The Max-SAT problem requires to find a truth assignment that maximises the number of satisfied clauses. It is possible to consider a *weighted* version, defining a weight function on the set of clauses and maximising the sum of the weights of the satisfied clauses. For example, let the set of variables be

$$V = \{x_1, x_2, x_3, x_4\}$$

so that the set of literals is

$$L = \{x_1, \neg x_1, \cdots, x_4, \neg x_4\}$$

Let the CNF to satisfy be

$$CNF = (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_2 \wedge \neg x_4) \wedge x_1 \wedge x_2$$

In the unweighted case ($\forall C_i : w_i = 1$), solution $x = \{(x_1, 0), (x_2, 0), (x_3, 1), (x_4, 1)\}$ satisfies 5 clauses out of 7, so $f(x) = 5$.

The ground set (see also Section 2.7.5) consists of all the possible assignments of values to the single variables: $B = V \times \{1,0\}$. Any truth assignment, in fact, is a subset of this ground set. Not all subsets are solutions, however. The feasible region includes all subsets of truth assignments that are **complete** (each variable has at least a literal) and **consistent** (each variable has at most one literal). We therefore define the feasible region as

$$X = \{x \subseteq B : \forall v \in V | x \cap B_v| = 1\}$$

where $B_{x_j} = \{(x_j,0),(x_j,1)\}$. Notice the similarity with the partition problems discussed above: the variables must be partitioned into true and false ones.

The objective is to maximise the total weight of the satisfied clauses

$$\max_{x \in X} f(x) = \sum_{i:\, C_i(x)=1} w_i$$

The complexity of computing the objective function is polynomial. The objective is not defined on unfeasible solutions (unless one wants to adopt a convention for inconsistent assignments). The objective function is typically flat in unweighted instances, as the possible values of the objective function range from 0 to m, whereas the number of solutions is much larger (exactly 2^n).

2.4 Numerical matrix problems

2.4.1 Set covering problem

The Set Covering Problem (SCP) concerns numerical matrices. Given a binary matrix (i.e., filled with ones or zeros) and a cost function defined on the columns

$$c : C \to \mathbb{N}$$

a column j is said to *cover* a row i when $a_{i,j} = 1$. The SCP requires to cover all the rows using a subset of columns with minimum total cost.

Table 2.2 gives and example of the problem, where solution $x' = \{c_1, c_3, c_5\} \in X$ has cost $f(x') = 19$.

	c	4	6	10	14	5	6
		0	1	1	1	1	0
		0	0	1	1	1	0
	A	1	1	0	0	0	1
		0	0	1	1	1	1
		1	1	0	0	1	0

Table 2.2: Example of the SCP.

The most natural definition for the ground set is $B \equiv C$, the set containing all columns (see also Section **??**). The feasible region includes all subsets of columns that cover all rows

$$X = \left\{ x \subseteq B : \forall i \in R \sum_{j \in x} a_{ij} \geq 1 \right\}$$

where R is the set of all rows. The objective is to minimise the total cost of the selected columns

$$\min_{x \in X} f(x) = \sum_{j \in x} c(j)$$

that is additive.

2.4.2 Interlude III: the feasibility test

This interlude deals with checking feasibility. Heuristic algorithms often require to check whether a subset of elements x is feasible or not, that is whether $x \in X$ or not: how to compute this? It's a decision problem, according to the taxonomy described earlier. Considering SAT, in order to guarantee that a solution is feasible, one has to guarantee that it is a partition of the variable set, and it is not that difficult. Considering SCP, the feasibility can be decided going through each row and summing the 1s appearing in the chosen columns: if any rows has a 0 sum, then the solution is not feasible.

In the case of the KP, the feasibility test requires to compute from the data and from subset x a single number (the total weight) and compare it with the capacity. The same occurs for the MDP, where the test requires to compute the cardinality of x and compare it with the required value. Other problems, like Max-SAT and the PMSP, require to compute and test a set of numbers (how many logical values are assigned to each variable, how many machines are assigned to each task). Finally, problems like the BPP require to compute and test several sets of numbers (how many containers are assigned to each object and what is the total volume occupied in each container).

Considering the case of a feasible solution that is slightly modified, it is interesting to understand if the feasibility of the new subset can be simply updated, instead of recomputed from scratch. Some modifications can actually be forbidden *a priori* in order to avoid infeasibility. For example, given a feasible solution of the MDP, any change in which the number of points removed is not equal to the number of points added makes the subset unfeasible. Most of the time, however, there is no way to guarantee feasibility, so that an *a posteriori* test is required, and this can be more or less expensive with respect to the complete recomputation.

2.4.3 Set packing problem

The Set Packing Problem is strongly related to the SCP, as it is still defined on a binary matrix with sets of rows and columns, and a function $\phi : C \to \mathbb{N}$ that associates a value to each column. Contrary to the SCP, in the SPP two columns j' and j'' *conflict* if they have $a_{ij'} = a_{ij''} = 1$. In other words, the columns must be "packed", without overlapping. The problem amounts to finding a subset of columns with maximum total value.

Formally, the ground set can be defined as the set of columns $B \equiv C$ (see also Section 2.7.7). The feasible region includes all subsets of nonconflicting columns

$$X = \left\{ x \subseteq B : \forall i \in R \sum_{j \in x} a_{ji} \leq 1 \right\}$$

and the objective is to maximise the total value of the selected columns

$$\max_{x \in X} f(x) = \sum_{j \in x} \phi(j)$$

and is clearly additive.

2.4.4 Set partitioning problem

The set partitioning problem (SPP) is related to both previous problems: a binary matrix is given and each column has a cost represented by a function c, but in this problem the columns

in the solutions must cover each row exactly once. The ground set is once again $B \equiv C$, and the feasible region includes all the subsets of columns that cover all rows and are not conflicting

$$X = \left\{ x \subseteq B : \forall i \in R \sum_{j \in x} a_{ij} = 1 \right\}$$

The objective function is to minimise the total cost of the selected columns

$$\min_{x \in X} f(x) = \sum_{j \in x} c(j)$$

2.4.5 Interlude IV: the search for feasible solutions

Another subproblem that is often required in the design of heuristic algorithms is that of **finding a feasible solution** $x \in X$. This is clearly a search problem. For example, exchange (or recombination) heuristics by definition start from an initial solution (or many), and this requires to find one!

Sometimes finding a feasible solution is **trivial**, because trivial subsets are feasible: a valid initial solution for the KP and the Set Packing Problem is the empty set $x = \emptyset$; a valid initial solution for the SCP is the whole ground set $x = B = C$ (provided that a feasible solution exists). Sometimes, an initial solution must only satisfy some trivial **constraint**, like $|x| = k$ for the MDP. Other times, it has to satisfy a **consistency constraint**, such as assigning each task to exactly one machine in the PMSP, each logic variable to exactly one logical value in the Max-SAT, and so on. These solution are in general nonoptimal, but they are certainly feasible and valid as a starting point.

In some other situation, it can be very hard to find even a feasible solution. That is the case for the BPP (if the number of containers is small) and the SPP. No polynomial algorithm is known to solve the problem of finding a feasible solution in these cases.

In order to find a solution when the problem is really hard, it is possible to make a **relaxation** on some constraint of the problem and enlarge the feasible region X to X'. This means that f must be defined on X', not only on X (is it possible?). Moreover, typically the additional subsets in $X' \setminus X$ tend to have better objective values, and this complicates the task of finding an optimal feasible solution.

2.5 Graph problems

2.5.1 Vertex cover problem

In the Vertex Cover Problem (VCP), given an undirected graph $G = (V, E)$, one must select a subset of vertices of minimum cardinality, such that each edge of the graph is incident to at least a vertex in the solution. The natural ground set is $B \equiv V$ (see also Section 2.7.9). The feasible region is

$$X = \left\{ x \subseteq B : \forall (i, j) \in E x \cap (i, j) \neq \emptyset \right\}$$

where we remind that edge (i, j) is a set of two vertices, so that the intersection $x \cap (i, j)$ is well defined. The objective is to minimise the number of selected vertices

$$\min_{x \in X} f(x) = |x|$$

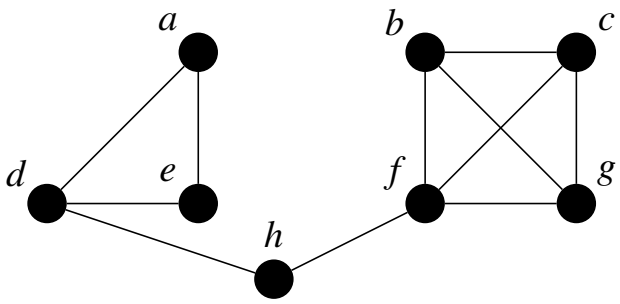


Figure 2.4: An undirected graph.

The VCP admits a weighted version, where a weight function $w : V \to \mathbb{N}$ is defined on the vertices and the objective function is the sum of the weights of the chosen vertices.

For example, Figure 2.4 shows an example of VCP instance, while Figure 2.5 shows a feasible solution $x = \{b,d,e,f,g\}$ and an unfeasible subset $x' = \{a,c,h\}$

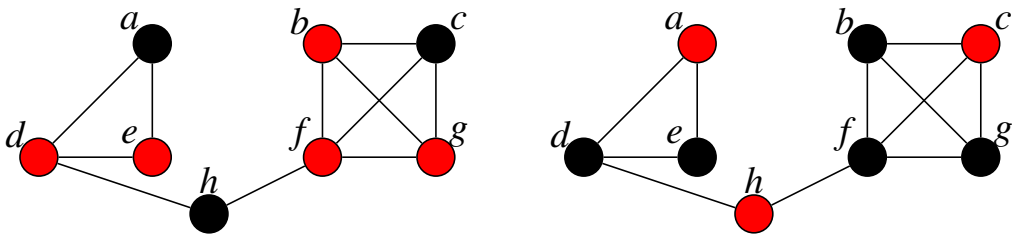


Figure 2.5: A feasible solution and an unfeasible subset for the VCP.

2.5.2 Maximum clique problem

In the Maximum Clique Problem (MCP), given an undirected graph $G = (V,E)$ and a weight function defined on the vertices $w : V \to \mathbb{N}$, one must select a subset of pairwise adjacent vertices of maximum total weight. In other words, a feasible solution is a subset of vertices such that all the pairs of vertices correspond to an edge. The MCP requires to find a feasible solution of maximum cardinality. In the weighted MCP, a weight function $w : V \to \mathbb{N}$ is defined on the vertices and the problems aims to maximise the sum of the weights of the selected vertices. In figure 2.6 the subsets $x = \{b,c,f,g\}$ and $x' = \{a,d,e\}$ are feasible solutions of value 3 and 4, while $x'' = \{b,e,f\}$ is unfeasible.

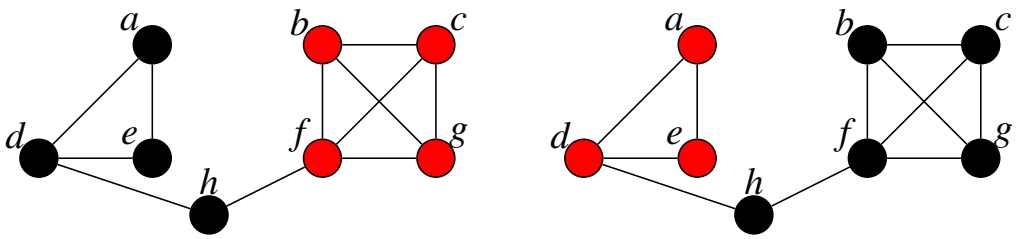


Figure 2.6: Two feasible solutions for the MCP.

Formally, then, the ground set is $B \equiv V$, the feasible region includes all subsets of pairwise

adjacent vertices

$$X = \{x \subseteq B : \forall i \in x, \forall j \in \{x \setminus \{i\}\}(i,j) \in E\}$$

and the objective is to maximise the weight of the selected vertices

$$\max_{x \in X} f(x) = \sum_{j \in x} w(j)$$

that for unweighted problems is simply the cardinality of x.

2.5.3 Maximum independent set problem

In the maximum independent set problem (MISP), given an undirected graph $G = (V,E)$ and a weight function defined on the vertices $w : V \to \mathbb{N}$, one must select a subset of pairwise nonadjacent vertices of maximum weight. This problem is kind of the opposite of the MCP: the solutions are subsets of vertices of that do not correspond to edges. For example, in Figure 2.4 subsets $x' = \{1,6\}$ and $x'' = \{1,3\}$ are feasible solutions.

The ground set is $B \equiv V$. The feasible region includes all subsets of pairwise nonadjacent vertices

$$X = \{x \subseteq B : \forall i \in x, \forall j \in \{x \setminus \{i\}\}(i,j) \notin E\}$$

and the objective is to maximise the weight of the selected vertices:

$$\max_{x \in X} f(x) = \sum_{j \in x} w(j)$$

In order to find a simple solution, one can take a singleton, e.g. $x = \{1\}$, which is indeed a feasible solution.

2.5.4 Interlude V: relations between problems

The last three graph problems are quite similar problems. Indeed, the theory of computational complexity shows that some problems can be **reduced** to other problems, that is one can "use" a problem to solve another problem. A clear example of this is offered by the MCP and the MISP: given an instance of the MCP, one can build the complement graph of the initial instance (i. e., the graph with the same vertices and the edges that link all and only the pairs of vertices not linked by edges in the original graph). Then, one can find a solution of the MISP on this graph: the vertices of this solution are pairwise nonadjacent. However, the corresponding vertices in the MCP instance are pairwise adjacent, by construction. So, they are a valid solution of the MCP. In short, the MCP can be solved using a transformation of the instance, an algorithm for the MISP and a back-transformation of the solution (that in this case is trivial, but in general could be more complex). The converse is also true: we can find a solution for a MISP instance by reducing it to an instance of a MCP. This works for optimal solutions (that remain optimal) and heuristics solutions (that remain heuristic).

$$\text{MCP} \leftrightarrow \text{MISP}$$

Less expectedly, the VCP is reducible to the SCP: given the graph that is the instance of VCP, build a binary matrix transforming each edge e_i of the graph in a row r_i of the matrix and each vertex v_j in a column c_j. For each edge e_i that is incident to a vertex v_j, set $a_{ij} = 1$ (every row, therefore, has exactly two 1's); otherwise, set $a_{ij} = 0$. For a weighted instance of the VCP, assign the cost of each vertex to the associated column; otherwise, set a uniform cost equal to 1.

Solving the SCP identifies a set of columns that cover all rows. Correspondingly, the associated vertices covering all edges. This identifies a feasible solution for the VCP. If the solution of the SCP is optimal, the same holds for the VCP; if it is heuristic, the VCP solution is simply heuristic.

The reduction is only unidirectional: it is impossible, except for special cases in which each row has exactly two 1's, to reduce in a simple way[1] a SCP instance to a VCP instance:

$$\text{VCP} \rightarrow \text{SCP but SCP} \nleftarrow \text{VCP}$$

A more refined relation exists between the BPP and the PMSP: tasks and objects are rather similar: both have a "size" (time length or volume) and both have to be *assigned* in some way (to machines or to containers). The big difference is that the machines have an unlimited "capacity" - their working time - but a given number; on the other hand, bins are unlimited in number, but limited in capacity. However, it is still possible to find a reduction.

Let us reduce a BPP instance into a PMSP instance: objects can be easily transformed into tasks, turning volumes into time lengths, and containers can be tranformed into machines ignoring the capacity, but the number of machines cannot be derived from the original data. So, we make an assumption, a "guess". Solving the PSMP instance exactly will yield an optimal solution. Interpreting the assignment of tasks to machines as the assignment of the corresponding objects to the corresponding machines, the total execution time on each machine represents the total volume of each container. If any execution time exceeds the capacity of a container, the solution is not feasible. Since it is optimal, one concludes that the number of machines assumed is insufficient. It must be increased and the solution must be recomputed. When the optimal completion time of the PMSP is not larger than the capacity of the containers, the number of machines is feasible. The smallest such number is the optimal number of containers for the BPP.

The reduction can also be performed in the opposite direction, turning a PMSP instance into a BPP instance by assuming a capacity for the containers. If the optimal number of containers is larger than the original number of available machines, then the solution is unfeasible. The smallest capacity that allows to find a number of containers not larger than the available machines is the optimal value of the BPP. It can be proved that the number of iterations of this process required to reach the optimum is polynomial.

Again, it is important to stress the fact that using a heuristic algorithm to solve the reduced instance ends up providing a heuristic solution for the original problem, because feasible solutions remain feasible, whereas declaring a value as unfeasible can be the result of a mistake (the heuristic just failed to find a solution) and therefore better solutions could still exist.

In summary, studying relations between problems is important even before starting to think about algorithms: it can even avoid the design of dedicated algorithms!

2.5.5 The travelling salesman problem

Given a directed graph $G = (N, A)$ and a cost function defined on the arcs $c : A \rightarrow \mathbb{N}$, solving the Travelling Salesman Problem (TSP) requires to find a circuit visiting all the nodes of the graph (Hamiltonian circuit) at minimum cost (see Figure 6.3). This is a very famous graph optimisation problem.

The ground set is the arc set, $B \equiv A$ (see also Section 2.7.12). The feasible region includes all the subsets of arcs that identify circuits visiting all the nodes. The objective is to minimise

[1] To be correct, since both problems are $\mathcal{NP}$-complete, a reduction in the opposite direction is actually possible in polynomial time, but far from being simple. Therefore, it is never used in practice.

the total cost of the selected arcs

$$\min_{x \in x} f(x) = \sum_{j \in x} c(j)$$

and is additive.

To determine whether a subset is a feasible solution, one has to check whether every node has exactly one arc entering it and one going out of it, but also that the selected arcs form a connected subgraph, instead of a collection of disconnected subtours.

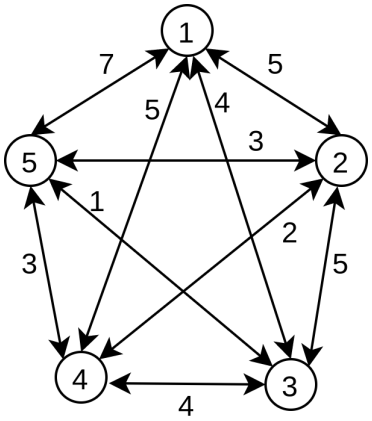


Figure 2.7: A TSP instance.

Finding a feasible solution for this problem is in general hard. In fact, checking if a graph contains a Hamiltonian circuit is a strongly NP-complete problem. In complete graphs, that are very frequent in applications, this problem is however trivial (starting from any node, go from the last visited node to a random new one until all nodes are visited, then go back to the first node).

2.5.6 Capacitated minimum spanning tree problem

Given an undirected graph $G = (V,E)$, a root vertex $r \in V$, a cost function defined on the edges $c : E \to \mathbb{N}$, a weight function defined on the vertices $w : V \to \mathbb{N}$ and a capacity, the Capacitated Minimum Spanning Tree Problem (CMSTP) requires to find the minimum cost spanning tree such that each subtree appended to the root has total weight not larger than the capacity (see Figure 2.8). Since a minimum spanning tree is a subset of edges, the ground set can be defined as $B \equiv E$. The feasible region includes all spanning trees such that the weight of the vertices spanned by each subtree appended to the root does not exceed W and the objective is to minimise the total cost of the edges selected

$$\min_{x \in X} f(x) = \sum_{j \in x} c(j)$$

Since the objective function is additive, it is quite easy to evaluate it and also to recompute it in case of small modifications of the subset of edges. It is however harder to verify the feasibility of a given set of edges: it is necessary to visit each subtree (therefore, the representation of the graph and of the tree should make a visit easy), summing the weights of vertices, in order to check if it is a feasible solution or not. If a feasible solution is modified, the tree (or at least some subtrees) must be revisited, which is much less efficient than the update of the objective function.

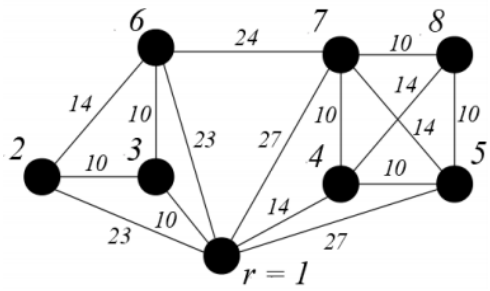


Figure 2.8: An instance of CMSTP.

Finding a capacitated spanning tree is a strongly $\mathcal{NP}$-complete problem, so it is quite hard to find a feasible solution. If the graph is complete, however, it becomes trivial.

In summary, the objective function is fast to evaluate and to update, whereas the feasibility test (acyclicity test and subtree weight computation) is not very fast to perform and to update. Moreover, it is easy to obtain subsets of edges that provide nonoptimal solutions: each subtree can easily be reoptimised changing the edges, but not the vertices. Is it possible to cope with these defects?

Changing the ground set can be helpful. In fact, the CMSTP is quite similar to the BPP: vertices with a weight must be distributed into subsets of limited capacity, just like objects with a volume must be distributed into containers of limited capacity. By introducing a set of subtrees T with $|T| = |V \setminus \{r\}|$, we could assign vertices to subtrees (possibly, leaving some subtree empty) and define the ground set as $B \equiv V \times T$. The feasible region then includes all partitions of the vertices into vertex subsets of weight less or equal to W inducing connected subgraphs on G. The objective is to minimise the sum of the costs of the subtrees that span each subset of vertices plus the edges that connect them to the root.

This representation makes the weight feasibility test easier, as it is sufficient to scan the pairs (vertex,subtree) summing the weight of each vertex to the weight of each subtree and finally check whether these respect the capacity. It is also easy to check that the vertices are partitioned: each vertex should belong to exactly one subtree. But where is the tree structure? The information about the edges is *implicit*: given the vertices of each subtree, to verify that a subtree exists requires to compute whether the induced subgraph is connected (and can be reached from the root). However, in a complete graph such a check is not necessary.

In order to compute the objective function, which is based on edges, one must compute the minimum spanning tree for each subtree identified by the vertices and add the cheapest edge that connects one of the vertices to the root. This operation has a cost larger than simply scanning the edges, as in the previous representation. On the other hand, however, this representation guarantees that the subtrees are optimal with respect to the spanned vertices: a huge number of nonoptimal solutions are implicitly cancelled.

2.5.7 Vehicle routing problem

Given a directed graph $G = (N,A)$, a "depot" node $d \in N$, a cost function defined on the arcs $c : A \to \mathbb{N}$, a weight function defined on the nodes $w : N \to \mathbb{N}$ and a capacity $W \in \mathbb{N}$, the Vehicle Routing Problem (VRP) is solved by selecting a set of minimum cost circuits that visit all nodes, starting from the depot and each having a total weight not larger than the capacity (see Figure 2.9). This is related to the TSP, but the nodes are spanned by several different subtours instead of a single one. The idea is that vehicles are used to visit customers in need, and one is not enough because the vehicles have a finite load capacity.

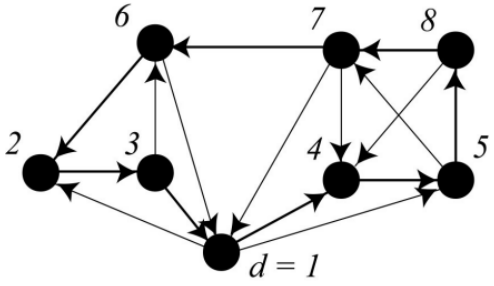


Figure 2.9: A VRP instance.

We consider two ground sets. One could be the set of arcs $B \equiv A$, the other the set of pairs (node,circuit), $B \equiv N \times C$, where C is the set of all possible circuits. The feasible region in the former case includes include all arc subsets that cover all nodes with circuits visiting the depot and whose weight does not exceed W. In the latter case, it includes all partitions of the nodes into subsets of weight not larger than W and admitting a spanning circuit.

The objective is to minimise the total cost of the selected arcs:

$$\min_{x \in X} f(x) = \sum_{j \in x} c(j)$$

The feasibility check in the former case requires a visit of the graph, in the latter a simple computation of the total weight of each circuit (if the graph is complete). Evaluating the objective function requires a simple sum of costs in the former case. The latter is much harder, as finding the arcs implicitly defined by the assignment of nodes to circuits actually requires to find a minimum cost Hamiltonian circuit, that is to solve a TSP instance. This is an $\mathcal{NP}$-hard problem, so that the idea seems to be unacceptable. In practice, it is not a very good idea, but it can be remarked that, for small capacities, the set of nodes of the auxiliary TSP instance will typically be small, so that solving the problem might actually be practical. Moreover, nothing really forbids to solve the TSP problem heuristically, instead of exactly.

2.5.8 Interlude VI: combining alternative representations

Different ground set for the same problem can have advantages and disadvantages: a ground set can be good for the feasibility checks and a different one can be good for the evaluation of the objective function. Which one to choose depends on what operations are more frequently used in the algorithms that we want to design.

It is actually possible to use both ground sets. A disadvantage is that this increases the space required to represent solutions and the time required to update them. However, if solutions are used more frequently than updated (and this is commonly the case), the burden of keeping them up to date is acceptable.

A second disadvantage is that the different representations of the solutions must be kept consistent during the algorithm.

2.6 Summary

In summary, the questions that one must ask when facing a Combinatorial Optimization problem are:

- what is the most appropriate ground set?

- are there several possible definitions?
- how to compute the objective function?
- how to update it when a solution is modified?
- how to check whether a subset of the ground set is a feasible solution?
- how to update the feasibility status when a solution is modified?
- how to find a feasible solution?
- are there useful relations between the given problem and other ones?

Always answer these questions when starting to deal with this family of problems.

2.7 Exercises

The first part of these notes and, correspondingly, the first item of the exam deal with combinatorial optimisation problems. A practitioner and, correspondingly, a successful student should be able to identify combinatorial optimisation problem, defining a reasonable ground set, and consequently to answer the main questions discussed throughout this chapter: how to distinguish feasible solutions from general subsets, how to find a feasible solution, how to compute the objective function for a given solution, what kind of function it is, and what costs are required to perform these operations. All these questions basically require to start choosing a representation for the data and the solutions.

Most of the following exercises solve these problems for the several examples provided in this chapter. The identification of the ground set is therefore already revealed in the previous sections, though in some cases alternative ground sets can and will be discussed.

Notice two points:

1. the exercises do not require the most efficient implementation possible, provided that it is always polynomial;

2. the complexities described can usually be strongly improved by adopting a more sophisticated representation of the solutions.

Indeed, the purpose of these exercises it to get a feeling of which elements of a problem influence the complexity of the basic operations that heuristic algorithms perform, and how to overcome the resulting problems. That is exactly what happens when a problem is investigated in practice.

Also notice that estimating the complexity of computing the objective function and testing a subset for feasibility requires background knowledge on worst-case asymptotic complexity that is recalled in Section 3.1. Since the practical skills required by the course on this topic are already tested here, Chapter 3 provides no numerical exercise.

2.7.1 Exercise 1

Given a set E of elementary objects, a positive integer volume v_e and a positive integer value ϕ_e for each object $e \in E$ and a positive integer capacity V, the Knapsack Problem *requires to find a subset of objects of maximum total value, such that their maximum total volume does not exceed the capacity.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution As already suggested, the most obvious ground set is the set of objects, $B \equiv E$.

How to represent it? An integer number n for its cardinality, an integer number for the capacity V and two integer vectors v and ϕ, respectively, for the volumes and values are the

most natural representations. Subsets x can be represented as incidence vectors of length n, but also as lists or vectors of elements.

Given an incidence vector[2] x, the feasibility test requires to scan it summing the volumes v_e of the elements for which $x_e = 1$. Subset x is a feasible solution if and only if the sum does not exceed the capacity. The computational time required is $\Theta(n)$.

Given a feasible solution x, it is possible to compute its value by scanning the corresponding incidence vector and summing the values ϕ_e of the elements for which $x_e = 1$. The computational time required is $\Theta(n)$. The objective function is additive, because it is a sum *on the elements of the solution*. The emphasised words are fundamental: a sum on something different from the elements of the solution is not additive!

Feasible solutions always exist: the empty solution is feasible.

2.7.2 Exercise 2

Given a set of points P, a nonnegative distance d_{ij} between each pair of points $i, j \in P$ and a positive integer $k \leq |P|$, the Maximum Diversity Problem *requires to find a subset of k points such that the sum of their pairwise distances is maximum.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution As already suggested, the most obvious ground set is the set of points, $B \equiv P$, but one could also think of considering the pairs of points $B \equiv P \times P$.

The problem can be represented by the cardinality $n = |P|$ and an integer square matrix $D = \{d_{ij}\}$ for the pairwise distances[3].

Let us consider the first ground set. Given an incidence vector x of length n, the feasibility test requires to scan it and check whether exactly k elements are equal to 1. This requires $\Theta(n)$ time. Given a feasible solution x, its value can be computed with a double loop on the elements equal to 1, summing the distances d_{ij}. If one saves such elements in a vector, the cost is $\Theta\left(k^2\right)$; otherwise, it is $\Theta(kn)$ or $\Theta\left(n^2\right)$, according to how smart the loop is. The objective function is nonadditive (quadratic, to be precise). Feasible solutions always exist: get the first k elements (or a random subset of k).

Let us now consider the second ground set. The solution x is an incidence matrix of n rows and n columns. The feasibility test requires to scan it and check whether it contains exactly a subsquare of k rows and k columns whose intersection is equal to 1. This requires $\Theta\left(n^2\right)$ time. Given a feasible solution x, its value can be computed with a double loop on the elements equal to 1, summing the distances d_{ij}. The cost is $\Theta\left(k^2\right)$, or $\Theta(kn)$ or $\Theta\left(n^2\right)$, exactly as above, depending on the exact implementation. The objective function is additive. Of course, feasible

[2]For the sake of simplicity, we give the same name to the subset and the vector, with a slight abuse of notation.

[3]Further details on the representation of data and solution, and the computational costs are discussed in the first laboratory session, in Chapter A.

solutions always exist: this is a feature of the problem, independent from its representation. Overall, the second ground set does not seem a good idea, even neglecting the space complexity.

2.7.3 Exercise 3

Given a set of elementary objects E and a set of containers C, a nonnegative integer volume v_e for each object $e \in E$ and a positive integer capacity V for the containers, the Bin Packing Problem *requires to assign each object to exactly one container so that the capacity of all containers is respected while using the smallest possible number.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The suggested ground set is $B \equiv E \times C$, that is the assignment of single objects to single containers.

The problem can be represented by the cardinalities $n = |E|$ and $m = |C|$, the integer value of the capacity V and an integer vector v for the volumes. Given an incidence matrix x of n rows and m columns, the feasibility test requires to scan it and check whether exactly one element is equal to 1 in each row and whether the total volume occupied in each container respects the capacity V. A Boolean vector f of n elements can store the information whether an object is assigned or not to any container; an integer vector w of m elements can store the volume occupied in each container. A double loop on the elements $x_{ec} = 1$ can check whether f_e is already 1 and setting it to 1 and sum the volume v_e to w_c and check whether this respects the capacity V. The overall time is $\Theta(nm)$. A better representation of the solution would use an integer vector c of n elements, such that c_e is the index of the container used by element e: the feasibility test would simply check whether c_e makes sense and sum v_e to w_{c_e}. That would take $\Theta(n+m)$ (both vectors must be initialised). The objective function can be evaluated by performing the same double loop and increasing a counter every time the volume of a container increases from 0 to a positive value. That takes the same time as above. The objective function is nonadditive. Feasible solutions always exist if there are enough containers of sufficiently large volume (for example, $m = n$ and $V = \max_{e \in E} v_e$. Otherwise, the problem can be $\mathcal{NP}$-complete.

2.7.4 Exercise 4

Given a set of tasks T, a set of machines M and a duration d_t for each task $t \in T$, the Parallel Machine Scheduling Problem *requires to assign each task to a machine so that the total completion time is minimum.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The suggested ground set is $B = T \times M$. The problem can be represented by the numbers of tasks and machines, $n = |T|$ and $m = |M|$ and an integer vector d for the durations.

Given an incidence matrix x of n rows and m columns, the feasibility test requires to scan it and check whether exactly one element is equal to 1 in each row. A Boolean vector f of n elements can store the information whether a task is assigned or not to any machine. A double loop on the elements $x_{tm} = 1$ can check whether f_t is already 1 and setting it to 1. The overall time is $\Theta(nm)$. A better representation of the solution would use an integer vector m of n elements, such that m_t is the index of the machine used to perform task t: the feasibility test would simply check whether m_t makes sense. That would take $\Theta(n)$ time. The objective function can be evaluated by defining an integer vector t of m elements to store the total operating time on each machine, performing the same double loop as above and increasing t_m by d_e whenever $x_{em} = 1$. That takes the same time as the feasibility check. The objective function is nonadditive, even if it involves sums, because it is not a sum on the ground set. Feasible solutions always exist: for example, all tasks can be assigned to a single machine.

2.7.5 Exercise 5

Given a logical formula in conjunctive normal form, composed by a product of m clauses C_j on a set V of n variables x_i, and an integer weight w_j for each sum, the Maximum Satisfiability Problem *requires to find a truth assignment to the variables that satisfies a subset of clauses of maximum weight.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The problem can be represented in several ways. A simple one is to build a matrix $A = \{a_{ij}\}$ where $a_{ij} = -1$ if the j-th clause includes literal $\bar{x}_i$, $a_{ij} = 1$ if it includes literal x_i, $a_{ij} = 0$ if it includes none of the two. Alternatively, a vector of n lists (one for each clause) could include in each list the indices i of the variables that appear as affirmed literals, and another vector of n lists could include the indices of the negated ones. Or the vectors could refer to the variables and the indices in the lists to the clauses. More refined representations can improve the average-case complexity, or the worst-case one in special classes of instances, or possibly even the worst-case complexity, but this is beyond the purpose of the exercise.

The ground set is suggested to be $B = V \times \{1,0\}$. Given an incidence matrix x of n rows and 2 columns, the feasibility test requires to scan it and check whether exactly 1 element in each row is equal to 1. This requires $\Theta(n)$ time. Given a feasible solution x, its value can be computed scanning the m clauses and for each clause scanning the variables (or the occurrences

of the literals) to check whether at least one satisfies the clause. In this case, the weight of the clause is added to a total. The overall cost is $\Theta(mn)$[4]. The objective function is nonadditive. A feasible solution always exist: give random values (or a fixed deterministic value) to all variables.

Are alternative ground sets possible? In general, they are. A set nearly equivalent to the chosen one is $B = V$, meaning that the variables in x must be set to 1 and those out of x to 0. This definition has the advantage that all subsets are feasible solutions.

Is it possible to define the ground set as the set of clauses, $B = \{C_j\}$, meaning that a solution is composed by the satisfied clauses? That would make the objective function additive, as its value would be the sum of the weights on the satisfied clauses. But the feasibility test would require to determine whether a given subset of clauses is satisfiable or not. This is the well-known $\mathcal{NP}$-complete *SAT* problem: solving it repeatedly several times would be impractical, so it is not acceptable.

2.7.6 Exercise 6

Given a set of rows R, a set of columns C, a binary matrix A with rows and columns corresponding to the two sets and a cost c_j for each column, the Set Covering Problem *requires to cover all the rows with a subset of columns of minimum total cost.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The natural ground set is $B = C$. The problem can be represented by a Boolean matrix for A and an integer vector for c. More refined representations could replace the matrix with a vector of lists associated to the rows and reporting the indices of the columns that cover each row, or a vector of lists associated to the columns and reporting the indices of the rows covered by each row, or even both at the same time.

Given an incidence vector x of $n = |C|$ elements, the feasibility test defines a vector σ of $m = |R|$ elements that count how many times each row is covered by the columns of x. Scanning x, each element with $x_j = 1$ determines the column of A which must be scanned increasing the counter σ_i for each row such that $a_{ij} = 1$. At the end, the solution is feasible if and only if $\sigma_i \geq 1$ for every row $i \in R$. With the basic representation, this takes time $\Theta(nm)$, and even the more refined representation do not improve the worst-case. The objective function can be evaluated by summing the costs for all elements with $x_j = 1$, which takes $\Theta(n)$ time. The objective function is additive. Feasible solutions exist if and only if there is no row with $a_{ij} = 0$ for all $j \in C$ or, equivalently, $x = C$ is feasible.

Can the ground set be the set of rows R? That would make no sense: given that all rows must be covered, selecting a subset of rows would have no meaning for this problem.

[4] If the number of literals in a clause is limited, the more refined representations are more efficient.

2.7.7 Exercise 7

Given a set of rows R, a set of columns C, a binary matrix A with rows and columns corresponding to the two sets and a value ϕ_j for each column, the Set Packing Problem *requires to select a subset of columns of maximum total value such that no two columns cover a common row.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The natural ground set is $B = C$. The representation of the problem is the same as for the Set Covering Problem of the previous exercise.

Given an incidence vector x of $n = |C|$ elements, the feasibility test still exploits the counter vector σ of $m = |R|$ elements, that is built as in the previous exercise, but at the end is checked to determine whether $\sigma_i \leq 1$ for every row $i \in R$. This takes time $\Theta(nm)$. The objective function is evaluated summing the values for all elements with $x_j = 1$, in time $\Theta(n)$, and it is additive. The empty solution is always feasible.

Can the ground set be the set of rows R? In general, the covered rows form a subset, but each row can be covered by several columns, and knowing the row does not specify which column covers it: neither the feasibility nor the value of a subset x can be computed.

2.7.8 Exercise 8

Given a set of rows R, a set of columns C, a binary matrix A with rows and columns corresponding to the two sets and a cost c_j for each column, the Set Partitioning Problem *requires to find a subset of columns of minimum total cost such that they cover exactly once each row.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The natural ground set is $B = C$. The representation of the problem is the same as for the Set Covering Problem and the Set Packing Problem of the previous two exercises.

Given an incidence vector x of $n = |C|$ elements, the feasibility test exploits an integer counter vector σ of $m = |R|$ elements, that is built scanning the binary matrix as in the previous exercises, but at the end is checked to determine whether $\sigma_i = 1$ for every row $i \in R$. This takes time $\Theta(nm)$. The objective function is evaluated summing the values for all elements

with $x_j = 1$, in time $\Theta(n)$, and it is additive. As discussed in the notes, it is $\mathcal{NP}$-complete to determine whether a feasible solution exists or not for a given instance.

Can the ground set be the set of rows R? The two reasons considered in the previous exercises to give a negative answer both hold in this case.

2.7.9 Exercise 9

Given an undirected graph $G = (V,E)$, the Vertex Cover Problem *requires find a minimum cardinality subset of vertices such that all edges are indicent to at least one vertex of the subset.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The natural ground set is $B = V$. A graph can be represented in several ways: the main ones are the adjacency matrix (a binary matrix of $n = |V|$ rows and columns stating whether each pair of vertices corresponds to an edge or not), the list of edges (a list of $m = |E|$ pairs of indices identifying the extreme vertices of the edges of the graph, and the vector of forward stars (a vector of n lists, associated to the vertices, each one including the indices of the vertices adjacent to the current one). The choice influences the complexity of the operations, but they can all be transformed into each other.

Given an incidence vector x of n elements, the feasibility test scans the edges of the graph and for each edge $(u,v) \in E$ checks whether $x_u = 1$ and $x_v = 1$. The list of edges allows to perform this operation in $\Theta(m)$ time. The objective function can be evaluated by counting the vertices $v \in V$ for which $x_v = 1$, in $\Theta(n)$ time. Since this is equivalent to summing all values of x, the objective function is additive. The whole vertex set V is always a feasible solution.

2.7.10 Exercise 10

Given an undirected graph $G = (V,E)$, the Maximum Clique Problem *requires find a maximum cardinality subset of vertices that are all pairwise adjacent to each other.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The natural ground set is $B = V$. We will choose a representation for the graph depending on the operations to be performed.

Given an incidence vector x of n elements, the feasibility test scans it to count the vertices and scans the edges of the graph to count the edges such that both extremes have $x_v = 1$. If $|x|$ is the number of vertices, the number of edges must be $|x|(|x|-1)/2$. This takes time $\Theta(n+m)$. Alternatively, the vector of forward stars allows to check only the edges that have one extreme with $x_v = 1$. This improves the average-case performance.

The objective function can be evaluated by counting the vertices $v \in V$ for which $x_v = 1$, in $\Theta(n)$ time. Since this is equivalent to summing all values of x, the objective function is additive. Any edge $(u,v) \in E$ is a feasible solution.

2.7.11 Exercise 11

Given an undirected graph $G = (V,E)$, the Maximum Independent Set Problem *requires find a maximum cardinality subset of vertices that are all pairwise nonadjacent to each other.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The natural ground set is $B = V$. We will choose a representation for the graph depending on the operations to be performed.

Given an incidence vector x of n elements, the feasibility test scans the edges of the graph to count the edges such that both extremes have $x_v = 1$. This number must be zero. The check takes time $\Theta(m)$ (improvable in the average case by scanning only the forward stars of the vertices with $x_v = 1$).

The objective function can be evaluated by counting the vertices $v \in V$ for which $x_v = 1$, in $\Theta(n)$ time. Since this is equivalent to summing all values of x, the objective function is additive. Any single vertex $v \in V$ is a feasible solution.

2.7.12 Exercise 12

Given a directed graph $G = (N,A)$ and a cost c_{ij} for each arc $(i,j) \in A$, the Travelling Salesman Problem *requires to find a circuit visiting all nodes with minimum total cost.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution The suggested ground set is $B = A$. We will choose a representation for the graph depending on the operations to be performed.

Given an incidence matrix x of n rows and columns, the feasibility test scans it to check that each row $i \in N$ has a single cell with $x_{ij} = 1$ and each column $j \in N$ has a single cell with $x_{ij} = 1$. While scanning the matrix, we build a vector J_i providing for each node $j \in N$ the only node j for which $x_{ij} = 1$. Then, starting from an arbitrary node i_0, we move to $i_1 = J_{i_0}$ and so on, flagging the index of each visited node in an auxiliary Boolean vector f_i, until we reach an already visited node. The subset x is a feasible solution only if all indices are flagged at the end of the visit. If the graph is not complete, one also has to check all cells with $x_{ij} = 1$ correspond to existing arcs. The overall check requires $\Theta\left(n^2\right)$ time.

The objective function can be evaluated by scanning the incidence matrix x and summing all costs for the elements with $x_{ij} = 1$. The objective function is therefore additive. In a complete graph, any matrix $x_{i\pi(i)} = 1$ for which $(\pi(1), \pi(2), \ldots, \pi(n))$ is a permutation of the nodes, is a feasible solution. As discussed in the text, in incomplete graphs, the existence of feasible solutions is an $\mathcal{NP}$-complete problem.

2.7.13 Exercise 13

Given an undirected graph $G = (V,E)$, a root vertex $r \in V$, a cost c_e for each edge $e \in E$, a weight w_v for each vertex $v \in V$ and a capacity W, the Capacitated Minimum Spanning Tree Problem *requires to find a spanning tree of minimum total cost such that the total weight of each subtree appended to the root vertex does not exceed the capacity.*[5]

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution A first possible ground set is $B = E$. A good representation for the graph is the vector of forward stars, because the subtrees appended to the root vertex will be identified by a visit. On complete graphs, however, the adjacency matrix is probably better. Correspondingly, the solution can be represented marking the elements of the forward stars that correspond to the edges of the solution; on complete graphs, a binary matrix could be appropriate.

The feasibility test must check whether the subtrees appended to the root have a weight not larger than the capacity and whether they are subtrees or cyclic subgraphs. That can be done with a visit from the root using only the edges marked, summing the weights of the vertices visited and checking that no vertex is visited twice. That takes $\Theta(m)$ time.

The objective function can be evaluated by summing all costs for the edges with $x_e = 1$, once again in $\Theta(m)$ time. The objective function is additive.

A second possible ground set is $B = V \times T$, where $T = \{1, \ldots, |V| - 1\}$ is a set of indices for potential subtrees. The vector of forward stars is still a good representation for the graph.

[5] This exercise must be revised: until then, take it as a suggestion for sophisticated problems.

Given an incidence matrix x of $n = |V|$ rows and $n-1$ columns indicating which pairs (vertex,subtree) are used, the feasibility test defines an integer vector ω of $|V|-1$ elements to store the weights of the potential subtrees appended to the root. The check scans the elements with $x_{vt} = 1$ and sums w_v to ω_t, checking whether it exceeds W or not. Then, for each index t with $\omega_t > 0$ it finds the connected components of the subgraph induced by the vertices with $x_{vt} = 1$: there must always be a single component. Overall, the test takes $\Theta(n+nm)$ time. The objective function can be evaluated by defining and building a vector t_v that provides the subtree to which each vertex v belongs (the one for which $x_{vt} = 1$) solving a minimum spanning tree problem on the edges that have both extremes in the same component. The time is that of Prim's or Kruskal's algorithm. The objective function is not additive on the (vertex,subtree) pairs.

A complete graph always admits feasible solutions. As discussed in the text, the existence of feasible solutions is in general an $\mathcal{NP}$-complete problem.

2.7.14 Exercise 14

Given a directed graph $G = (N,A)$, a depot node $d \in N$, a cost c_{ij} for each arc $(i,j) \in A$, a weight w_i for each node $i \in N$ and a capacity W, the Vehicle Routing Problem *requires to find a collection of circuits of minimum total cost such that the total weight of each circuit does not exceed the capacity and they all visit the depot.*[6]

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution A first possible ground set is $B = A$. A good representation for the graph is the vector of forward stars, but on complete graphs the adjacency matrix is probably better. Correspondingly, the solution can be represented marking the elements of the forward stars that correspond to the arcs of the solution; on complete graphs, a binary matrix could be appropriate.

The feasibility test must check whether the solution consists of circuits visiting the depot and whether these have a weight not larger than the capacity. That can be done with a visit from the depot using only the arcs marked, summing the weights of the vertices visited, checking that the circuits get back to the depot and they overall visit each node exactly once. That takes $\Theta(m)$ time.

The objective function can be evaluated by summing all costs for the arcs with $x_e = 1$, once again in $\Theta(m)$ time. The objective function is additive.

A second possible ground set is $B = N \times C$, where $C = \{1,\ldots,|N|-1\}$ is a set of indices for potential circuits. The vector of forward stars is still a good representation for the graph.

Given an incidence matrix x of $n = |V|$ rows and $n-1$ columns indicating which pairs (node,circuit) are used, the feasibility test defines an integer vector ω of $|V|-1$ elements to store the weights of the potential circuits going out of the depot. The check scans the elements with $x_{ic} = 1$ and sums w_i to ω_c, checking whether it exceeds W or not. It also checks whether there is

[6] This exercise must be revised: until then, take it as a suggestion for sophisticated problems.

always exactly one arc going in and out of every node (except for the depot), whether all circuits end in the depot and whether all nodes are visited. Overall, the test takes $\Theta(n+nm)$ time. The objective function can be evaluated by defining and building a vector c_i that provides the circuit to which each node i belongs (the one for which $x_{ic} = 1$) solving a Travelling Salesman Problem on the subgraph induced by the nodes assigned to each circuit. This is impractical, but it can be done for small instances; for large ones, it can be done heuristically, or with sophisticated tools that go beyond the scope of the course.

A complete graph always admits feasible solutions. As discussed in the text, the existence of feasible solutions is in general an $\mathcal{NP}$-complete problem.

2.7.15 Exercise 15

Given a set of points P, an integer distance d_{ij} for each pair of points $i, j \in P$ and a positive number $p < n = |P|$, the p-median problem *requires to find an assignment of each point to one of p points (including itself), such that the sum of the distances from all points to the assigned ones is minimum.*

Explain why it is a Combinatorial Optimization problem, proposing a possible ground set. Answer the following questions describing how the data are represented and how a subset x of your ground set is represented.

Suggest a procedure to evaluate whether x is a feasible solution and discuss its computational complexity.

If x is a feasible solution, suggest a procedure to compute the value of the objective and discuss its computational complexity. Is the objective function additive?

Do feasible solutions always exist? If they do, propose an easy way to compute one.

Solution A first possible ground set is $B = P \times P$, that is the set of possible assignments. A good representation for the problem is given by a square matrix of n rows and columns for the distances and an integer number p.

Given an incidence matrix x of n rows and columns, the feasibility test scans it to check that each row $i \in N$ has a single cell with $x_{ij} = 1$ and exactly p columns $j \in N$ have cells with $x_{ij} = 1$. The overall check requires $\Theta\left(n^2\right)$ time.

The objective function can be evaluated by scanning the incidence matrix x and summing all distances for the elements with $x_{ij} = 1$. The objective function is therefore additive.

Feasible solutions always exist: it is enough to select p random points as medians and assign the other ones to them at random (or, better, each one to the closest median).

A second possible ground set is $B = P$, that is the set of medians. Given an incidence vector of length n, the feasibility test scans it to check that exactly p elements have $x_i = 1$. This takes $\Theta(n)$ time.

The objective function can be evaluated by scanning the points and for each of them scanning the incidence vector x to find the selected median that is closest to the point. The objective function is not additive, because it is not a sum of values associated to the elements of the solution.

Feasible solutions always exist: it is enough to select p random points as medians.

Part II

Algorithm analysis

CHAPTER 3

Theoretical efficiency

The second part of the course is dedicated to the *features* of heuristic algorithms: we previously described heuristic algorithms as algorithms that do not always provide correct solutions but are characterized by two good aspects: their cost is much lower than an exact algorithm and they "frequently" yield something "close" to the correct solution. We will consider these two aspects, costs and quality, meaning *distance* and *probability* of achieving a certain quality, from two points of view: an *a priori* analysis, based on theory, and *a posteriori*, based on evidence and empirical data gained from the execution of the algorithm on a benchmark dataset.

3.1 Cost as computational complexity

This section deals with basics notions of **computational complexity**, that should be common background for computer science students.

3.1.1 Problems

We've seen that a problem is a question on some system of mathematical objects. Usually, there are infinitely many systems that are very similar to each other. For example, one could wonder whether 7 is a prime number, but also whether 10 is, 50 is, and so on. These are not intrinsically different problems. Instead, they are different *instances* of the same problem. So, in formal terminology, a problem is the **function** which relates each instance I in a set $\mathcal{I}$ of all instances with the corresponding solution S in the set $\mathcal{S}$ of all solutions.

$$P : \mathcal{I} \to \mathcal{S}$$

Defining a function doesn't mean to know *how* to compute it. This knowledge is expressed by an **algorithm**.

3.1.2 Algorithms

An algorithm is a **formal procedure** composed of elementary steps forming a finite sequence. These steps are determined completely by the input of the algorithm and by the results of the previous steps. An algorithm is said to *solve* a certain problem if, given in input an instance $I \in \mathcal{I}$ of that problem, it returns in output the solution $S \in \mathcal{S}$ corresponding to the instance. Thus, an algorithm is a function relating instances and solutions:

$$A : \mathcal{I} \to \mathcal{S}$$

which seems identical to the definition of the problem. However, the algorithm defines both a function and the way to compute it. In particular, an algorithm is **exact** if its associated function coincides with the problem itself. So, for every instance it provides the "correct" solution. Otherwise, it is **heuristic**. A heuristic algorithm is useful if it is **efficient** and **effective**.

3.1.3 Cost of a heuristic algorithm

The cost of a heuristic algorithm is the computational cost in space (the memory used) and in time (the time necessary for the computation to conclude). Usually, in the definition of complexity, the importance of time is more stressed than that of space. First, space is a renewable resource, while time definitely is not. Second, using an amount of space nearly always requires to use at least the same amount of time. Finally, it is technically easier to distribute the use of space than of time. Space and time are *partly* interchangeable, as it is possible to reduce the use of one by increasing the use of the other.

The time required to solve a problem depends on several aspects: the specific instance to solve, the algorithm used, the machine running the algorithm and so on. What is needed is a measure that allows to make comparisons. First of all, the measurement should be **independent from the technology** used, to avoid changing the measure each time the machine used changes. Second, it must be **concise**, that is summarized in a symbolic expression. Third, it should be **ordinal**, to allow comparing different algorithms. The computational time in seconds for all possible instances of a problem violates all these requirements.

3.1.4 Worst-case asymptotic time complexity

The classic definition adopted is the worst-case asymptotic time complexity. This replaces the physical time with the number of steps performed by an algorithm to transform the input into the output. The abstract concept of **elementary operation**, that takes exactly one step, must be defined: usually, sums, differences, products, divisions, comparisons, read and write operations are considered as elementary. The time is therefore defined as the number T of elementary operations performed, expressed as a function of the size of an instance. This is a suitable value n, for example the number of elements of the ground set, variables of clauses of the CNF, nodes or arcs of the graph and so on. What matters is the **worst-case**, the maximum value of T on all instances of size n

$$T(n) = \max_{I \in \mathcal{I}_n} T(I) \quad \text{with } n \in \mathbb{N}$$

that is **approximated** from above and/or below with a simpler function $f(n)$ considering only their asymptotic behaviour for $n \to +\infty$.

The Θ functional space

The number of operations required to solve an instance of size n in the worst case is

$$T(n) \in \Theta(f(n))$$

when, formally,

$$\exists c_1 \exists c_2 \exists n_0 : \forall n((n \geq n_0) \implies (c_1 f(n) \leq T(n) \leq c_2 f(n)))$$

with c_1 and c_2 two positive real constants and n_0 a natural number. In words, one can find two real positive coefficients and a natural number such that the computational time is restricted

between $c_1 f(n)$ and $c_2 f(n)$ for all values of n greater than n_0. Asymptotically, $f(n)$ estimates $T(n)$ up to a multiplying factor: for large instances, the computational time is at least and at most proportional to the values of $f(n)$.

The O functional space

Considering only the approximation from above, one gets the Big-O functional space:

$$T(n) \in O(f(n))$$

which formally means that

$$\exists c \exists n_0 : \forall n((n \geq n_0) \implies (T(n) \leq cf(n)))$$

with $c \in \mathbb{R}^+$ and $n \in \mathbb{N}$. Asymptotically, $f(n)$ overestimates $T(n)$ up to a multiplying factor; in other words, there is some real constant c such that $c \cdot f(n)$ is always bigger than $T(n)$ after n_0.

The Ω functional space

Considering only the approximation from below, one gets the Ω functional space:

$$T(n) \in \Omega(f(n))$$

formally means that

$$\exists c \exists n_0 : \forall n((n \geq n_0) \implies (T(n) \geq cf(n)))$$

with $c \in \mathbb{R}^+$ and $n \in \mathbb{N}$. Asymptotically, $f(n)$ underestimates $T(n)$ up to a multiplying factor; in other words, there is some real constant c such that $c \cdot f(n)$ is always smaller than $T(n)$ after n_0.

Combinatorial optimization problems always admit an exact algorithm that is **exhaustive**: in the worst case, it considers *all* possible subsets of the ground set, $x \in 2^B$. Each subset must be tested for its feasibility, taking time $\alpha(n)$, and the value of the objective $f(x)$ must be computed, taking time $\beta(n)$. Consequently, the time complexity of the exhaustive algorithm is

$$T(n) \in \Theta(2^n(\alpha(n) + \beta(n)))$$

which is **at least** exponential and most of the time impractical. In combinatorial optimization, the main distinction is between polynomial complexity and exponential complexity. In general, heuristic algorithms are polynomial.

3.1.5 Transformations and reductions

We have already seen that problems can be transformed into other problems: take any instance I_p of problem P, transform it into an instance I_Q of problem Q (using some time), solve it with an algorithm A_Q to obtain S_Q, transform the result back into S_P (using some time). This can be done in a single iteration or in several ones. If transforming the instance and transforming back the solution take polynomial time, and if the number of iterations required by the transformation is polynomial (or constant, such as 1), then the complexity of the overall algorithm mainly depends on the complexity of A_Q[1].

[1]Of course, the transformation increases the complexity, but the point is that, if A_Q is polynomial, then the resulting algorithm remains polynomial.

3.2 Going beyond worst-case complexity

The standard worst-case definition of complexity has many disadvantages: it considers only instances of large size, and only very hard instances. Small instances are neglected. Easy instances are also ignored. We would like to know if a given instance is "easy" or not. Secondly, the worst-case complexity gives an estimate which is usually meaningful, but in general overly rough. There are algorithms for which the estimate is nearly always so excessive to become useless. An example is the simplex algorithm for the Linear Programming problem. The instances of this problem are infinitely many, but only a finite set of basic solutions are relevant. The simplex algorithm explores them with exponential worst-case complexity. However, for an extremely large majority of instances, the complexity of this algorithm is a polynomial of very low complexity.

What can we do to go beyond the standard definition of worst-case complexity?

There are two ways to escape from the limitations of the classical description of the complexity of an algorithm:

- **Parameterised complexity**, that is to introduce some other relevant parameter k, besides n, and express the time as $T(n,k)$, which may lead to the understanding that the algorithm is hard with respect to k, but actually easy with respect to n;
- **Average-case complexity**, that is to assume a probability distribution on $\mathcal{I}$ and express the time as the expected value

$$T(n) = E[T(I)|I \in \mathcal{I}_n]$$

In the first case, k can be as large as n, implying that the problem is hard, but k can be small also for very large instances, meaning that the problem is easy *for those instances*. In the second case, the probability distribution weighs not only the hard instances, but also the easy ones; if the hard instances are very rare, their weight will have a small impact on the overall measure.

3.2.1 Parameterised complexity

Some algorithms are actually not exponential with respect to n, but with respect to some other parameter k. If both k and n are large, then obviously the algorithm is inefficient, but if k is small, then the algorithm will be cheaper also for large instances, possibly even polynomial.

The parameter k can be a part of the input, like a capacity, the maximum number of literals per clause in logic function problems, the number of nonzero elements in numerical matrix problems, the maximum degree, the diameter in graph problems and so on. In these cases, it is possible to know *a priori* whether the algorithm is efficient on a given instance.

In other cases, the additional parameter k is part of the solution: then, one cannot know a priori if the algorithm is efficient or not. However, an estimate of k could be provided by heuristic solutions, and this could provide the required information.

Example: the vertex covering problem

Given a graph $G = (V,E)$, the VCP requires to find a minimum subset of vertices that cover every edge of the graph. The exhaustive algorithm for this problem tests each of the 2^n subsets of vertices to check whether it covers all edges; for such subsets, it computes the cardinality and saves the smallest one. This algorithm has time complexity

$$T(n,m) \in \Theta(2^n(m+n))$$

where $n = |V|$ and $m = |E| \leq n(n-1)/2$. If we already know a solution x with $f(x) = |x| = k+1$, we restrict the search to solutions of at most k vertices. Consequently, a naive algorithm consists in scanning all subsets of exactly k vertices, instead of all subsets. This algorithm has time complexity

$$T(n,m,k) \in \Theta(n^k m)$$

where factor n^k approximates $\binom{n}{k}$. If k is a small constant, the algorithm is polynomial. But we can do better.

Bounded tree search

A better algorithm can be based on the following useful property:

$$\forall x \in X, \forall (u,v) \in E \; x \cap (u,v) \neq \emptyset$$

that is the definition of a feasible solution: in words, they must include at least one vertex for each edge. The bounded tree search algorithm selects an edge (u,v). Since either u or v (or both) are contained in the solution, the problem can be split into two subproblems: the first has $u \in x$ and the second has $u \notin x$ and $v \in x$. In both cases, to check if the subproblem is feasible, it is necessary to compute

$$V = V \setminus x \;\text{ and }\; E = E \setminus \{e \in E : e \cap x \neq \emptyset\}$$

removing all the vertices in the partial solution and all the covered edges. If $|x| < k$ and $E \neq \emptyset$, select another edge $(u,v) \in E$ and split the subproblem. If $|x| \leq k$ and $E = \emptyset$ then x is the required solution. If $|x| = k$ and $E \neq \emptyset$ the subproblem has no solution better than $k+1$.

The complexity is $T(n,m,k) \in \Theta(2^k m)$, polynomial in n and much more efficient than the naive algorithm for $n \gg 2$.

Kernelization

The concept of **kernelization** is a special case of problem transformation: instead of reducing the instances of the problem to instances of another one, kernelization transforms the instances of P into smaller instances of P. To this purpose, it exploits properties that allow to prove the existence of optimal solution that include certain elements of B or do not include some elements of B. So, kernelization works by removing from B elements that are not needed, without affecting the quality of the solution. In this way, one obtains smaller instances. Sometimes, the final size of the instance no longer depends on n, and an exhaustive algorithm can solve it in polynomial time. Even if that is the case, kernelization can still be useful, because it generates smaller problems, on which a heuristic algorithm will be faster and probably better. In principle, the kernelization itself could be heuristic, meaning that the elements excluded or fixed could be chosen heuristically, instead of being provably optimal.

Kernelization of the VCP The useful property exploited for the VCP is that if the degree of vertex v is $\delta_v \geq k+1$, then vertex v belongs to any feasible solution of cardinality $\leq k$. This is proved by contradiction: assuming that a vertex v of large degree ($\geq k+1$) does not belong to the required solution means that the $k+1$ incident edges must be covered in another way. The only way is to use the adjacent $k+1$ vertices, generating a solution of cardinality $\geq k+1$. By contradiction, vertex v must belong to any solution of cardinality $\leq k$. So, the steps of the kernelization algorithm are as follows:

1. start at step $t = 0$ with $k_0 = k$ and an empty vertex subset $x_t := \emptyset$
2. set $t = t + 1$ and add to the solution the vertices of degree $\geq k_t + 1$:

$$\forall v \in V \ \delta_v \geq k_t + 1 \Rightarrow x_t := x_{t-1} \cup v$$

3. update $k_t := k_0 - |x_t|$
4. remove the covered edges, the vertices of zero degree and the vertices in x:

$$E := \{e \in E : e \cap x_t = \emptyset\} \text{ and } V := \{v \in V : \delta_v > 0\} \setminus x_t$$

5. if $|E| > k_t^2$ there is no feasible solution; if $|E| \leq k_t^2 \Rightarrow |V| \leq 2k_t^2$, apply the exhaustive algorithm.

The kernelization algorithm starts assuming a value $k+1$ for $|x|$ (guess or heuristic). All vertices with degree $\geq k+1$ are added to the solution x. Now, the vertices to be found are $k - |x|$, that is less than k. The updated value of k becomes the new threshold for the degree of vertices. This can imply that a new batch of vertices is added to the solution. After "cleaning" the vertex and edge sets, the algorithm compares the the number of remaining edges with the current value of k^2. If if it is $> k^2$, the remaining edges cannot be covered with less then k vertices, because all remaining vertices have degree $\geq k$. This means that no solution is feasible. If, on the contrary, the number of remaining edges is $\leq k^2$, there are at most $2k_t^2$ vertices. The kernelization procedure, followed by the exhaustive algorithm, have a complexity $T(n,k) \in \Theta(n+m+2^{2k^2}k^2)$, which is very good if k is small.

3.2.2 Average case complexity

Instead of characterising the worst-case complexity of an algorithm using secondary parameters, we can replace it with the **average-case** complexity, that computes the expected value of $T(n)$:

$$T(n) = E[T(I)|I \in \mathcal{I}_n]$$

instead of the maximum $\max_{I \in \mathcal{I}_n} T(n)$.

This requires to define a **probability distribution** on the instance, which means in turn defining a suitable probabilistic model, which is far from being an easy task. In the context of theoretical studies, this means that a theoretical probability distribution model is introduced, and theorems are proved to compute the time complexity of the algorithm on instances following that model. This is interesting, and can be useful when it can be found empirically that instances follow some kind of distribution. In the context of empirical studies, a simulation model can be derived evaluating the probability distribution of measured data. This allows to generate other realistic instances at random to test the algorithms. We now survey some examples of probabilistic models for the main families of problems in combinatorial optimisation.

Probabilistic models for random binary matrices

There are three main probability models with respect to the generation of a binary matrix. We neglect the generation of cost vectors and value vectors.

Equiprobability In the **equiprobability** model, all 2^{mn} possible matrices of m rows and n columns are assigned the same probability to be selected.

Uniform probability In the **uniform probability model**, each cell of a matrix is set to 1 with a given probability p

$$\forall i = 1, \cdots, m; \forall j = 1, \cdots, n \;\; P(a_{ij} = 1) = p$$

This model is more general than equiprobability, as it allows to assign different probabilities to sparse or dense matrices.

Fixed density The last model is the **fixed density model**, where a parameter $\delta \in [0,1]$ is defined and out of the mn cells, $\delta \cdot mn$ are extracted with uniform probability and set to 1. It resembles the uniform probability, but in this case the number of 1s is fixed exactly.

Probabilistic models for graphs

Equiprobability Since a graph can be represented through its adjacency matrix, the probability models for graphs correspond to those for matrices. In the **equiprobability model**, all $2^{\frac{n(n-1)}{2}}$ graphs have the same probability to be selected.

Uniform probability In **Gilbert's model** (or **uniform probability model**) the probability $G(n,p)$ is defined as

$$\forall (i \in V, j \in V \setminus i) \;\; P[(i,j) \in E] = p$$

All graphs with the same number of edges m have the same probability $p^m(1-p)^{\frac{n(n-1)}{2}-m}$, different for each m. If $p = 0.5$, this model is the same as the equiprobability.

Erdős-Rényi Finally, in the **Erdős-Rényi model** the probability $G(n,m)$ is defined as follows: extract m unordered vertex pairs out of the $\frac{n(n-1)}{2}$ with uniform probability and create an edge for each one. If $p = \frac{2m}{n(n-1)}$, it resembles the uniform probability model, but the number of edges is fixed.

Probabilistic models for logic functions (random CNF)

A CNF is a conjunction of disjunctions (an AND of ORs). Logical clauses have a certain number n of variables and a given number of literals k for each disjunction (called logical clause) - let us assume k is the same for each logic clause.

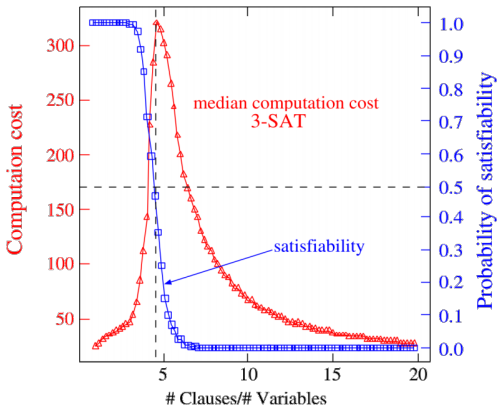
Fixed probability The **fixed-probability ensemble model** lists all $\binom{n}{k}2^k$ (there are $\binom{n}{k}$ ways to dispose n variables in groups of k, and for each group there are 2^k possible truth assignment to its literals) clauses of k distinct and consistent literals and adds each one to the CNF with probability p.

Fixed size The **fixed-size ensemble model** builds m clauses, adding to each one k distinct and consistent literals, extracted with uniform probability p. The process is somewhat opposite to the previous one, as it starts from a fixed number of clauses and constructs them with variable size. The fixed-probability model assumed a fixed size, but a variable number of clauses. If $p = \frac{m}{\binom{n}{k}2^k}$, the two models resemble each other, but the fixed size model cannot generate the same set of instances.

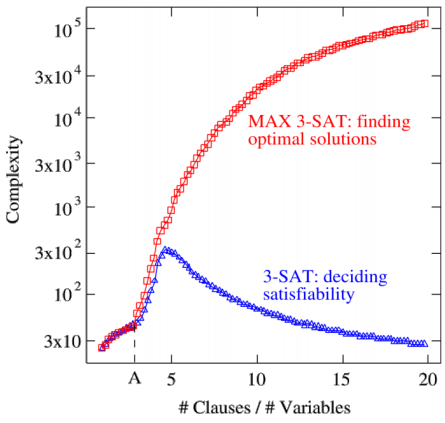
3.2.2.1 Phase transitions

The concept of **phase transition** is a practical insight coming from the empirical study of problems. It has been observed that the instance set can be divided into different regions, characterised by different values of deterministic or probabilistic parameters. The instances in some of these regions require a much longer time, whereas those in others prove easier. For example, for graphs, $m = 0$ and $p = 0$ correspond to empty graphs, $m = \frac{n \cdot (n-1)}{2}$ and $p = 1$ correspond to complete graphs and intermediate values correspond to graphs of intermediate density, deterministically if one sets m and probabilistically if one sets p. Sometimes, an algorithm has strongly different performance on sparse and on dense graphs. This concerns the computational time (for both exact and heuristic algorithms) and the quality of the solution (for heuristic algorithms). The interesting point is that quite often the performance varies steeply and abruptly in small regions of the parameter space: this phenomenon is similar to the phase transition of physical systems, for example when ice melts into water, which happens at a very strict temperature and physical zone. It is not only interesting, but also allows to analyse a given instance and determine *a priori* in which "region of complexity" it lies.

Example: phase transition in 3-SAT and MAX-3-SAT The first example of phase transition concerns the satisfiability problem. Given a CNF on n variables, with logical clauses containing 3 literals, the 3-SAT (which was proved to be $\mathcal{NP}$-complete by Karp) is solved by finding a truth assignment that satisfies all clauses. The MAX-3-SAT is solved by finding the maximum number of satisfiable clauses. So, the parameters are the number of variables n and the number of literals for each disjunction, which is 3. Another important parameter is $\alpha = \frac{m}{n}$, the ratio of clauses to variables, where m is the number of clauses in the CNF.



(a) Complexity VS Probability of Satisfiability in 3-SAT.



(b) Complexity of the decision vs the optimization problem.

Figure 3.1: Phase plots.

As figure 3.1 (a) shows, for low values of α the probability of a positive solution (the CNF is satisfiable) is high and the computational cost of finding this answer is low. But as α increases (so there are more clauses or less variables), the problem becomes harder to solve and its probability to be satisfied drops - in fact, having lots of clauses and few variables quickly induces some contradiction. Figure 3.1 (b), instead, plots the computational costs of the two version of the problem, showing the cost of the 3-SAT problem in blue and that of Max-3-SAT in red. The computational time is in logarithmic scale, this time. The optimization version (Max-3-SAT) is harder than the decision one (SAT), because it is not sufficient to say that the

CNF is unsatisfiable: one must find how many of the clauses are satisfiable at most.

The practical value of this information is that it is sufficient to calculate α in order to know approximately where the instance lies, and how long an algorithm will take to solve it. Notice that these results refer to a single specific algorithm, but experience often shows that they extend to large families of algorithms, and therefore they highlight a property of the problem itself.

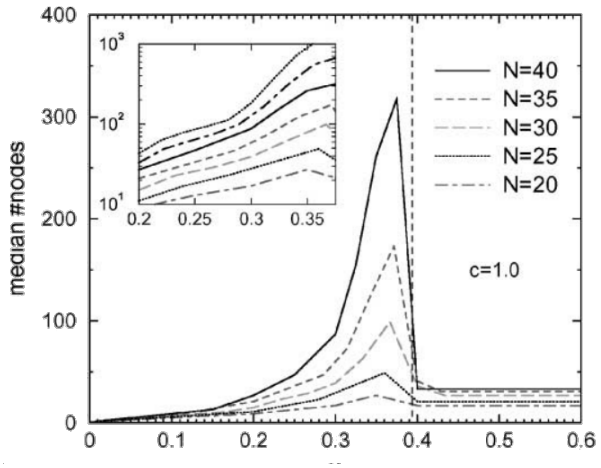


Figure 3.2: VCP Phase.

Example: phase transition in the vertex covering problem The VCP exhibits a somewhat similar phase transition, as Figure 3.2 shows. This time, the critical parameter is $\frac{|x|}{|V|}$, the ratio of the cardinality of the solution to the total number of vertices in the graph. This is an *a posteriori* value, which cannot be evalued in advance, since x is unknown. If the ratio is small, a small number of vertices is enough to cover all the edges: these vertices can be found quickly. As the solution starts to be around 30% of the total number of vertices, finding them is increasingly difficult as the size of the instance increases. If the number of vertices in the solution becomes larger, around 40% of the total vertices, it will become easier to find them, because most of them are evidently necessary in any feasible solution of a good quality.

In partial contradiction to what discussed so far, the time complexity of a heuristic algorithm is usually **strictly polynomial** with low exponents and it is fairly robust with respect to secondary parameters. If the algorithm uses random steps or memory (that is, if the heuristic is a *metaheuristic*) the complexity is not even clearly defined, as a basic procedure is repeatedly applied, obtaining a different solution at each iteration. While the computational time is well defined for each single iteration, in theory, a metaheuristic could run indefinitely. In practice, it is terminated by a user-defined condition. So, the previous concepts mainly apply to exact algorithms.

However, there are many reasons to discuss them in a course on heuristics. First and foremost, to guide the search for the correct algorithm: an exact algorithm can be efficient in a specific case and inefficient in the worst ones. Secondly, to show that exact and heuristic algorithms can interact proficuously, as the latter can provide information to improve the former. Third, to show that kernelization is not restricted to exact algorithms, but it also improves heuristic ones, making them more efficient and more effective. Lastly, to show that the harder instances of a problem can be identified *a priori*.

CHAPTER 4

Theoretical effectiveness

A heuristic algorithm is **effective** if it "frequently" gives a solution "close" to an exact (that is, optimal) one. We will now discuss the meaning of "frequently" and "close".

We can formally discuss this concept by introducing the concept of *distance* of a solution to the optimal one and *frequence* (probability) of "hitting" a solution within a given distance from an optimal one. We can do this analysis in two ways, just as in the case of efficiency: via a **theoretical analysis** (a priori) or via an **experimental analysis** (a posteriori).

4.1 A measure of distance from the optimum

The **effectiveness** of a heuristic optimization algorithm A is related to the **difference** between the "heuristic value" $f_A(I)$ (the objective function value of the result of the algorithm A on the instance I) and the optimum $f^*(I)$ (the best solution existing for I). The relation can be specified in several different ways.

Absolute difference The most direct definition is the absolute value of the difference itself:

$$\widetilde{\delta}_A(I) = |f_A(I) - f^*(I)|$$

While this definition is natural and obvious, it is rarely used in practice, as it depends on the unit of measure of the objective function. Suppose that we are minimizing the overall cost of a circuit in the TSP, measuring it in days of travel time. You could choose hours, or minutes, or seconds, instead, and the value of $\widetilde{\delta}_A(I)$ would strongly depend on this choice. This definition makes sense only when the objective function is a pure number.

Relative difference A second definition relates the difference to the value of the optimum:

$$\delta_A(I) = \frac{|f_A(I) - f^*(I)|}{f^*(I)}$$

This is quite frequent in experimental analysis and it has the advantage to be a pure number, also when the objective is a physical quantity with a unit of measure.

Approximation ratio A third, widely used, possibility is the approximation ratio:

$$\rho_A(I) = \max\left[\frac{f_A(I)}{f^*(I)}, \frac{f^*(I)}{f_A(I)}\right] \geq 1$$

The expression holds both for minimization and maximization problems: in the former case, $f^*(I) < f_a(I)$; in the latter, the opposite is true. It is clearly related to the relative difference:

$$\delta_A(I) = \begin{cases} \rho_A(I) - 1 \text{ for minimisation problems} \\ \dfrac{\rho_A(I) - 1}{\rho_A(I)} \text{ for maximisation problems} \end{cases}$$

4.2 Theoretical analysis: approximation guarantees

In order to guarantee a priori the performance of an algorithm, the idea is once again to consider the worst case, exactly as for the efficiency.

4.2.1 Absolute and relative approximation

In general, the value $f_A(I)$ returned by a heuristic may be very bad with respect to the optimal value $f^*(I)$. If the heuristic is good, however, the difference will not be very large. For example, there might be an **absolute approximation**:

$$\forall I \in \mathcal{I}\ \exists \widetilde{\alpha}_A \in \mathbb{N} : \widetilde{\delta}_A \leq \widetilde{\alpha}_A$$

meaning that the absolute distance is somewhat limited by an integer constant. This is very rare, even for problems whose objective value is a pure number. An example is Vizing's algorithm for the edge coloring problem.

More frequently, a **relative approximation** might be possible:

$$\forall I \in \mathcal{I}\ \exists \alpha_A \in \mathbb{R}^+ : \rho_A(I) \leq \alpha_A$$

meaning that the approximation ratio $\rho_A(I)$ has an upper bound limited by the real constant α_A.

If any of the previous holds, the algorithm provides an **approximation guarantee**, absolute or relative. An interesting thing is that the definition can be extended to cases in which a constant approximation guarantee cannot be found, by replacing the approximation constant with a suitable function of the instance size:

$$\forall I \in \mathcal{I}_n \forall n \in \mathbb{N}\ \rho_A(I) \leq \alpha_A(n)$$

where $\alpha_A : \mathbb{N} \to \mathbb{R}^+$.

It is important to remark that, while the efficiency of an algorithm necessarily has a size-dependent expression, its effectiveness may have an independent one.

4.2.2 How to obtain an approximation guarantee

We can now discuss a general abstract strategy to prove that an algorithm has an approximation guarantee, provided that the guarantee exists. We aim to prove that $\rho_A \leq \alpha_A$, that is (for minimization problems)

$$\forall I \in \mathcal{I}\ \exists \alpha_A \in \mathbb{R}^+ : f_A(I) \leq \alpha_A f^*(I)$$

The first step to do that is typically to analyse the problem and design a procedure that generates a lower bound $LB(I)$

$$\forall I \in \mathcal{I}\ LB(I) \leq f^*(I)$$

that is an underestimate of the objective function. Once this value is found, the second step is to design a procedure that builds a feasible solution, whose value $UB(I)$ is by definition (being feasible) an overestimate of the optimum, but is also related to $LB(I)$ by a coefficient α_A

$$\forall I \in \mathcal{I}\ UB(I) = \alpha_A LB(I)$$

A possible third and final step is to design a procedure whose solution is not worse than $UB(I)$

$$\forall I \in \mathcal{I}\ f_A(I) \leq UB(I)$$

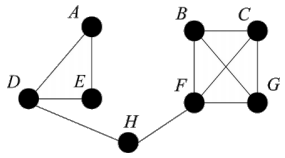
Then, the concatenation of these three procedures provides an algorithm such that

$$\begin{aligned} \forall I \in \mathcal{I}\ f_A(I) \leq UB(I) &\implies \forall I \in \mathcal{I} \alpha LB(I) \leq \alpha f^*(I) \\ &\implies \forall I \in \mathcal{I} f_A(I) \leq \alpha_A f^*(I) \end{aligned}$$

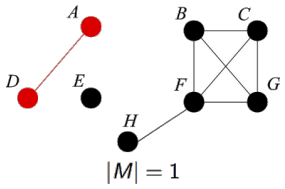
The trickiest part is usually the second step, so let us delve into it by showing an example.

A 2-approximated algorithm for the VCP

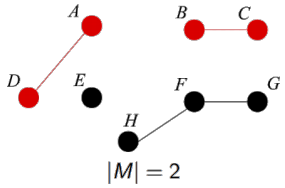
Given an undirected graph $G = (V, E)$, we aim to find the subset of vertices of minimum cardinality such that each edge of the graph is incident to it. We define a set of nonadjacent edges as **matching**, and a matching such that any edge outside of it is adjacent to one of the edges inside of it (so that it cannot be enlarged) as **maximal matching**. Notice that a maximal matching is not necessarily *maximum*, as there can be maximal matchings of different cardinality.



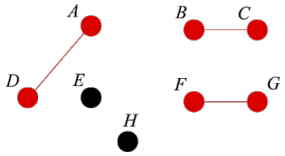
(a) VCP instance.



(b) Matching 1.



(c) Matching 2.



(d) Matching 3.

Figure 4.1: VCP instance and matchings.

We can solve the VCP with the following *matching algorithm*:

1. build a maximal matching $M \subseteq E$ scanning the edges of E in any arbitrary order and including in M those not adjacent to any edge already in M
2. the set of extreme vertices of the edges in the matching is a VCP solution

$$x_A := \bigcup_{(u,v) \in M} \{u, v\}$$

3. it can be improved removing the redundant vertices.

Proof. The first step of the algorithm provides a lower bound $LB(I)$ on the optimum $f^*(I)$. Consider the graph $G' = (V, M)$ made of all vertices of V and only the edges in any maximal matching M. A covering of E is always also a covering for any subset of edges $E' \subseteq E$. Therefore, the cardinality of an optimal covering for E' does not exceed that of an optimal covering for E

$$E' \subseteq E \implies |x^*_{E'}| \leq |x^*_E|$$

But any optimal covering for $E' = M$ includes exactly *one* of the extreme vertices for each edge, and therefore has cardinality $|M|$.

$$\begin{aligned} \forall M\ |M| \leq |E| &\implies |x^*_M| \leq |x^*_E| \implies |M| \leq |x^*_E| \\ &\Rightarrow LB(I) = |M| \leq |x^*_E| = f^*(I) \end{aligned}$$

Figure 4.1 gives a numerical example: the matching algorithm finds a maximal matching M with 3 edges. To cover all the edges in M, exactly $|M| = 3$ vertices are necessary and sufficient, as each edge must be and is covered independently from the other ones. The VCP on the whole graph has an optimal solution $f^* = 5$ (5 vertices can cover all its edges), that is larger.

$$LB(I) = |M| \leq f^*(I)$$

The second step of the algorithm includes in a solution *both* extreme vertices for each edge in the matching, so that $UB(I) = 2 \cdot |M|$ (in the example, 6 edges). This set of vertices is actually a feasible solution for the VCP, because it clearly covers all edges in M, but it also covers all the other edges in $E \setminus M$. In fact, by the definition of maximal matching, such edges are adjacent to those of the matching, that is are incident to the vertices of the matching. Consequently, those vertices cover the whole edge set. But the cost of such a solution is twice the lower bound. The third step just removes redundant vertices, keeping a feasible solution while possibly reducing its cardinality.

$$f_A(I) \leq UB(I) = 2 \cdot |M| = 2 \cdot LB(I) \leq 2 \cdot f^*(I)$$

So, we have found both an upper and a lower bound, and they are related. Hence the thesis.

$$\forall I \in \mathcal{I}\ f_A(I) \leq 2 f^*(I) \therefore \alpha_A = 2$$

□ □

Model-based heuristics This short section is an addendum not included in the topics required for the exam. It can be interesting to notice that the heuristic described above does not actually fall into any of the three categories of solution-based heuristics listed in Section 1.3.1: it does not build a subset of the ground set (vertices) progressively adding elements; it does not exchange elements between the current solution and its complement; it does not recombine two or more solutions. Actually, even if it works on the elements of the Combinatorial Optimisation problem, it is implicitly guided by manipulations to a hidden "model". The *VCP* admits the following Mathematical Programming formulation:

$$\begin{aligned} \min & \sum_{v \in V} x_v \\ & x_u + x_v \geq 1 && (u,v) \in E \\ & x_u \in \{0,1\} && u \in V \end{aligned}$$

where the binary variable x_v is equal to 1 when the solution includes vertex v and equal to 0 otherwise.

This formulation can be relaxed (admitting new feasible solutions) by replacing the integrality constraints $x_v \in \{0,1\}$ with weaker nonnegativity constraints $x_v \geq 0$ for all $v \in V$, which yields a Linear Programming problem. This admits the following dual problem:

$$\begin{aligned} \max \sum_{(u,v)\in E} y_{uv} & \\ \sum_{(u,v)\in \Delta_u} y_{uv} \leq 1 & \qquad u \in V \\ y_{uv} \geq 0 & \qquad (u,v) \in E \end{aligned}$$

which can be interpreted as a Maximum Matching problem, where the y_{uv} variables (necessarily in $[0,1]$) indicate whether an edge is selected or not. Any heuristis solution of the dual problem (a maximal matching) underestimates its optimum, that is equal to the optimum of the original primal problem, that underestimates the optimum of the original unrelaxed *VCP*. This shows more directly the relation between the two problems.

A 2-approximated algorithm for the TSP under the triangle inequality

Consider the TSP with the additional assumptions that graph $G = (N,A)$ is complete and the cost function c is symmetric and satisfies the triangle inequality:

$$\forall i, j \in N \; c_{ij} = c_{ji}$$

and

$$\forall i, j, k \in N \; c_{ij} + c_{jk} \geq c_{ik}$$

This problem is, in general, strongly $\mathcal{NP}$-complete. Under the previous assumptions, it is still hard, but it admits a heuristic algorithm with an approximation guarantee.

This algorithm is known as the *double-tree algorithm*:

1. consider the complete undirected graph corresponding to G (i. e., replace each pair of opposite arcs with a single edge)
2. build a minimum cost spanning tree $T^* = (N, X^*)$
3. make a pre-order visit of T^*, building two lists of arcs:
   (a) x lists the arcs used both by the visit and the backtracking: this is a circuit visiting each node, possibly several times
   (b) x' lists the arcs linking the nodes in pre-order ending with the first: this is a circuit visiting each node exactly once

As for the VCP, the algorithm finds a lower bound on the optimum, a feasible solution that provides an upper bound with a value related to the lower bound by a multiplication factor 2 and a potential improvement of the upper bound. This means that the final solution is within a certain factor of the optimum. In short, the **double-tree algorithm is 2-approximated**.

Proof. Given any Hamiltonian circuit, deleting an arc yields a Hamiltonian path, that is necessarily cheaper. A Hamiltonian path spans all nodes, therefore it corresponds to a spanning tree in the corresponding undirected graph. Usually, it is not a spanning tree of minimum cost. So, a minimum spanning tree is surely a lower bound on the cost of any Hamiltonian circuit, even the optimal one. So, $LB(I) = c(X^*)$. List x contains both the arcs that correspond to each edge

of X^*, and by symmetry both have the same cost as the edge. Therefore, $c(x) = 2LB(I)$. The Hamiltonian circuit obtained connecting in sequence the nodes visited in pre-order has a cost $c(x') \leq c(x)$ thanks to the triangle inequality. In fact, for any path $A \to \ldots \to Z$, the edge $A \to Z$ that directly connects its extreme vertices has a cost not larger than the sum of the total cost of the path. Hence, $UB(I) = c(x') \leq 2LB(I)$.

$$\forall I \in \mathcal{I}\ f_A(I) \leq UB(I) \leq 2LB(I) \leq 2f^*(I)$$

□ □

4.2.3 Tight approximation bounds

An interesting question now is whether there is actually a subset $\bar{I}$ of instances such that

$$f_A(\bar{I}) = \alpha_A f^*(\bar{I})$$

that is, if the approximation guarantee is *tight* or just an upper bound that could possibly be improved by a more precise analysis. If "the bound is tight" (this is the expression commonly used to state the property), the study of the instances of $\bar{I}$ could evaluate whether they are rare or frequent, explain the weak points of the algorithm and possibly introduce *ad hoc* modifications to improve it.

4.2.4 Inapproximability

Not all problems admit approximated algorithms of polynomial complexity (of course, all combinatorial optimisation problems can be solved exactly, so they can also be approximated, but usually in exponential time). The concept of an **inapproximable problem**, however, is rather complicated, as it is currently impossible to rule out definitely the existence of an approximation guarantee for a problem, due to the unsolved $\mathcal{P}$ versus $\mathcal{NP}$ question.

"Inapproximability" is a short way to say that a problem cannot be approximated unless $\mathcal{NP}$-complete problems can be solved exactly in polynomial time, a possibility that is left open for future research. In that case, the approximation guarantee would be $\alpha = 1$ for all problems of $\mathcal{NP}$. Such a thing is considered extremely unlikely to happen, but a proof is still missing. Showing that a problem is "inapproximable", therefore, amounts to proving that the existence of an approximation guarantee would imply the possibility to solve in polynomial time an $\mathcal{NP}$-complete problem.

For example, consider the family of TSP instances on complete graphs with cost function:

$$c_{ij} = \begin{cases} 0 & \forall (i,j) \in A_0 \\ 1 & \forall (i,j) \in (N \times N) \setminus A_0 \end{cases}$$

that violates the triangular inequality. If any solution of zero cost ($f(I) = 0$) is found, it means that the solution is a Hamiltonian circuit made only of edges from A_0. So, the noncomplete graph $G(N, A_0)$ has a feasible Hamiltonian circuit. If the optimal solution has a positive cost, then A_0 does not include any Hamiltonian circuit. In short

$$\begin{cases} f^*(\bar{I}) = 0 & \text{if } A_0 \text{ contains a Hamiltonian circuit} \\ f^*(\bar{I}) \geq 1 & \text{otherwise} \end{cases}$$

This means that, given any noncomplete graph with arc set A_0, one could build a "completion" with the cost function described above, solve the TSP and determine whether the original graph contains or not a Hamiltonian circuit. In short, one could solve the TSP in decision form on a general graph in polynomial time. This is a strongly $\mathcal{NP}$-complete problem and the existence of a polynomial algorithm implies that $\mathcal{P}$ versus $\mathcal{NP}$ an unlikely result that scholars have been trying to prove or disprove since the Seventies. This concerns an exact polynomial algorithm. What about an approximated one?

Assuming that such an approximation algorithm could exist, it would yield a solution of cost

$$\forall I \in \mathcal{I} f_A(I) \leq \alpha f^*(I)$$

but when the graph has a Hamiltonian circuit of zero cost, necessarily the approximation algorithm would yield a solution of cost

$$f^*(\bar{I}) = 0 \iff f_A(\bar{I}) = 0$$

In other words, if the approximation algorithm finds a zero cost solution, it proves that a zero cost Hamiltonian circuit exists; when it does not find any, it proves that such a circuit does not exist, because otherwise the approximation guarantee would be violated. Consequently, also a polynomial approximation algorithm solves the decision version of the TSP on general graphs, proving that $\mathcal{P} = \mathcal{NP}$.

4.2.5 Approximation schemes

On the bright side, sometimes it is possible to find several approximation guarantees. Exact algorithms, such as the exhaustive one, provide the best possible approximation guarantee $\alpha_A = 1$, usually in exponential time T_A. An approximated algorithm usually provides a worse guarantee ($\alpha_a > 1$) requiring a shorter time T_A.

Instead of a single polynomial algorithm with time T_A and guarantee α_A, there may be a complete family of different compromises between efficiency and effectiveness, that improve the approximation guarantee while increasing the computational complexity. Such a family of algorithms is called an **approximation scheme**, and can be seen as a parametric algorithm A_α, that allows to choose the guarantee α. An example of this situation is the Knapsack Problem.

4.3 Beyond worst-case approximation

Just as the efficiency, the effectiveness of an algorithm can be measured without resorting to the worst-case approach. An algorithm could have a very bad worst-case performance, with no approximation or an approximation $\alpha = 1\,000\,000$, but in practice yield nearly always an optimal solution. How can we describe this situation? There are two alternative approaches similar to those used for the complexity and an additional third one.

Parametrisation Instead of dividing the instances only by size, other parameters k_i can be identified and an approximation guarantee depending on k_i (not constant) can be found.

Average-case Assuming a probabilistic model for the instances, the expected value of the approximation factor can be proved to respect a certain threshold. In this case, the algorithm could perform badly on single instances, but on average it will not.

4.3.1 Randomised approximation

Metaheuristics differ from classical heuristics in that they use randomisation or memory. The former mechanism is implemented by feeding the algorithm not only with an instance of the problem, but also with a *random seed* (typically an integer number). This is used in a pseudorandom number generator to generate other numbers, which will be used to take decisions inside the algorithm, together with the data of the problem. Changing the random seed, a different sequence of pseudorandom numbers will be generated, and a different solution will be obtained. Instead of considering the instance of the problem as the outcome of a random experiment, one can investigate the stochastic distribution of the results with respect to the random seed. In particular, the quality of the solution becomes a random variable, whose properties can be studied.

So, a randomised algorithm is run not on an instance I, but on a pair (I,ω), where ω is the random seed, obtaining a value $f_A(I,\omega)$. The approximation ratio is a function of ω as well ($\rho_A(I,\omega)$), so both are random variables. The expected value of the approximation ratio can be limited by an approximation guarantee:

$$\forall I \in \mathcal{I}\ E[\rho_a(I,\omega)] \leq \alpha_A$$

Each single run of the algorithm can violate the approximation α_A, but running it algorithm several times, the average of the approximation ratio will tend to converge to a value not larger than the guarantee. Of course, the best approximation ratio will become smaller than the guarantee faster than the average.

Randomisation could have an impact also on the computational time $T(I)$, but in general this is not really meaningful: while the decisions taken and their results strongly depend on the pseudorandom numbers, the number of steps typically is much less influenced.

A randomised approximation algorithm for Max-SAT

Given a CNF, we aim to assign values of truth to the logical variables so as to maximise the total weight of the logical clauses that are satisfied. A very simple randomised algorithm assigns a random value to each variable with equal probabilities, $\frac{1}{2}$ for true and $\frac{1}{2}$ for false. What is the expected value of the final solution?

Let $\mathcal{C}_x \subseteq \{1,\cdots,m\}$ be the subset of clauses satisfied by solution x. The objective value $f(x) = f_A(I,\omega)$ is the total weight of the clauses in $\mathcal{C}_x$ and its expected value is

$$E[f_A(I,\omega)] = E[\sum_{i\in\mathcal{C}_x} w_i] = \sum_{i\in\mathcal{C}} (w_i \cdot \mathbb{I}(i \in \mathcal{C}_x))$$

where $\mathbb{I}(i \in \mathcal{C}_x)$ is the indicator function of condition $i \in \mathcal{C}_x$, equal to 1 when the condition is true, to 0 when it is false. The properties of the expected value allow to show that

$$E[f_A(I,\omega)] = \sum_{i\in\mathcal{C}} (w_i \cdot \mathbb{I}(i \in \mathcal{C}_x)) = \sum_{i\in\mathcal{C}} (w_i \cdot Pr[i \in \mathcal{C}_x])$$

where the probability $Pr[i \in \mathcal{C}_x]$ is estimated as follows. Suppose that formula $i \in \mathcal{C}$ has k_i literals, and let $k_{min} = \min_{i\in\mathcal{C}} k_i$ be the minimum number of literals over all clauses. The probability of satisfying a certain clause $i \in \mathcal{C}$ is

$$Pr[i \in \mathcal{C}_x] = 1 - \left(\frac{1}{2}\right)^{k_i}$$

which can be minorised by a term independent from i

$$1 - \left(\frac{1}{2}\right)^{k_i} \geq 1 - \left(\frac{1}{2}\right)^{k_{min}}$$

This means that

$$E[f_A(I,\omega)] \geq \sum_{i \in \mathcal{C}} w_i \cdot \left[1 - \left(\frac{1}{2}\right)^{k_{min}}\right] = \left[1 - \left(\frac{1}{2}\right)^{k_{min}}\right] \sum_{i \in \mathcal{C}} w_i$$

and since $k_{min} \geq 1$ and $\forall I \in \mathcal{I}\ f^*(I) \leq \sum_{i \in \mathcal{C}} w_i$ one can obtain[1]

$$E[f_A(I,\omega)] \geq \left[1 - \left(\frac{1}{2}\right)\right] f^*(I) = \frac{1}{2} f^*(I)$$

Notice that this is an approximation guarantee on the average: in each single run, we could be very unlucky, even for a long sequence of runs. However, in a sufficient number of runs (that are very fast, due to the simplicity of the algorithm), the sample mean will tend to approach the theoretical expected value. And the best value in the sample will be better than that!

[1]Very strictly speaking, $E[f_A(I,\omega)] \geq f^*(I)/2$ implies that $f^*(I)/E[f_A(I,\omega)] \leq 2$, not that $E[\rho_A(I,\omega)] = E[f^*(I)/f_A(I)] \leq 2$, but the main message of this section is that approximation guarantees of some sort can be obtained for an average with respect to repeated runs of a randomised algorithm.

4.4 Exercises

4.4.1 Exercise 1

A heuristic algorithm A is applied to an instance I of a minimization problem and finds a solution of cost $f_A(I) = 107$. Assuming that the optimum value is equal to $f^*(I) = 100$, compute the absolute difference $\tilde{\delta}_A(I)$, the relative difference $\delta_A(I)$ and the approximation ratio $\rho_A(I)$.

Solution For minimization problems, the absolute difference is the absolute value of the difference between the value of the solution found by the heuristic and the value of the optimal solution:

$$\tilde{\delta}_A(I) = f_A(I) - f^*(I) = 107 - 100 = 7$$

The relative difference is the ratio between the absolute difference and the value of the optimum:

$$\delta_A(I) = \frac{f_A(I) - f^*(I)}{f^*(I)} = \frac{107 - 100}{100} = 0.07 = 7\%$$

The approximation ratio is the ratio between the value of the solution found by the heuristic and the value of the optimal solution:

$$\rho_A(I) = \frac{f_A(I)}{f^*(I)} = \frac{107}{100} = 1.07$$

4.4.2 Exercise 2

A heuristic algorithm A is applied to an instance I of a maximization problem and finds a solution of cost $f_A(I) = 195$. Assuming that the optimum value is equal to $f^*(I) = 200$, compute the absolute difference $\tilde{\delta}_A(I)$, the relative difference $\delta_A(I)$ and the approximation ratio $\rho_A(I)$.

Solution For minimization problems, the absolute difference is the absolute value of the difference between the value of the optimal solution and the value of the solution found by the heuristic:

$$\tilde{\delta}_A(I) = f^*(I) - f_A(I) = 200 - 195 = 5$$

The relative difference is the ratio between the absolute difference and the value of the optimum:

$$\delta_A(I) = \frac{f^*(I) - f_A(I)}{f^*(I)} = \frac{200 - 195}{200} = 0.025 = 2.5\%$$

The approximation ratio is the ratio between the value of the optimal solution and the value of the solution found by the heuristic:

$$\rho_A(I) = \frac{f^*(I)}{f_A(I)} = \frac{200}{195} \approx 1.026$$

4.4.3 Exercise 3

A heuristic algorithm A is applied to an instance I of a minimization problem, whose optimal solution has a cost equal to $f^*(I) = 30$. Assuming that the approximation ratio is equal to $\rho_A(I) = 1.2$, compute the relative difference, the absolute difference and the value of the solution obtained by the algorithm.

Solution For minimization problems, the relative difference is simply related to the approximation ratio:

$$\delta_A(I) = \frac{f_A(I) - f^*(I)}{f^*(I)} = \frac{f_A(I)}{f^*(I)} - 1 = \rho_A(I) - 1 = 0.2$$

Therefore, the absolute difference is:

$$\tilde{\delta}_A(I) = f_A(I) - f^*(I) = \delta_A(I) f^*(I) = 0.2 \cdot 30 = 6$$

and the value obtained by the algorithm is

$$f_A(I) = f^*(I) + \tilde{\delta}_A(I) = 30 + 6 = 36$$

4.4.4 Exercise 4

A heuristic algorithm A is applied to an instance I of a maximization problem, whose optimal solution has a cost equal to $f^*(I) = 50$. Assuming that the approximation ratio is equal to $\rho_A(I) = 1.25$, compute the relative difference, the absolute difference and the value of the solution obtained by the algorithm.

Solution For maximization problems, the relative difference is related to the approximation ratio:

$$\delta_A(I) = \frac{f^*(I) - f_A(I)}{f^*(I)} = 1 - \frac{f_A(I)}{f^*(I)} = 1 - \frac{1}{\rho_A(I)} = 1 - \frac{1}{1.25} = \frac{1}{5} \approx 0.2$$

Therefore, the absolute difference is:

$$\tilde{\delta}_A(I) = f^*(I) - f_A(I) = \delta_A(I) f^*(I) = 0.2 \cdot 50 = 10$$

and the value obtained by the algorithm is

$$f_A(I) = f^*(I) - \tilde{\delta}_A(I) = 50 - 10 = 40$$

4.4.5 Exercise 5

A heuristic algorithm A for a minimization problem has a constant approximation guarantee equal to $\alpha_A = 1.5$. Assuming that it is applied to an instance I whose optimal solution has a value equal to $f^*(I) = 80$, compute the maximum possible value of the solution obtained by the algorithm.

Can the algorithm obtain the optimal solution for instance I?

Has the algorithm a constant absolute approximation?

Solution By definition, a constant approximation guarantee $\alpha_A = 1.5$ implies that

$$\rho_A(I) = \frac{f_A(I)}{f^*(I)} \leq \alpha_A = 1.5 \quad \texttt{for all } I$$

Therefore, for the specific instance considered:

$$f_A(I) \leq \alpha_A f^*(I) = 1.5 \cdot 80 = 120$$

However, $f_A(I)$ could be better, and even optimal:

$$f_A(I) \in [f^*(I), \alpha_A f^*(I)] = [80, 120]$$

A constant absolute approximation would require

$$f_A(I) - f^*(I) \leq \tilde{\alpha}_A \quad \texttt{for all } I$$

but

$$f_A(I) - f^*(I) \leq (\alpha_A - 1) f^*(I) \quad \texttt{for all } I$$

that is not in general a constant (unless $f^*(I)$ is limited).

4.4.6 Exercise 6

A heuristic algorithm A for a maximization problem has a constant approximation guarantee equal to $\alpha_A = 2$. Assuming that it is applied to an instance I obtaining a solution of value $f_A(I) = 500$, compute the possible values of the optimum.

Solution By definition, a constant approximation guarantee $\alpha_A = 2$ implies that

$$\rho_A(I) = \frac{f^*(I)}{f_A(I)} \leq \alpha_A = 2 \quad \texttt{for all } I$$

Therefore, for the specific instance considered:

$$f^*(I) \leq \alpha_A f_A(I) = 2 \cdot 500 = 1000$$

Since, of course, the optimum cannot be worse than $f_A(I) = 500$, its possible values fall in the range:

$$f^*(I) \in [f_A(I), \alpha_A f_A(I)] = [500, 1000]$$

4.4.7 Exercise 7

The following table reports the result obtained by algorithm A on a benchmark $\bar{\mathcal{I}}$ of 8 instances of a minimization problem. For the sake of simplicity, assume that all instances have optimal value equal to 100.

$\bar{\mathcal{I}}$	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
f_A	104	100	105	120	101	106	101	110

Discuss the approximation properties of algorithm A, in particular whether A is exact, 1.1-approximated or 2-approximated.

Solution Algorithm A is certainly not exact, as several results are strictly worse than the optimum. The worst approximation ratio on the benchmark is

$$\max_{I \in \bar{\mathcal{I}}} \rho_A(I) = \max_{I \in \bar{\mathcal{I}}} \frac{f_A(I)}{f^*(I)} = 1.2$$

so that A cannot be 1.1-approximated.

It could be 2-approximated, or have any approximation guarantee $\alpha_A \geq 1.2$. Proving it requires to know the results on the instances not belonging to the benchmark, or to know that the benchmark includes the instance with the worst approximation ratio overall.

4.4.8 Exercise 8

The following table reports the result obtained by algorithm A on a benchmark $\bar{\mathcal{I}}$ of 8 instances of a minimization problem. For the sake of simplicity, assume that all instances have optimal value equal to 100.

$\bar{\mathcal{I}}$	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
f_A	120	100	105	180	130	160	100	110

Discuss the approximation properties of algorithm A, in particular whether A is exact, $3/2$-approximated or 2-approximated.

Solution Algorithm A is certainly not exact, as several results are strictly worse than the optimum. The worst approximation ratio on the benchmark is

$$\max_{I \in \bar{\mathcal{I}}} \rho_A(I) = \max_{I \in \bar{\mathcal{I}}} \frac{f_A(I)}{f^*(I)} = 1.8$$

so that A cannot be $3/2$-approximated.

It could be 2-approximated, or have any approximation guarantee $\alpha_A \geq 1.8$. Proving it requires to know the results on the instances not belonging to the benchmark, or to know that the benchmark includes the instance with the worst approximation ratio overall.

4.4.9 Exercise 9

The following table reports the result obtained by algorithm A on a benchmark $\bar{\mathcal{I}}$ of 8 instances of a maximization problem. For the sake of simplicity, assume that all instances have optimal value equal to 10.

$\bar{\mathcal{I}}$	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
f_A	10	10	5	8	0	9	10	7

Discuss the approximation properties of algorithm A, in particular whether A is exact, $3/2$-approximated or 2-approximated.

Solution Algorithm A is certainly not exact, as several results are strictly worse than the optimum. The worst approximation ratio on the benchmark is

$$\max_{I \in \bar{\mathcal{I}}} \rho_A(I) = \max_{I \in \bar{\mathcal{I}}} \frac{f^*(I)}{f_A(I)} = +\infty$$

In other words, there is no maximum. No approximation guarantee exists in this case: on instance I_5, algorithm A provides the worst possible solution.

CHAPTER 5

Empirical performance evaluation

The *a priori* analysis of the effectiveness of an algorithm is usually much more complicated than that of its efficiency, for several reasons. First, the steps of an algorithm have an immediate relation to its cost, but not to the quality of its result. The average-case analysis (or the analysis of randomised algorithms) requires statistical tools, that are usually far from being simple. Even if the theoretical analysis is useful, its results may be not very representative in practice, if the assumptions are not met by the actual instance considered: for example, if the worst case is not representative of what actually happens, the analysis will give an approximation much worse than what is observed in practice. This is rarer for the computational cost.

The **experimental analysis** of the efficiency and the effectiveness of an algorithm, that is of the computational time (not space, usually) and of the quality of the result, is performed by choosing a benchmark of instances and measuring quantitatively the performance of the algorithm on such instances.

5.1 Introduction to experimental analysis

The **experimental method** is the typical approach that is used in science. Mathematics is an exception, as it is more based on formal approaches, but algorithms are *an exception in the exception*, as only some properties of the performance of an algorithm can be proved and interesting information can be extracted also from their practical performance. The experimental analysis of algorithms is an empirical island in a sea of theoretical formalism!

The experimental approach consists in *observing* the reality and making some assumption on how the reality "works" by formulating a model. Then, a sequence of steps is repeated until a **satisfactory model** is obtained:

1. design computational experiments to validate the model
2. perform the experiments and collect their results
3. analyse the results with quantitative methods
4. revise the model based on the results

5.1.1 Models

What is a *model* in the context of algorithms? In physics, a model is a law that rules the behaviour of phenomena. By analogy, in algorithmics a model is a law that is supposed to rule the behaviour of an algorithm. Such laws may concern the relation between the computational

complexity and the size of the instances, or some other parameter, such as the maximum degree of the vertices of the graph and so on. These assumptions are made on the basis of the knowledge of the algorithm, but also of a *trend* exposed by the empirical measurements.

Experimental analysis aims to obtain compact **indices** of efficiency and effectiveness of an algorithm, in order to sort and compare them by some measure of quality. Also, experimental analysis can be used to describe the relation between the indices and the parametric values of the instances (such as the size n etcetera) and to suggest improvements on the algorithms.

5.1.2 Benchmarks

The main point in the experimental evaluation of an algorithm is the use of a **benchmark sample**. Usually, problems are made of infinitely many instances, so not all of them can be tested and a meaningful subset of them has to be chosen instead: this subset is the benchmark sample. "Meaningful" means that the chosen instances should represent all the different features that could be relevant for the laws that rule the algorithm. For example, a benchmark sample should include instances of different sizes because that is obviously interesting in terms of the computational complexity and possibly in terms of the quality of the solution, as well. Other structural features of the instances that may be relevant are: for graphs, the density, degree, diameter, connectivity and a lot of other indices; for matrices, the density, the ratio between rows and columns; for logical functions the number of literals per clause, the ratio between the number of clauses and variables, and so on. Another important point is that instances usually come from "different sources" which can influence their characteristics: some arise from real-world examples, others from random, artificial or transformative generation, and others from a probabilistic distribution assumed *a priori*.

An important and frequently overlooked aspect is that it would seem natural to introduce a concept of **equiprobable benchmark**, meaning that the instances are extracted from the overall set assigning them a uniform probability. However, this approach makes no sense, because infinite sets **do not admit equiprobability**. This is not a course on statistics, but an intuitive reason is that any finite benchmark sample has a largest instance (in term of size), and therefore leaves totally unrepresented the infinite subset of instances that are larger than all the instances in the benchmark. On the contrary, we can impose equiprobability on finite classes of instances. Additionally, benchmark instances should not be *too easy* to solve as the aim is to learn something when solving them, but also not *too hard* because solving them would take a lot of time, and therefore severely reduce the number of instances. Finally, they should have a structure that matches a practical application, if one exists and is of interest.

Reproducibility The scientific method requires **reproducible** and **controllable** results. After an experiment is performed, other people should be to repeat the experiment and obtain the same results. So, it is necessary to use publicly available instances (or generate new instances and make them publicly available), to specify all implementation details, the programming language and the compiler used to implement algorithm, and technological aspects such as the machine used, the OS and the available memory.

5.1.3 Comparing heuristic algorithms

A basic principle in the comparison of algorithms is that an algorithm is better than another one only if it obtains betters results **while** requiring smaller time. Slow algorithms with good results and fast algorithms with bad results cannot be compared in a meaningful way. The

computational time can be neglected in some specific situation, such as when no comparison is performed or when the computational time for the algorithms compared is intrinsically more or less the same for structural reasons- For example, the two algorithms could be the same but for different values of numerical parameters, or for the use of secondary procedure that use a negligible part of the time.

5.1.4 Statistical models of performance

The performance of an algorithm is described with a **statistical model**. The idea is to model the execution of an algorithm as a random experiment, whose results depend on the specific *outcome* extracted from a *sample space*. The word "outcome" is used here in the statistical sense, as the elementary situation arising from the random extraction. In the simpler models, the outcome is the specific instance $I \in \mathcal{I}$ or the specific finite benchmark sample of instances

$$\overline{\mathcal{I}} \subset \mathcal{I}$$

on which the algorithm is run. Considering the case of a single instance, the results of the random experiment are the computational time $T_A(I)$ and the relative difference $\delta_A(I)$. Both are random variables that describe the performance of algorithm A. Instead of considering the values of computational time and relative difference on all possible instances of the problem, we will describe the statistical properties of the random variables $T_A(I)$ and $\delta_A(I)$, obtaining a statistical description of the performance of the algorithm.

Notice that describing $\delta_A(I)$ requires to know the value of the optimum for I, $f^*(I)$. If possible, an exact algorithm should be used to compute it. If this is not possible (because the problem is computationally hard and the instances too large), the optimum $f^*(I)$ can be estimated from below or above, trying to achieve in both cases the best possible estimate. Let $LB(I)$ denote an underestimate and $UB(I)$ an overestimate of the optimum:

$$LB(I) \leq f^*(I) \leq UB(I)$$

Considering the reciprocal values:

$$\frac{1}{LB(I)} \geq \frac{1}{f^*(I)} \geq \frac{1}{UB(I)}$$

Multiplying by the value $f_A(I)$ obtained by algorithm A on instance I, and subtracting 1:

$$\frac{f_A(I)}{LB(I)} - 1 \geq \frac{f_A(I)}{f^*(I)} - 1 \geq \frac{f_A(I)}{UB(I)} - 1$$

which is strictly related to the relative difference. In fact:

$$\frac{f_A(I)}{f^*(I)} - 1 = \begin{cases} \delta_A(I) \text{ (minimization)} \Rightarrow \dfrac{f_A(I) - UB(I)}{UB(I)} \leq \delta_A(I) \leq \dfrac{f_A(I) - LB(I)}{LB(I)} \\ -\delta_A(I) \text{ (maximization)} \Rightarrow \dfrac{UB(I) - f_A(I)}{UB(I)} \leq \delta_A(I) \leq \dfrac{LB(I) - f_A(I)}{LB(I)} \end{cases}$$

and therefore

$$\frac{|f_A(I) - UB(I)|}{UB(I)} \leq \delta_A(I) \leq \frac{|f_A(I) - LB(I)|}{LB(I)}$$

which means that, even if the relative difference is unknown, a range of possible values can be determined[1].

[1] Extended to a whole benchmark of instances, this range yields a region estimate for the diagrams described in the following.

5.2 A posteriori efficiency evaluation

5.2.1 Run time distribution diagram

We first describe the performance with respect to the computational time. A basic descriptive tool is the so-called **Run Time Distribution** (RTD) diagram, that plots the distribution function of the time $T_A(I)$ required to solve an instance I in the benchmark sample $\overline{\mathcal{I}}$ (ideally, a diagram referring to the whole instance set $\mathcal{I}$ could be defined, but never drawn in practice, for problems with an infinite set of instances).

This diagram plots the cumulative distribution function of $T_A(I)$:

$$\forall t \in \mathbb{R}\ F_{T_A}(t) = Pr[T_A(I) \leq t]$$

that is the probability that the random variable $T_A(I)$ (the time used to solve instance I with algorithm A) has a value not larger than a parametric value t for all possible t. Of course, this diagram depends on the specific benchmark considered, but becomes more and more meaningful for the whole problem as the benchmark becomes more representative. Since the computational time is always positive, the probability is 0 for all $t \leq 0$; as t grows, the probability grows from 0 to 1, because the number of solved instances cannot decrease giving more time to the algorithm. When the time is enough to solve also the hardest instance of the benchmark, the probability reaches 1.

Figure 5.1 shows an example, that deserves some discussion to understand the practical meaning of this diagram.

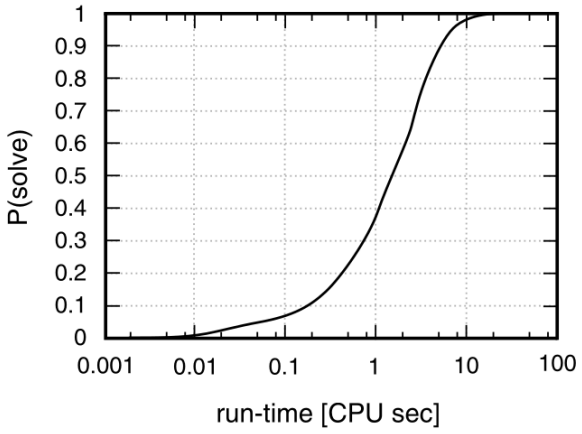


Figure 5.1: An example of RTD diagram.

This profile raises from a zero probability of solving an instance in a time ≤ 0.001s (a thousand of a second) to probability 1 for a time slightly larger than 10s. So, on the given benchmark the computational time is between 0.001s and slightly more than 10s. The vertical coordinate for $t = 1$s, that is 0.35, is the fraction of instances solved by the algorithm in a time not larger than one second. So, about one third ($\approx 35\%$) of the instances are solved in at most one second, while about two thirds are solved in a time between one second and around twenty seconds.

Notice that the time axis is in logarithmic scale: regularly spaced times are obtained multiplying the previous time by ten. In fact, the computational time ranges on a huge set of values (three orders of magnitude).

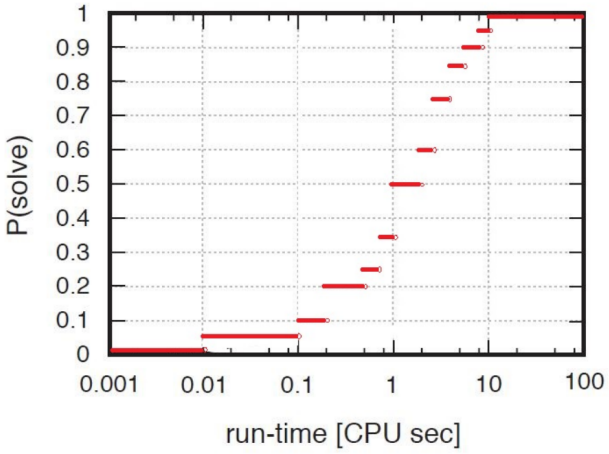


Figure 5.2: Construction of an RTD plot.

Of course, the computational time strongly depends on the size of the instances. So, it makes little sense to draw such a diagram on a benchmark including instances of very different sizes. In that case, in fact, it is rather obvious that small instances will be solved quickly and large instances will be solved slowly. The diagram is not adding any useful information: it describes the structure of the benchmark rather than the algorithm itself. The solution is to draw the diagram for benchmark samples $\overline{\mathcal{I}}_n$ with *fixed* n (for example, graphs with exactly 1000 vertices). If the diagram spreads over a wide range of computational times even for a uniform instance size, this is more interesting. Some other parameter could be influencing the efficiency (such as the density of the graphs), and the benchmark sample could be constructed by fixing that parameter as well. If all influential parameters are identified and fixed, all instances are solved requiring the same time, and the RTD diagram degenerates into a step function. Alternatively, the computational time of the algorithm could actually be strongly variable, depending on very specific factors. This is not very common for heuristics, but typically occurs for exact algorithms of exponential complexity, in which even minor aspects can be magnified by the combinatorial explosion of the number of solutions.

Properties and construction of the RTD diagram

The RTD is by definition monotone and nondecreasing. It is also a stepwise and right-continuous (so discontinuous) function, as shown in Figure 5.2. In fact, corresponding to each time required by one or more instances, the diagram increases abruptly by a step proportional to the number of instances solved. Considering the picture, no instance is solved in a time smaller than $t = 0.01$, so $F(t) = 0$, but at $t = 0.01$ at least one instance is solved, namely $1/20$th of all instances in the benchmark, so that $F(t) = 0.05$ for all $t \in [0.01, 0.1)$. The number of new instances solved can be inferred multiplying the height of each step by the overall cardinality of the benchmark: if $\overline{\mathcal{I}} = 20$, a step of 0.05 (as in $t = 0.01$) corresponds to a single new instance solved, while a step of 0.15 (as in $t = 1$) corresponds to three new instances. If $\overline{\mathcal{I}} = 40$, these values must be doubled.

In order to build the *RTD* diagram, this steps are to be followed:

1. run the algorithm on each instance $I \in \overline{\mathcal{I}}$
2. build the set $T_A(\overline{\mathcal{I}}) = \{T_A(I) : I \in \overline{\mathcal{I}}\}$
3. sort $T_A(\overline{\mathcal{I}})$ by nondecreasing values $t_1 \leq \cdots \leq t_{|\overline{\mathcal{I}}|}$

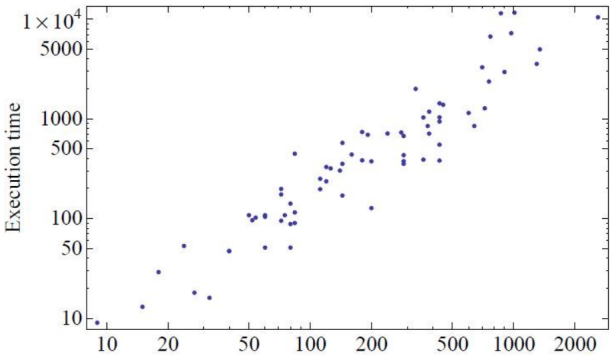


Figure 5.3: An example of a scaling diagram.

4. plot the points $(t_j, \frac{j}{|\overline{\mathcal{I}}|})$ for $j = 1, \cdots, |\overline{\mathcal{I}}|$, keeping only the maximum value of j if several t_j coincide.

5.2.2 Scaling diagram

The previous analysis makes sense only if all the parameters of the instances that strongly influence the computational time (first of all, the size) are fixed. Therefore, a description of the depencency of the run time on such parameters is needed. This can be done with the **scaling diagram**, that represents how the computational time scales with respect to the size (or other parameters) of the instance. In other words, it describes the dependence of $T_A(I)$ on the size n of I.

The diagram is built by choosing a sequence of values for the size n and generating a benchmark sample $\overline{\mathcal{I}}_n$ for each value. Then, the algorithm is applied to each $I \in \overline{\mathcal{I}}_n$ for all n and the points $(n(I), T(I))$ can be plotted. To simplify the diagram, especially if the times for equal sizes are very similar, a single point $(n, \sum_{I \in \overline{\mathcal{I}}_n} T(I))/|\overline{\mathcal{I}}_n|)$ can be plotted for each value of the size, computing the arithmetic mean of the run times. Once this is done, an **interpolating function** must be assumed. This is usually suggested by the theoretical analysis, that provides an upper bound on the run time, associated to the hardest instances, and with undefined multiplying coefficients. The empirical analysis provided by the scaling diagram estimates *a posteriori* the average-case complexity and determines analitically the multiplying factors. It is quite obvious that this empirical average-case complexity must be better than the theoretical worst-case complexity. If it is found otherwise, then either the theoretical analysis is wrong or something in the implementation does not match the theoretical algorithm.

Choosing the interpolating function

The interpolating function can be suggested by the theoretical analysis on the algorithm, but also by a graphical manipulation on the plot, based on the use of logarithmic scales. Our eyes, in fact, are unable to interpret nonlinear functions with precision, but clearly spot linear trends. An algorithm typically shows an increasing profile with an upward concavity, due to the time increasing faster than the size. However, it is hard to distinguish whether the profile is exponential or polynomial, and even harder to guess the specific function in each of the two classes.

However, if an algorithm is exponential, using a *semilogarithmic scale*, that is keeping the regular scale on the horizontal axis and a logarithmic scale[2] on the vertical axis allows to say

[2]Any base is good, but base 10 is the most familiar one.

that

$$\log_{10} T(n) = \alpha \cdot n + \beta \iff T(n) = 10^{\beta} 10^{\alpha \cdot n}$$

So, the plot becomes linear in this scale if and only if it is exponential in the original diagram. The two parameters α and β can be estimated (or calculated with methods such as the least squares), providing a detailed approximation of the "real" complexity function. Of course, this approximation neglects additional or multiplicative terms of minor complexity.

If, instead, an algorithm is polynomial, representing it on a logarithmic scale on both axes allows to say that

$$\log_{10} T(n) = \alpha \log_{10} n + \beta \iff T(n) = 10^{\beta} n^{\alpha}$$

In this case, the coefficient α directly provides the degree of the polynomial. The coefficient 10^{β} measures the coefficients left unknown by the theoretical analysis, but also the technological coefficients that transform the number of elementary operations into a physical time in *seconds*. Therefore, β depends on the machine used, whereas α should not. Also in this case, other terms of minor complexity are ignored, so the estimate is approximate (besides the approximation introduced by the fact that it refers to a specific benchmark sample).

5.3 A posteriori effectiveness evaluation

5.3.1 Solution quality distribution diagram

The statistical model of effectiveness is similar to that of efficiency. The **Solution Quality Distribution** (SQD) diagram plots the cumulative distribution function of $\delta_A(I)$ on a benchmark $\overline{\mathcal{I}}$:

$$\forall \alpha \in \mathbb{R}\ F_{\delta_A}(\alpha) = Pr[\delta_A(I) \leq \alpha]$$

In words, the SQD diagram shows the probability that the relative difference obtained by the algorithm on the instances of the benchmark is smaller than or equal to a value α for any possible real α. We remind that, if the optimal values are unknown, the relative difference is overestimated or underestimated, and therefore the SQD obtained is only an approximation of the true one. If both estimates are known, the true SQD is somewhere in the region between the two estimated diagrams.

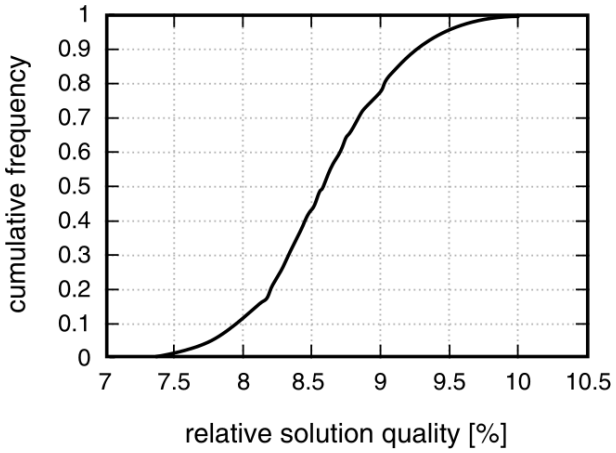


Figure 5.4: An example of SQD diagram.

The SQD diagram in Figure 5.4 states that the algorithm investigated never obtains a relative difference smaller than 7.5% on the benchmark. A relative difference $\leq 8\%$ can be obtained in $\approx 12\%$ of the instances. Finally, all instances can be solved with a gap at most equal to 10%.

Properties and construction of the SQD diagram

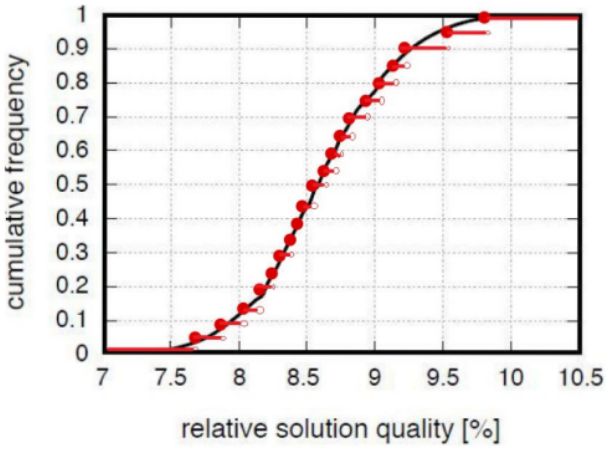


Figure 5.5: Construction of an SQD plot.

The SQD is a monotone nondecreasing function, as accepting worse gaps allows to solve more problems. It is equal to 0 for $\alpha < 0$ (but not necessarily for $\alpha = 0$!), and equal to 1 for $\alpha \geq \max_{I \in \overline{\mathcal{I}}} \delta(I)$, because it is not possible to have a negative gap and the maximum gap is associated to the hardest instance in the benchmark. Though it is often approximated as a continuous profile, the SQD diagram is stepwise and right-continuous (see Figure 5.5), as the graph steps up at each new value of $\delta_A(I)$. As in the RTD diagram, the number of instances in the benchmark determines the smallest step.

For an exact algorithm, which solves all instances to optimality, the SQD is a step function that reaches cumulative frequency 1 at $\alpha = 0\%$. For an $\overline{\alpha}$-approximated algorithm, the plot certainly reaches a cumulative frequency equal to 1 when the relative difference $\overline{\alpha}$ corresponds to the approximation ratio guarantee.

The SQD diagram is built in the same way as the RTD:

1. run the algorithm on each instance $I \in \overline{\mathcal{I}}$
2. build the set $\Delta_a(\overline{\mathcal{I}}) = \{\delta_A(I) : I \in \overline{\mathcal{I}}\}$
3. sort $\Delta_A(I)$ by nondecreasing values $\delta_1 \leq \cdots \leq \delta_{|\overline{\mathcal{I}}|}$
4. plot the points $(\delta_j, \frac{j}{|\overline{\mathcal{I}}|})$ for $j = 1, \cdots, |\overline{\mathcal{I}}|$, keeping only the maximum value of j if several δ_j coincide.

5.3.2 Parametric SQD diagrams

Meaningful RTD diagrams usually require to fix the size of the instances. This is not obviously required by the SQD diagram, since the quality of solutions may not depend so strongly on the size of the instance. Therefore, the SQD can be meaningful also when drawn with respect to the whole benchmark. Still, there may be some interest in analysing fixed-size subbenchmarks, obtaining a different profile for each one.

Figure 5.6 shows an example in which different SQD diagrams are drawn on the same scale, parameterised by the size of the instances in each subset of the benchmark. In this example, the gap tends to grow as the size of the instances grows. This is not always the case: there

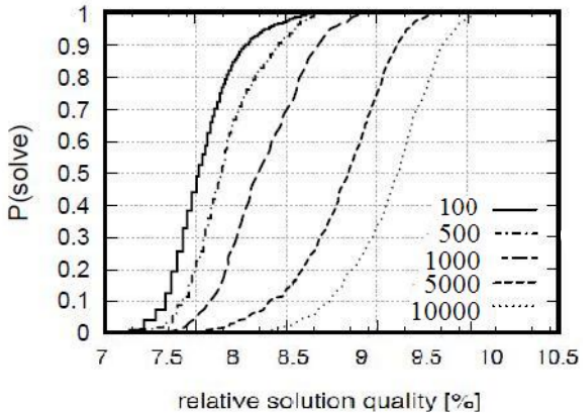


Figure 5.6: An example of parametric SQD diagram.

are lots of problem in which getting larger instances gives smaller relative gaps. In fact, the relative difference is the ratio of the absolute gap to the optimal value: large instances tend to have larger absolute gaps, but also larger optimal values; if the latter prevails, the relative gap δ decreases overall.

5.3.3 Algorithm comparison with SQD diagrams

We have already stated that an algorithmic comparison can be performed only if the run time is, at least nearly, the same for all instance of the considered benchmark. Let us assume that we are in this case. An algorithm A_2 **strictly dominates** an algorithm A_1 when it obtains better results on all instances:

$$\forall I \in \mathcal{I}\ \delta_{A_2}(I) \leq \delta_{A_1}(I)$$

This is a natural definition of dominance, but very rare in practice. It usually happens only on trivial cases, when A_2 "includes" A_1, that is first computes the same solution and then improves it.

A weaker, but more useful, definition is the so called **probabilistic dominance** of A_2 on A_1, that is based on the distribution functions, or the SQD diagrams of the two algorithms:

$$\forall \alpha \in \mathbb{R}\ (\alpha > \tau) \implies (F_{\delta_{A_2}}(\alpha) \geq F_{\delta_{A_1}}(\alpha))$$

An algorithm shows probabilistic dominance on another when the SQD diagram of the former is "above" that of the latter: it achieves the same gap more frequently. Equivalently, the former is on the left of the latter: an equal subsample of the best instances (the best half, for example) corresponds to a better gap.

Figure 5.7 shows a situation in which two algorithms do not dominate each other. Yet, they are quite different: the quality of the results for A_1 is more dispersed than that of A_2, as there are both better and worse gaps. In this case, one can say that A_1 is less "robust" than A_2, but choosing one or the other is legitimate, based on whether one adopts an optimistic or pessimistic view.

5.3.4 Position indices and boxplots

The previous descriptions take a lot of space, while sometimes it is useful to describe the performance of an algorithm in a compact way, possibly even with a single number. This can be

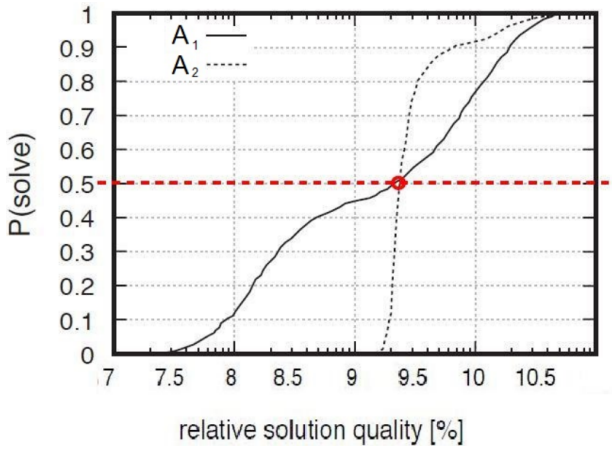


Figure 5.7: An example of SQD-based comparison.

done (for the relative difference: the run time is already described by the interpolating function) with statistical indices. In particular, classical statistical indices of *position*, such as the sample mean (the arithmetic mean of all measured values)

$$\overline{\delta}_A = \frac{\sum_{I \in \overline{\mathcal{I}}} \delta_A(I)}{|\overline{\mathcal{I}}|}$$

and of *dispersion*, such as the sample variance

$$\overline{\sigma}_A^2 = \frac{\sum_{I \in \overline{\mathcal{I}}} (\delta_A(I) - \overline{\delta}_A)^2}{|\overline{\mathcal{I}}|}$$

can be used in this context. However, these classical indices tend to be very influenced by **outliers**, that is instances on which the algorithm is very good or very bad. For example, suppose that an algorithm hits 90% of the times the optimum and 10% of the time a solution ten times larger than the optimum (that is, a percent gap of $(10f^* - f^*)/f^* = 900\%$). This is clearly an outlier: the algorithm is quite unstable, but not so bad! However, the corresponding sample mean of the gap is $0 \cdot 90\% + 10 \cdot 900\% = 90\%$, suggesting that on average the algorithm provides solutions that cost nearly twice the optimum.

So, sometimes, indices which are more stable with respect to the outliers are used, such as the **sample median** and suitable **sample quartiles**.

Boxplots

The median and all quartiles can be represented graphically with a **boxplot** (or *box and whiskers* diagram).

The example of Figure 5.8 represents a benchmark of 20 instances, in which the relative difference ranges from $\approx 4\%$ to $\approx 11.5\%$, that are the best and the worst value, respectively. They are also known as zero-th and fifth quartile. The **first quartile**, or lower quartile, is a value above one quarter and below three quarters of the elements. The **median**, or second quartile is a value above half and below half of the elements. The **third quartile**, or upper quartile, is a value above three quarters and below one quarter of the elements. The exact definitions of these numbers are not perfectly standard: different books and softwares adopt slightly different definitions.

If the sample consists of an odd number n of elements $(\delta_1, \ldots, \delta_n)$, the median is the value of the central one: $\delta_{(n+1)/2}$. In a sample made of an even number of elements, a possible definition

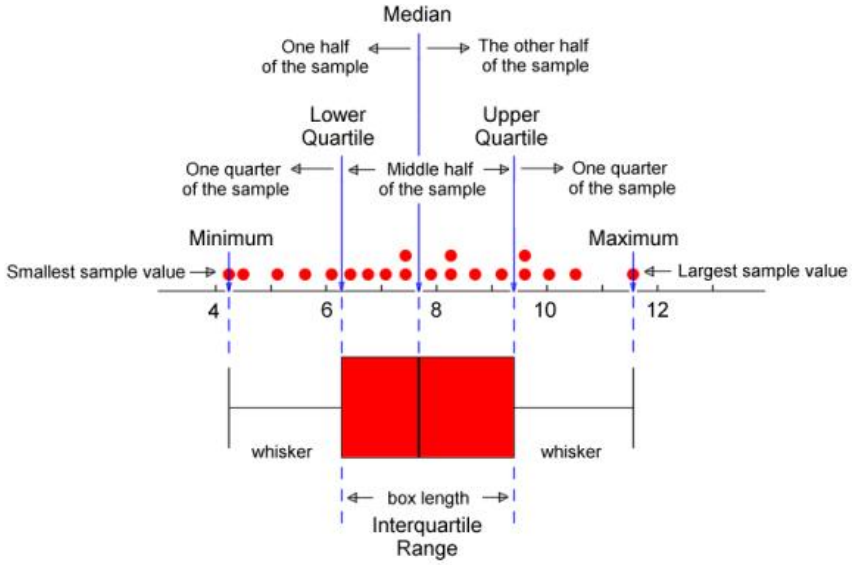


Figure 5.8: An example of boxplot.

is the midpoint between the two central elements: $\delta_{n/2} + \delta_{n/2+1}$, but any intermediate value between them is actually a defensible option. Following the same line, a possible definition of the first quartile is to compute $(n+1)/4$ and interpolate between $\delta_{\lceil (n+1)/4 \rceil}$ and the following element according to the fractionary part of $(n+1)/4$; in the same way, the third quartile can be computed interpolating between $\delta_{\lceil 3(n+1)/4 \rceil}$ and the following element based on the fractionary part of $3(n+1)/4$.[3] So, in a benchmark of 20 instances, the lower quartile is between the fifth and the sixth element, because $(n+1)/4 = 5.25$. In the precise definition, is is at one quarter of the distance from the fifth element and three quarters from the sixth, because the fractionary part is 0.25. The median is bewteen the tenth and eleventh element (for example, in the middle point), because $(n+1)/2 = 10.5$. The upper quartile is between the fifteenth and the sixteenth element (closer to the latter, to be precise), because $3(n+1)/4 = 15.75$.

The boxplot diagram is composed by a rectangle whose borders correspond to the first and third quartile, with a line corresponding to the median. Two other lines (the "whiskers") connect the rectangle to the values of the minimum and maximum. The boxplot gives both position and dispersion of the benchmark, since all elements fall between the minimum and the maximum, half of them fall inside the box, half outside, half before the median and half after, a quarter between each pair of consecutive quartiles.

The boxplot diagram is a simplified version of the SQD, as it reports five points of the overall profile. It is therefore possible to draw a boxplot starting from an SQD diagram and to approximately reconstruct the SQD diagram starting from the boxplot (see the exercises in Section 5.7). In fact, the five quartiles identify four rectangular "zones" in which the SQD diagram is confined, but they do not provide exact points, except for the minimum and the maximum.

As two algorithms can be compared using the SQD, they can be compared using boxplots, as well. Of course, since the description is less precise, less information can be extracted. Figure 5.9 shows an example with 8 boxplots that represent the relative difference of 8 algorithms on a benchmark sample. The circles represent outliers: they are extracted from the population before drawing the boxplots, but reinserted to complete the available information. Some in-

[3] I have still to check that this definition is the one adopted in the exercises. Anyway, any number in the correct range is acceptable.

teresting conclusion can be drawn from the picture. For example, the boxplots of A_7 and A_8 are completely separated: this means that algorithm A_7 has values better than A_8 on all the instances. This is the definition of *strict dominance*. Notice that having separate boxplots is a sufficient, but not necessary, condition for strict dominance: an algorithm could have slightly better results on all instances, with a boxplot only slightly lower.

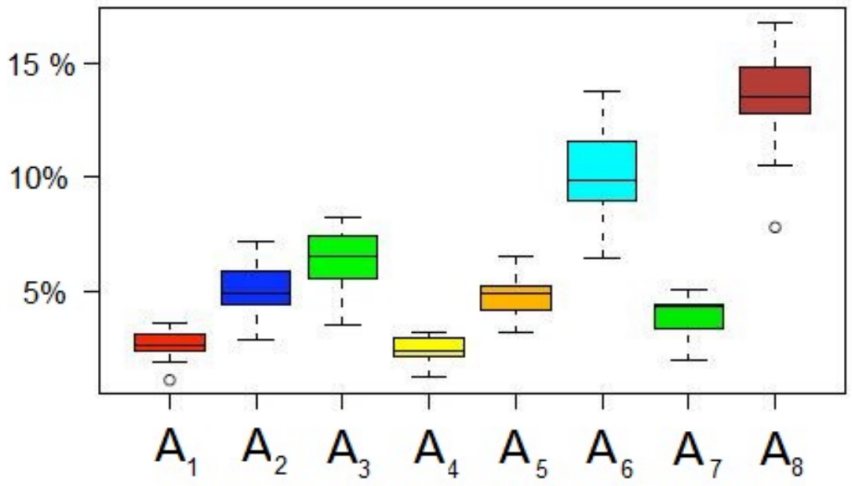


Figure 5.9: Algorithm comparison with boxplots.

As for *probabilistic dominance*, the boxplots once again provide necessary conditions or sufficient conditions. A necessary condition is that each of the five quartiles for an algorithm be not larger than the corresponding quartile for the other algorithm. For example, each quartile of A_2 is below the corresponding quartile of A_3. However, one instance of A_2 could sit on the lower whisker (the minimum) while all the other instances of the first quarter could be close to the first quartile (they sit on the upper bound); the opposite could happen for A_3, forbidding probabilistic dominance. On the other hand, a sufficient condition can be formulated if each quartile of an algorithm is below the previous quartile of the other algorithm. In this case, in fact, the four "zones" of the first boxplot are below the corresponding zones of the other, and this guarantees that the first SQD diagram is below the second.

5.4 Relation between quality and computational time

A heuristic algorithm is better than another one when it gives *better results in shorter time*. Otherwise, none of the two algorithms dominates the other. In general, different algorithms take different times, so it is actually difficult to compare them. This is in particular complex for the classical algorithms taught in basic courses, which generate a single solution immediately before their termination. However, many algorithm (all metaheuristic ones, in particular, but also heuristic algorithms such as the exchange heuristics) find several solutions during their execution, and just return the best one. This means that the algorithm can be stopped in advance, still obtaining a feasible solution (if the termination is not too premature). Therefore, two algorithms can be stopped at the same time and compared. In turn, this also means that the relative difference depends not only on the instance, but also on the actual computational time. The relative difference, therefore, should be defined as $\delta_A(t,I)$, that is the relative difference achieved by algorithm A at time t on instance I. By convention, if at time t algorithm A has not found yet a feasible solution, then $\delta_A(t,I) = +\infty$.

Considering the plot with respect to t for a fixed instance I, $\delta_A(t,I)$ is a stepwise monotone nonincreasing function, since more time yields a better solution. After the regular termination, for $t \geq T(I)$, the function is constant.

Randomised algorithms

Considering a metaheuristic algorithm with random steps, the relative difference is also a function of the random seed ω, so the gap should be defined as $\delta_A(t,I,\omega)$. When testing such an algorithm, therefore, ther are two random elements: the instance I and the random seed. While testing the algorithm, these two factors can be combined or kept separated. For example, an experiment can solve a fixed instance I with a batch of seeds $\overline{\Omega}$; another experiment can fix ω (running the algorithm in a deterministic way), and solve a sample of instances $\overline{\mathcal{I}}$. The results with respect to ω are usually summarised providing both the minimum relative difference $\delta_A^*(I,t)$ associated with the total time $|\overline{\Omega}|t$ and the average relative difference $\overline{\delta}_A(I,t)$ with the single-run time t.

5.4.1 A classification of algorithms

It is now possible to build a **classification** of algorithms for combinatorial optimisation problems, based on the relation between their runtime and solution quality. In my personal opinion, the names of the classes are quite hard to remember and not very clear, but they are used at least in some community, and the basic ideas are, on the contrary, clear and useful.

Classification with respect to optimal solutions

All algorithms for combinatorial optimisation problems can be divided into the following three classes.

Complete algorithms A **complete** algorithm, more commonly denoted as **exact algorithm** returns a solution with relative difference $\delta_A(I,t) = 0$, that is an optimal solution, in finite time for all the instances of the problem. Formally:

$$\forall I \in \mathcal{I}\ \exists \bar{t}_I \in \mathbb{R}^+\ \forall t \geq \bar{t}_I : \delta_A(I,t) = 0$$

Probabilistically approximately complete algorithms Randomised algorithms do not provide an optimal solution in a specific finite time, but the probability to find one does not decrease as time passes, and possibly increases converging to certainty. A **probabilistically approximately complete** algorithm has a probability of getting a relative difference equal to zero that grows up to one as time goes up to infinity, for all instances. Formally:

$$\forall I \in \mathcal{I}\ \lim_{t \to +\infty} Pr[\delta_A(I,t) = 0] = 1$$

where the probability refers to the random seed $\omega \in \Omega$, since the property holds for every fixed instance $I \in \mathcal{I}$. The idea is that, sooner or later, such an algorithm will find the optimum of any instance. It is a rather strange property and its utility can be questionable, given that a combinatorial optimisation problem always admits an exhaustive algorithm that finds the optimum certainly in finite time. Finding the optimum very likely in an infinite time does not sound as big news. However, it is an interesting theoretical property to supports the choice of specific

randomisation mechanisms, and from the practical point of view, the idea is that the convergence could be fast enough to make the probability close to one in a time much smaller than the exponential time required by the exhaustive algorithm. Lots of randomised metaheuristics can be proved to enjoy this property.

Essentially incomplete algorithms An **essentially incomplete** algorithm is *truly* heuristic, with none of the previous guarantees: for some instances $I \in \mathcal{I}$ its probability to find the optimum is strictly lower than 1, even as the runtime approaches infinity. Formally:

$$\exists I \in \mathcal{I} : \lim_{t \to +\infty} Pr[\delta_A(I,t) = 0] < 1$$

Most constructive and exchange heuristics fall within this class. In particolar, for deterministic algorithms, there is no random seed, and the "probability" becomes 1 for the instances solved exactly and 0 for the other instances.

Classification with respect to approximated solutions

A more general classification, based on the same concepts, can be introduced by replacing the search for optimal solutions with the search for approximation guarantees. One just replaces 0 with a threshold α in the three previous definitions.

An **α-complete** algorithm admits, for each instance $I \in \mathcal{I}$, a time $\bar{t}_I$ fter which the algorithm returns a solution with relative difference $\leq \alpha$. This is just an equivalent way to define an α-approximated algorithm.

A **probabilistically approximately α-complete** algorithm finds, for each instance $I \in \mathcal{I}$, an α-approximated solution with probability converging to 1 as the runtime approaches infinity.

Finally, an **essentially α-incomplete** algorithm admits some instance $I \in \mathcal{I}$ such that an α-approximated solution can be found only with probability strictly smaller than 1, even as time approaches infinity.

5.5 Complete performance diagrams

Instead of considering computational time and solution quality as separate aspects, one can build diagrams that describe them together, showing their trade-offs. These diagrams depend on two parameters: t is the upper threshold on the runtime, related to the efficiency of the algorithm; α is the upper threshold on the relative difference, related to the effectiveness of the algorithm.

We therefore define the **success probability** $\pi_{A,n}(\alpha,t)$ as the probability for algorithm A to find in a time not larger than t a solution with a gap not larger than α on an instance of size n:

$$\pi_{A,n}(\alpha,t) = Pr[\delta_A(I,t) \leq \alpha | I \in \mathcal{I}_n, \omega \in \Omega]$$

where the probability is expressed with respect to both possible sources of randomisation, that are the choice of the instance (with a fixed size, because that influences strongly the runtime) and the choice of the random seed (if the algorithm is randomised).

As represented in Figure 5.10, this diagram is three-dimensional, as it depends on two variables, the relative difference and the time.

Three auxiliary families of diagrams can be extracted from it by "cutting" the surface along planes orthogonal to one of the three axes: quality, time and probability.

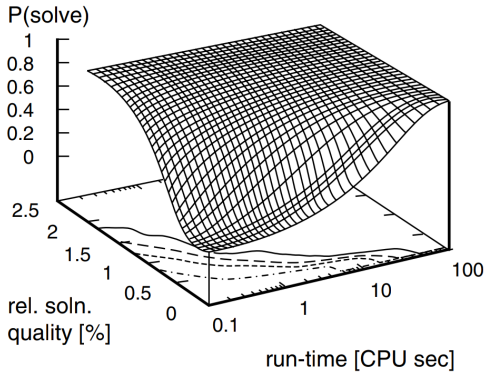


Figure 5.10: An example of success probability diagram

5.5.1 Qualified run time distribution diagrams

The first parametric family includes the **qualified run time distribution** diagrams, in which a quality level α is fixed and the profile considered is the intersection of the corresponding plane with the overall surface (see Figure 5.11).

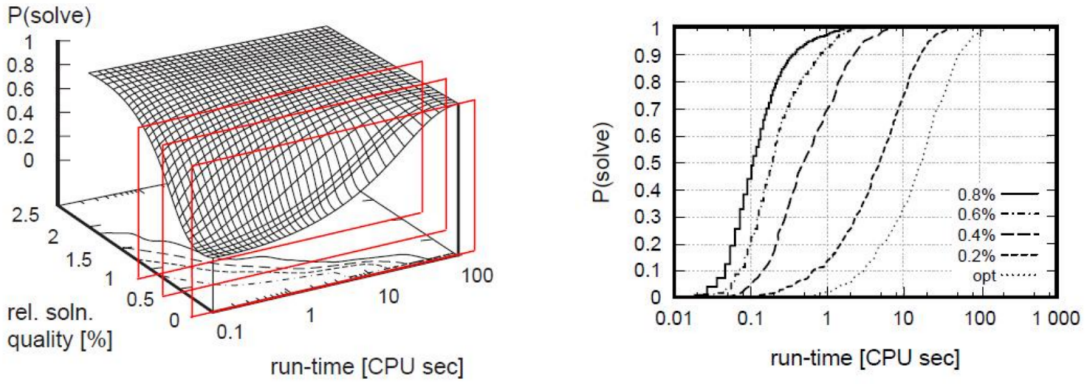


Figure 5.11: An example of QRTD diagram.

The practical meaning of this diagram is that, if one aims to obtain a certain quality (expressed by $\overline{\alpha}$), the plot shows the probability of reaching such a level of quality for each given runtime. For example, in an exact algorithm the diagram corresponding to $\alpha = 0$ reaches probability 1 in a suitable finite time. In an $\overline{\alpha}$-approximated algorithm, optimality may never be reached, so that diagram could stop below 1, but all diagrams associated to values $\alpha \geq \overline{\alpha}$ reach 1 in finite time. Finally, the $\overline{\alpha}$-incomplete algorithms have diagrams that do not reach 1 for any $\alpha \leq \overline{\alpha}$.

5.5.2 Timed solution quality distribution diagrams

The second parametric family includes the **timed solution quality distribution** diagrams, in which a maximum time t is fixed and the profile considered is the intersection of the corresponding plane with the overall surface (see Figure 5.12).

The practical meaning is that, given a certain available time, one will have a certain probability of reaching each possible solution quality. For example, the diagram of an exact algorithm will be a step functions in $\alpha = 0$, provided that t is not smaller than the runtime of the algorithm. If the algorithm is α-approximated, all diagrams with t not smaller than the runtime reach 1 in

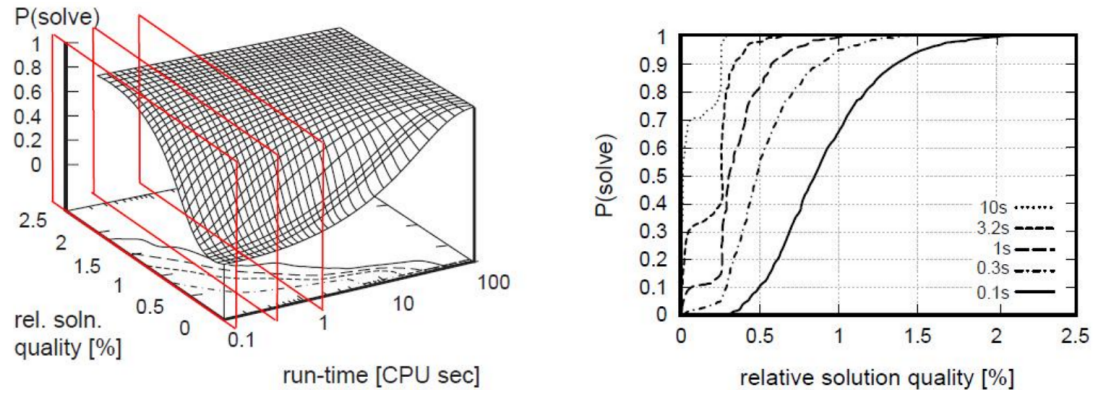


Figure 5.12: An example of TSQD diagram.

$\alpha = \overline{\alpha}$. If the algorithm is probabilistically approximately α-complete, the diagrams converge to 1 in $\alpha = \overline{\alpha}$ as t increases. Finally, if the algorithm is $\overline{\alpha}$-incomplete, all diagrams remain below 1 in $\alpha = \overline{\alpha}$.

5.5.3 Solution quality statistics over time diagrams

Finally, the **solution quality statistics over time diagrams** draw the level lines associated to different quantiles, as represented in figure 5.13.

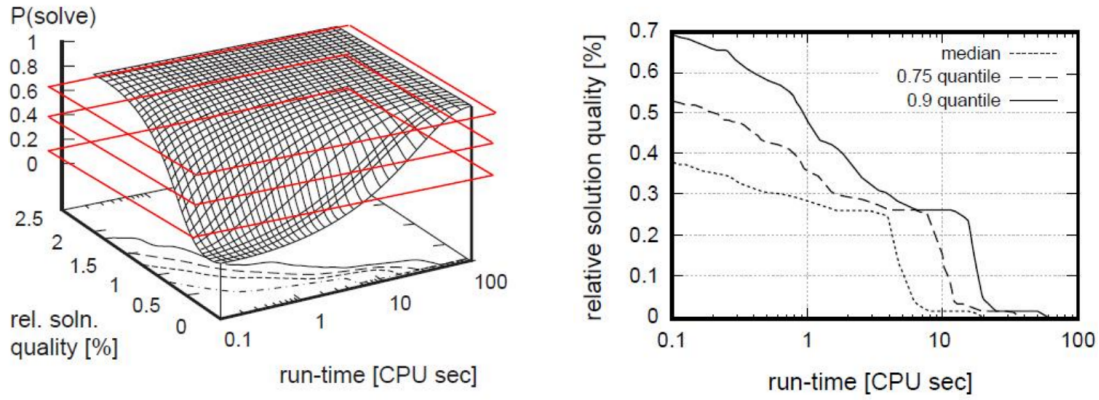


Figure 5.13: An example of SQT diagram.

These diagrams are less straightforward in their meaning. Suppose that we are interested in the median of the result obtained for different instances and, if used, random seeds. In other words, we want results that hold at least half of the time. The profile considered is the intersection of the corresponding plane with the overall surface (see the dotted line in Figure 5.13), and shows the possible compromises between time and quality for the best half of the cases. For example, an optimal solution can be found (half of the times) in at most 20 seconds, whereas a worse quality (up to a 0.4% gap) has to be accepted for shorter runs. If one wants a stronger guarantee, one can consider a larger quantile: the continuous line describes the trade-off holding for 90% of the instances (and random seeds).

5.6 Wilcoxon test

SQD diagrams and boxplots allow to compare algorithms, but such comparisons are intrinsically qualitative, unless probabilistic or strict dominance can be proved. Statistical tests can be used to quantitatively evaluate the significancy of the empirical difference between the performance of two algorithms on a benchmark sample. In the following, we consider **Wilcoxon's signed-rank test**, which focuses on the relative effectiveness of two algorithms. Many other tests exist (and could possibly give different results: all results are probabilistic in nature, of course). Moreover, some tests focus on different aspects. For example, the Kolmogorov-Smirnoff test determines whether the SQD of two algorithms are significantly different, without determining if one of them is better than the other: the two profiles in Figure 5.7, are probably different according to Kolmogorov-Smirnoff, but not to the Wilcoxon test.

Statistical tests are based on formulating a hypotesis H_0, called **null hypotesis**, and computing the probablity to observe a certain empirical behaviour under the condition that the null hypotesis is true. If that probability is very low, we reject the null hypothesis, even if it could still be true. More than rejecting or accepting (in a binary way), one should think of trusting more or less the conclusion suggested by the experiments.

The application of WIlcoxon's test to the comparison of algorithms assumes as a null hypotesis that two algorithms A_1 and A_2 are equivalent, and therefore f_{A_1} will be half the time better than f_{A_2} and half the time worse. More in detail:

1. $f_{A_1}(I) - f_{A_2}(I)$ is a random variable defined on the sample space $\mathcal{I}$.
2. the **null hypotesis** H_0 states that the theoretical median of $f_{A_1}(I) - f_{A_2}(I)$ is 0.
3. extract a sample of instances $\overline{\mathcal{I}}$ and run the two algorithms on it, obtaining a sample of paired values (f_{A_1}, f_{A_2}).
4. the test computes the probability p of obtaining the observed result or a more "extreme" one, assuming that H_0 is true.
5. this is interpreted as the probability
   - to reject the null hypothesis H_0, assuming that it is true;
   - that is, to consider as different two medians that are actually the same;
   - that is, to consider as differently effective (with respect to the median of the gap) two algorithms that are actually equivalent.
6. setting a **significance level** $\overline{p}$, we reject H_0 when $p < \overline{p}$, concluding that one of the two algorithms is better than the other, because it is too unlikely to obtain the observed results in the case in which they are equivalent.

Therefore, in order to apply the test, we build a benchmark sample and run the two algorithms on all instances of the benchmark, obtaining a set of pairs of values. Assuming that H_0 is true, the difference of the two values in each pair should be positive or negative the same number of times, and the positive and negative differences should be approximately of the same size: having half large positive values and half small negative values, in fact, suggests that the first algorithm typically tends to return larger values. To take this aspect into account, the positive and negative results will not only be counted, but also ranked. Now, the aim of the test is to try and understand if the observed result is just due to "bad luck" or to the fact that the H_0

is actually false and one of the two algorithms is actually better than the other. This requires to compute the probability of obtaining the particular set of pairs observed, but also any other set of pairs that is more unbalanced than the one observed. Once the probability is computed, it is compared to the so called significance level $\overline{p}$, that is the maximum probability to consider safe not to reject H_0. Typical values for $\overline{p}$ are 5% and 1%. If the calculated probability is smaller, then H_0 is considered false and rejected, therefore the theoretical median between algorithms is considered not null. Whether it is positive or negative, and therefore which of the two algorithms is better, can be easily found based on the observations.f

5.6.1 Assumptions of Wilcoxon's test

Wilcoxon's test involves a number of complex methodological questions, that we shall briefly survey in the following. First of all, it is a **nonparametric test**: it does not assume *anything* on the distribution of the tested values. This is appropriate for the evaluation of the performance of heuristic algorithms, because the distribution of the results $f_A(I)$ of an algorithm is typically unknown.

Wilcoxon's test is based on the assumptions that:

1. all data are measured on an ordinal scale: it must be possible to *sort* the differences between the two values in each pair; in our case, the differences are numbers, therefore cardinal quantities, that can easily be sorted.

2. the two datasets are matched and derive from the same population; in our case, this is true, because each pair derives from the application of A_1 and A_2 to the same instance extracted from $\mathcal{I}$.

3. each pair of values is extracted independently from the others; this would require to generate each instance in the benchmark independently from the other ones; this assumption is not always satified in the generation of benchmarks (different instances can be correlated).

For example, vehicle routing problem instances are often generated considering the same cost function for arcs and different weight functions for nodes, or vice versa. In these case, some instances are not *independent* from each other.

Wilcoxon's test performs the following steps:

1. for every instance in the sample, compute the absolute difference $|f_{A_1}(I_i) - f_{A_2}(I_i)|$;

2. sort them by increasing absolute values and assign a rank R_i to each one;

3. separately sum the ranks of the pairs with a positive difference $W^+ = \sum_{i:f_{A_1}(I_i)>f_{A_2}(I_i)} R_i$ and those of the pairs with a negative difference $W^- = \sum_{i:f_{A_1}(I_i)<f_{A_2}(I_i)} R_i$. If the null hypotesis H_0 is true then the two sums should be equal.

4. the difference $W^+ - W^-$ allows to compute the value of p: each of the $|\overline{\mathcal{I}}|$ differences can be positive or negative, so there are $2^{|\overline{\mathcal{I}}|}$ outcomes. Then, p is the fraction of outcomes in which $|W^+ - W^-|$ is at least as large as the observed value.

5. if $p < \overline{p}$ the difference is significant; in this case, if $W^+ < W^-$, A_1 is better than A_2; if $W^+ > W^-$, A_1 is worse than A_2 (of course, if the problem is a minimisation problem; otherwise, the opposite holds).

An important detail is that in step (2) some absolute differences can be equal. In this case, they should be assigned the same rank, that is the average of all ranks with the same value. For example, suppose that 11 differences, ranking from 5 to 15 actually correspond to the same absolute difference. In this case, they should not be ranked differently, as 5, 6, etc..., but with the same rank, equal to the average value $\frac{\sum_{i=5}^{15} i}{11} = (5+15)/2 = 10$.

5.6.2 Computation of the p-value

Once the total positive and negative ranks W^+ and W^- have been calculated, it is necessary to compute the probability p that their difference is significant.

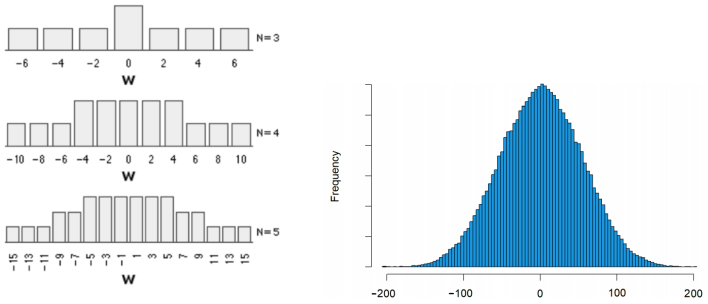


Figure 5.14: Wilcoxon's test distributions for various sample sizes.

In order to understand the computation, suppose to have only 3 instances (I_1, I_2 and I_3) as in the upper left corner of Figure 5.14. The differences will receive ranks 1, 2 and 3, with positive or negative signs. The most unbalanced situations occur when A_1 is always better than A_2, or the opposite, so that all three ranks have a negative sign, or all three ranks have a positive sign. The number of possible cases is $2^3 = 8$. The eight cases can be enumerated, as in the upper-left corner of Figure 5.14. Only in one case out of eight, all ranks are positive and $W^+ - W^- = (1+2+3) - 0 = 6$; only in one case out of eight all ranks are negative and $W^+ - W^- = 0 - (1+2+3) = -6$; in two cases the difference is zero: $W^+ - W^- = (1+2) - 3 = 0$ and $W^+ - W^- = 3 - (1+2) = 0$. The null hypothesis suggests that the two values of each pair are extracted from the same population and positive and negative differences have the same probability to occur with any ranking (that is, the same probability to be large or small). Therefore, the eight cases should have the same probability. Supposing that we observe an extreme case (all negative signs, for example). The probability of being in this biased situation **or in a worse one** is the sum of the probabilities for each case that is equally or more strongly unbalanced. In our example, it is $\frac{1}{8} + \frac{1}{8} = 0.25$. This is sufficiently large to think that the observed difference could be random. Given a larger number of instances, however, large differences correspond to small probabilities, that can lead to reject the null hypothesis. The p-value can be computed by enumeration, as in the previous example. If the instances are many, however, the *law of big number* proves that the distribution can be approximated it a normal one, and the probability can be derived from tables.

Wilcoxon's test can suggest either that one of the two algorithms is significantly better than the other, or that the two algorithms are statistically equivalent. In both cases, it is important to remember the answer is stochastic and to "keep an eye" on the value of p, without considering the answer as binary.

Another application of this test can be that two algorithms could be overall equivalent on a whole benchmark, but nonequivalent on the subclasses of instances within the benchmark.

The problem with this kind of experiments is that each new analysis on the same increases the probability that we are indeed in a case of "bad luck" and the suggestion of the test is wrong. This is why the value of p should be taken in serious account: a very small value allows many analysis before the *familywise error rate* (*FWER*) introduces incorrect deductions due to random results, a value close to the significance level makes these additional deductions quickly unjustified.

Another question is whether Wilcoxon's test should be applied to $\delta_A(I)$, instead of $f_A(I)$. Indeed, the results could be different: using $\delta_A(I)$ means giving a smaller weight to the instances with a larger optimum. There is no strict rule on this point: the definition of "better algorithm" is indeed at least partly arbitrary. Once again, the size of p is probably the best information: if the outcome of the test is different in the two cases, the value of p is probably not very small.

5.7 Exercises

5.7.1 Exercise 1

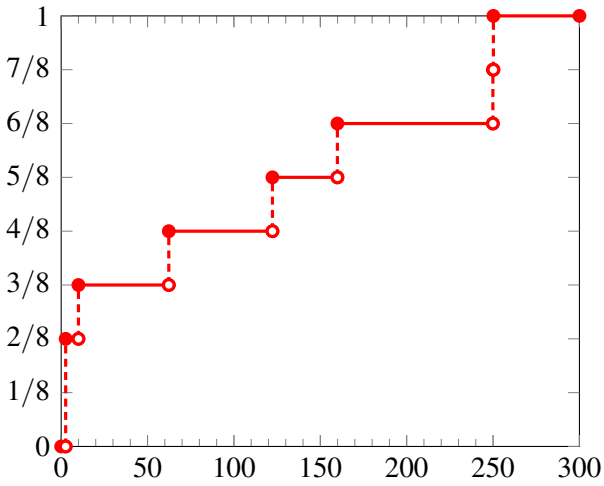
The following table reports the size n of 8 different instances and the computational time required by algorithm A to solve them.

	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
$n(I)$	10	50	10	100	20	80	70	100
$T_A(I)$	2.6	62.3	2.6	249.9	10.0	159.9	122.3	250.2

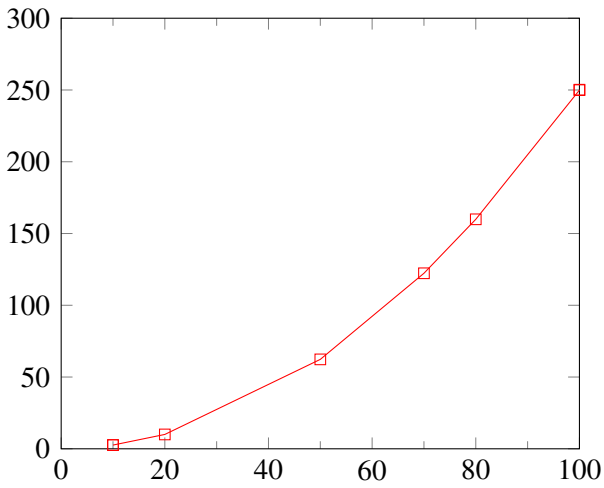
If possible, draw a *Run Time Distribution* diagram of A on the benchmark $\bar{\mathcal{I}}$; otherwise, explain why it is not possible. What can be deduced from this diagram, once it is drawn?

If possible, draw a *scaling* diagram of A on $\bar{\mathcal{I}}$; otherwise, explain why it is not possible. What can be deduced from this diagram, once it is drawn?

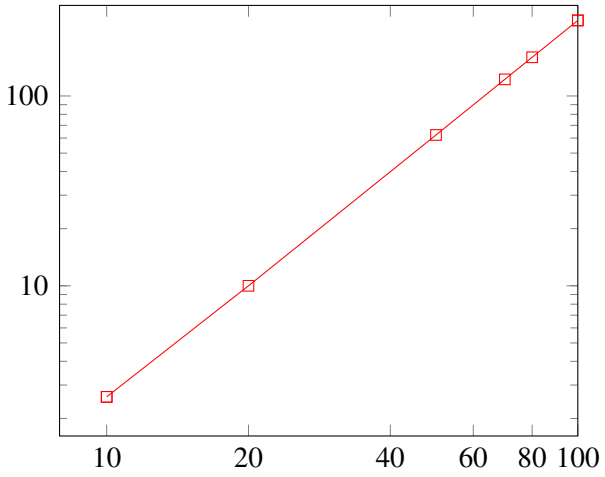
Solution Drawing a *RTD* on a benchmark of instances of different size is not very significant, because it reflects the distribution of instance sizes in the benchmark more than the time complexity of the algorithm, but of course it is always possible.



Drawing a *scaling* diagram simply takes to plot the points provided.



This is an expected profile, but still not very informative (exponential or polynomial? in the latter case which polynomial?), but one can try and use a logarithmic scale on both axes, keeping a uniform distance between values that are multiplied by the same amount (such as 2 or 10).



This shows that A is a polynomial algorithm. We can even venture to observe that, since $\beta 100^{\alpha} \approx 250$ and $\beta 20^{\alpha} \approx 10$ (or choose any other two points), then $(100/20)^{\alpha} \approx (250/10) \Rightarrow 5^{\alpha} = 25 \Rightarrow \alpha \approx 2$.

5.7.2 Exercise 2

The following table reports the result obtained by algorithm A on a benchmark $\bar{\mathcal{I}}$ of 8 instances of a minimization problem. For the sake of simplicity, assume that all instances have optimal value equal to 100.

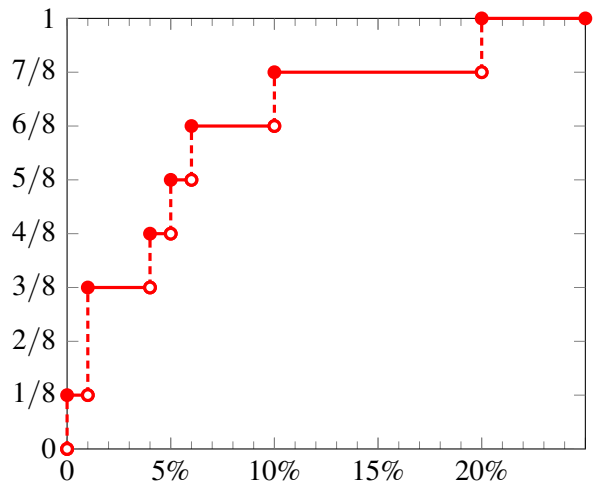
$\bar{\mathcal{I}}$	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
f_A	104	100	105	120	101	106	101	110

Draw the *Solution Quality Diagram* and the *boxplot* diagram of A on the benchmark.

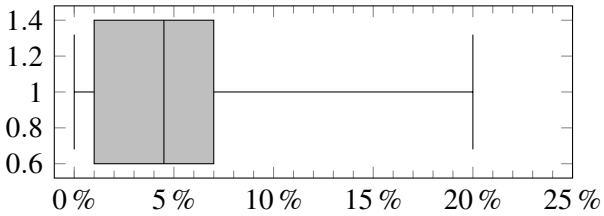
Solution The *SQD* that describes the performance of A on the benchmark is built computing all the relative differences and sorting them by nondecreasing values: $\{0\%, 1\%, 1\%, 4\%, 5\%, 6\%, 10\%, 20\%\}$. Since the cardinality of the benchmark is $\left|\bar{\mathcal{I}}\right| = 8$, and since two values coincide, the points that determine the plot are

$$\left(0\%, \frac{1}{8}\right) \left(1\%, \frac{3}{8}\right) \left(4\%, \frac{4}{8}\right) \left(5\%, \frac{5}{8}\right) \left(6\%, \frac{6}{8}\right) \left(10\%, \frac{7}{8}\right) (20\%, 1)$$

which yields the following diagram.



The corresponding boxplot diagram is as follows.



In order to draw the boxplot diagram, we have adopted the definitions of quartiles commonly used in spreadsheets. Since the cardinality of the benchmark is even ($n = 8$), the median is halfway between the $n/2 = 4$th and 5th element, the upper quartile is at one quarter of the way between the $3/4n = 6$th and 7th element, and the lower quartile is at three quarters of the way between the $1/4n = 2$nd and 3rd element. Notice that in this case the two elements coincide. In an exam, it is acceptable to use the simplified definitions for median ($n/2$th element), lower quartile ($n/4$th element) and upper quartile ($3n/4$th element).

5.7.3 Exercise 3 -

The following table reports the result obtained by algorithm A on 8 different instances.

	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
f_A	54	60	55	51	51	56	54	55

Assume (for the sake of simplicity) that all instances have optimum equal to 50.

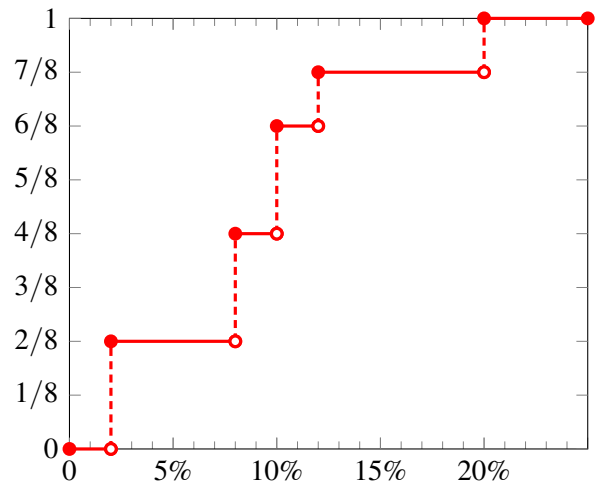
Draw the *Solution Quality Diagram* of A on the benchmark.

Draw the *boxplot* of A on the benchmark.

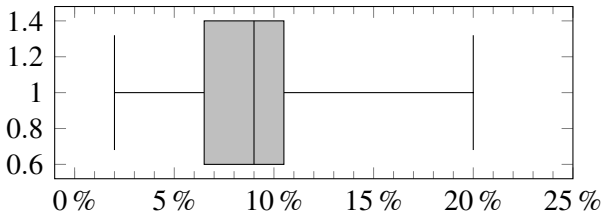
Solution The *SQD* that describes the performance of A on the benchmark is built computing all the relative differences and sorting them by nondecreasing values: $\{2\%, 2\%, 8\%, 8\%, 10\%, 10\%, 12\%, 20$
Since the cardinality of the benchmark is $\left|\bar{\mathcal{I}}\right| = 8$, and since three pairs of values coincide, the points that determine the plot are

$$\left(2\%, \frac{2}{8}\right) \left(8\%, \frac{4}{8}\right) \left(10\%, \frac{6}{8}\right) \left(12\%, \frac{7}{8}\right) (20\%, 1)$$

which yields the following diagram.

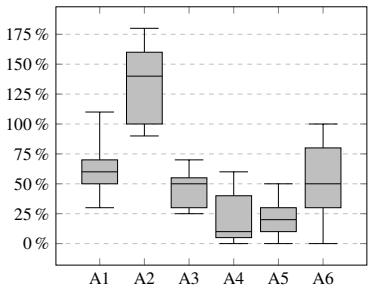


and the corresponding boxplot (with the same remarks of the previous exercise).



5.7.4 Exercise 4 -

The following picture represents the performance (percent relative difference δ) of six algorithms on a set of benchmark instances:



Is any of the algorithms tested $3/2$-approximated? Why?

What can be deduced on the relative quality of the six algorithms?

Solution All algorithms are certainly not $3/2$-approximated, with the exception of A_5, that could have such an approximation guarantee. A finite benchmark, however, is not enough to prove it.

Concerning strict dominances

$$A_3 \prec A_2 \qquad A_4 \prec A_2 \qquad A_5 \prec A_2$$

because, for each pair, the worst result of the first algorithm is not worse than the best result of the second one.

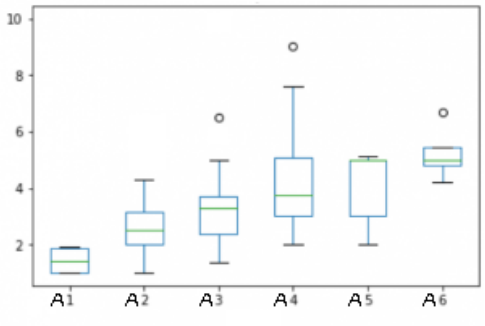
Concerning probabilistic dominances

$$A_1 \prec A_2 \qquad A_3 \prec A_1 \qquad A_4 \prec A_1 \qquad A_6 \prec A_2 \qquad A_5 \prec A_3$$

because, for each pair, each quartile of the first algorithm is not worse than the previous quartile of the second one. Other probabilistic dominances could exist, as they satisfy the necessary condition that each quartile of the first algorithm is not worse than the corresponding quartile of the second one (for example, $A_4 \prec A_3$, $A_4 \prec A_6$ and $A_5 \prec A_6$).

5.7.5 Exercise 5 -

The following figure represents the performance (percent relative difference δ) of six algorithms on a set of benchmark instances:



What can be deduced on the relative quality of the six algorithms?

Is it possible to draw (at least approximately) the *SQD* diagrams of the six algorithms? If it is, draw one of the diagrams.

Solution First, there are some strict dominances:

$$A_1 \prec A_4 \qquad A_1 \prec A_5 \qquad A_1 \prec A_6$$

Concerning probabilistic dominances, the sufficient condition holds for:

$$A_1 \prec A_2 \qquad A_1 \prec A_3 \qquad A_2 \prec A_4 \qquad A_2 \prec A_5 \qquad A_2 \prec A_6 \qquad A_3 \prec A_6$$

It is possible to find five points of the diagram. These points identify rectangles in which the true *SQD* diagrams is contained, but with an unknow behaviour, besides being stepwise nonincreasing[4]

5.7.6 Exercise 6 -

The following table reports the result obtained by algorithms A_1 and A_2 on 8 different instances.

	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
f_{A_1}	104	100	105	120	111	107	101	111
f_{A_2}	100	100	108	122	101	102	101	110

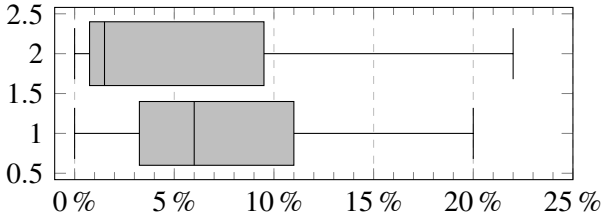
[4] I must get back on this answer to think about what happens when the median and the lower and upper quartile do not correspond to actual values, but are obtained with the spreadsheet definition. A correction could be necessary.

Assume (for the sake of simplicity) that all instances have optimum equal to 100.

Draw the *boxplots* of the two algorithms on the benchmark. What can be deduced from their comparison?

Apply the first steps of Wilcoxon's test to the benchmark, up to the computation of statistics W^+ and W^-, that is skipping only the computation of the final p-value.

Solution The two boxplots are the following.



The boxplots do not show any dominance, though, considering the worst result of A_2 as an outlier, the necessary condition for a probabilistic dominance would hold.

Wilcoxon's test computes the following differences: 4, 0, -3, -2, 10, 5, 0 and 1. Neglecting the two null differences, sorting the absolute values and saving the sign of the other differences provides the following signed ranks:

Rank	1	2	3	4	5	6
Difference	1	-2	-3	4	5	10
Signed rank	$+1$	-2	-3	4	5	6

The sums of the signed ranks are, therefore, $W^+ = 1+4+5+6 = 16$ and $W^- = 2+3 = 5$. The result of the test, that is not required during an exam, is

```
W+ = 16, W- = 5, N = 6, p <= 0.3125
```

suggesting that the difference between the two algorithms is likely to be due to chance.

5.7.7 Exercise 7 -

The following table reports the result obtained by algorithms A_1 and A_2 on 8 different instances.

	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
f_{A_1}	45	93	216	104	80	289	102	71
f_{A_2}	57	93	211	102	102	304	110	91

Apply the first steps of Wilcoxon's test to the benchmark, up to the computation of statistics W^+ and W^-, that is skipping only the computation of the final p-value.

Solution Wilcoxon's test computes the following differences: -12, 0, 5, 2, -22, -15, -8 and -20. Neglecting the null difference, sorting the absolute values and saving the sign of the other differences provides the following signed ranks:

Rank	1	2	3	4	5	6	7
Difference	2	5	-8	-12	-15	-20	-22
Signed rank	$+1$	$+2$	-3	-4	-5	-6	-7

The sums of the signed ranks are, therefore, $W^+ = 1+2 = 3$ and $W^- = 3+4+5+6+7 = 25$. The result of the test, that is not required during an exam, is

```
W+ = 3, W- = 25, N = 7, p <= 0.07813
```

suggesting that the difference between the two algorithms is not small, but still possibly due to chance. As it can be imagined, it is nearly impossible to obtain a statistically significant result with a benchmark of 8 instances.

Part III

Constructive algorithms

CHAPTER 6

Constructive heuristics

This part of the notes is dedicated to the first of the three classes of heuristics previously listed, the **constructive heuristics**. In particular, this chapter discusses heuristics, whereas the following one concerns metaheuristics. Together with constructive heuristics, we analyse **destructive heuristics**, whose structure is somewhat complementary.

6.1 Basic elements of constructive algorithms

In order to understand constructive heuristics, one should remember that **combinatorial optimization** problems are characterised by having solutions that are subsets of a suitable given finite ground set. The simplest way to obtain a solution, therefore, is to start from the empty subset $\emptyset$, and add one element at the time until it becomes impossible (or unlikely) that introducing new elements could generate optimal solutions. Then, the algorithm stops. Let us investigate this process in more detail.

A constructive heuristic updates a subset $x^{(t)}$ at each step t:

1. it starts from an empty subset ($x^0 = \emptyset$), that is obviously a subset of any optimal solution. If kernelisation or reduction procedures prove that some elements of the ground set are necessarily included in at least one optimal solution, it makes sense to include these "forced" elements in x^0. Heuristic reasons to think that some element should be included can also be taken into account.

2. it stops when a termination condition holds: the rationale of the termination condition is to guarantee that adding elements to the current subset $x^{(t)}$ will not generate any optimal solution.

3. the construction loop:

   (a) determines among all the elements $i^{(t)} \in B \setminus$ that do not belong to the current subset $x^{(t)}$ those that can extend it in an "acceptable" way.

   (b) choses an element $i^{(t)}$ and adds it to the current subset: $x^{(t+1)} := x^{(t)} \cup \{i^{(t)}\}$. The choice is based on the evaluation and optimisation of a suitable selection criterium.

4. go back to point 2.

This is a very simple scheme, that allows to generate several different algorithms for the same problem by simply specifying two characteristic elements: the **acceptable extensions** and the **selection criterium**.

Notice that at the beginning subset x enjoys the nice property of being part of any optimal solution, since $x^{(0)}$. This property is usually lost in some step by the addition of a "wrong" element. The algorithm must try to keep it by a smart definition of the acceptable extensions and the selection criterium. The same holds for feasibility: at the beginning, x is part of any feasible solution, but in the following it can lose such a property. This is an even worse outcome, but in general it is easier to avoid it. It depends on the problem.

6.1.1 The construction graph

The simple scheme of constructive heuristics allows to introduce nice modelling tools, on which useful theoretical results can be sometimes proved. The main modelling tool is the **construction graph**. This is a directed graph, identified by its nodes and arcs.

The node set $\mathcal{F}_A$ (where F stands for *find* and A is the specific algorithm described by the graph) is also known as **search space**. It is defined as the collection of subsets $x \subseteq B$ which can be potentially produced by algorithm A during its execution. Hence, $\mathcal{F}_A \subseteq 2^B$: every node is a subset of B, but not all subsets of B correspond to nodes of the construction graph (only the "acceptable" ones). The same problem admits different search spaces, corresponding to different construction algorithms.

The arcs connect pairs of nodes describing the acceptable extensions: the arc set is the collection of all pairs $(x, x \cup \{i\})$ such that $x \in \mathcal{F}_A$, $i \in B \setminus x$ and $(x \cup \{i\}) \in \mathcal{F}_A$. Every arc connects a subset to a slightly larger subset, representing an acceptable extension of a subset with a new element.

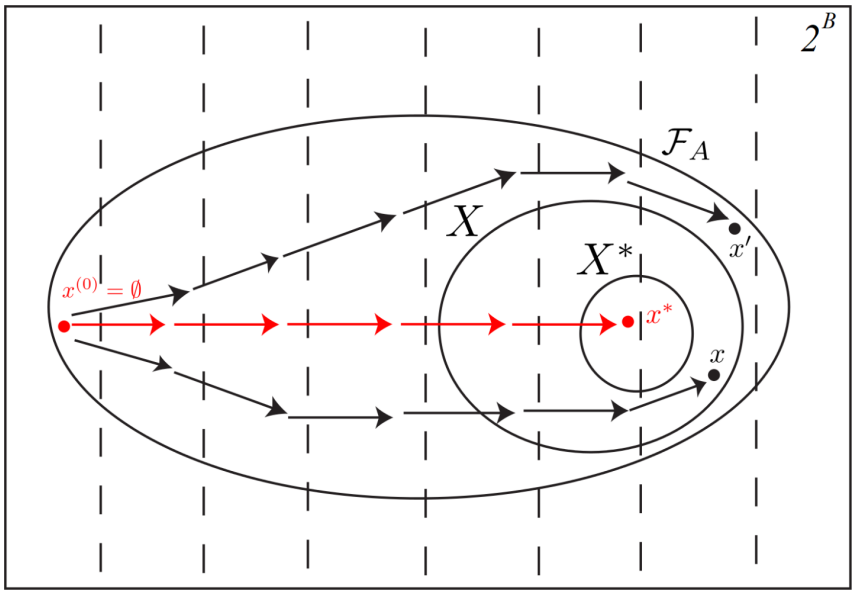


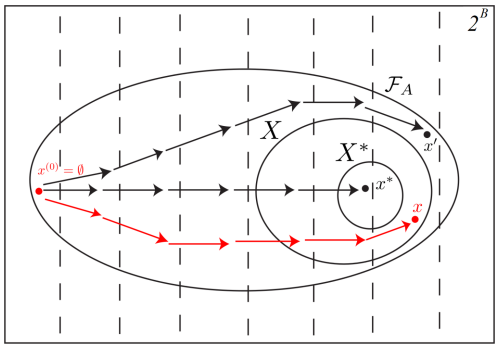
Figure 6.1: An example of construction graph, with an execution returning an optimal solution.

Figure 6.1 represents a construction graph. The rectangle represents 2^B, that is the power set of B. This collection of all possible subsets is partitioned by the dashed lines into classes containing subsets of different cardinality. The first and leftmost class contains all the subsets of 0 elements, and therefore includes only one element, the empty set $\emptyset$. The second class contains all *singletons*, the third the pairs of elements, and so on. Overall, there are $n+1$ classes, ranging from cardinality 0 to cardinality n.

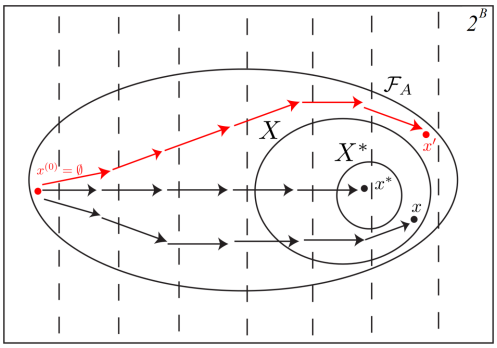
The search space $\mathcal{F}_A$ excludes all subsets that are unreasonable to visit because, for any reason, it is impossible (or higly unlikely) to reach an optimal solution from them. They belong

to $2^B \setminus \mathcal{F}_A$, they do not correspond to nodes and, of course, no arc can reach them. The search space $\mathcal{F}_A$ usually contains the set of all feasible solution X. Finally, X includes the set of all optimal solutions X^*. In principle, an algorithm A might exclude feasible solutions, and even optimal ones, from the search space. These cases, however, are based on suitable theoretical properties that guarantee the inclusion of other feasible and optimal solutions, to avoid crippling the search. The aim, then, is simply to concentrate the search on a smaller space, improving the efficiency and effectiveness of the algorithm.

The execution of a constructive algorithm corresponds to a maximal path in the construction graph: it starts from the empty set, visits some intermediate subsets and stops in a final subset, that cannot be extended in any acceptable way. Ideally, the final subset should be optimal, or at least feasible. Figuret6.1 represents the ideal case, in which the path ends in an optimal solution. It is the case of all exact constructive algorithms, such as Prim's, Kruskal's and Dijkstra's.



(a) A construction graph with an execution returning a feasibile nonoptimal solution.



(b) A construction graph with an execution returning an unfeasibile solution.

Figure 6.2: Example of construction graphs for nonexact algorithms.

In other cases, such as the ones represented in Figure **??**, the path may end in a nonoptimal, but feasible, solution $x \in X$ (Figure (a)), or even in subsets that are not feasible solutions $x' \notin X$ (Figure (b)). These are often called unfeasible solutions. The former is the case of heuristic algorithms such as those used for the KP and the MDP. The latter is the case of heuristic algorithms for the TSP on noncomplete graphs.

Properties of the search space in construction graphs

A fundamental property of the construction graph is that it is acyclic, since every arc links a smaller subset to a strictly larger one: no circuit is possible.

Other properties, though not strictly required, are typically imposed, because they correspond to practical aspects in the execution of a constructive algorithm. First of all, the search space $\mathcal{F}_A$ should include the empty subset ($\emptyset \in \mathcal{F}_A$), because any constructive heuristic algorithm starts from it. Most of the time, the search space $\mathcal{F}_A$ includes all feasible solutions ($X \subseteq \mathcal{F}_A$), as in Figure 6.1. However, some solutions that are obviously nonoptimal might be removed from the search space, if it can be proved that visiting them is useless to find an optimal solution. As well, usually an arc links every pair of subsets in the search space such that the second includes the same elements of the first plus one. However, some of these arcs could be removed if they are useless to find optimal solutions. For example, in the SCP, when a feasible solution is reached (that is, a subset of columns covering all rows), adding other columns makes no sense, because the resulting subsets will be more and more expensive, and none of them will be optimal. Therefore, the arcs connecting a feasible solution to a larger feasible solution can

be safely removed from the construction graph. Another common property is that $\mathcal{F}_A$ should contain only subsets that can be reached from the empty set with a directed path in the graph. If this is not possible, such nodes are useless.

A natural candidate for the definition of the search space is the collection of all **partial solutions**, defined as all subsets of all feasible solutions. This is a nice definition, that satisfies all properties discussed above. However, such a definition also poses a computational problem. In order to determine the acceptable extensions of a subset x, one must answer for each $i \in B \setminus x$ the following question: "Is $x \cup \{i\}$ a partial solution?" This question is called **inclusion test**, as it concerns the inclusion of a given subset in the search space. As it must be answered several times during the execution of the algorithm, the inclusion test must be polynomial, and possibly very fast. However, this question is also a decision problem (the answer is *true* or *false*) that generalises the feasibility test: "Does there exist a feasible solution?" In fact, answering the feasibility test is equivalent to answering the inclusion test for $x = \emptyset$. Depending on the specific problem considered, such a question can be trivial, easy or even $\mathcal{NP}$-complete.

If there is no easy known way to perform the feasibility test, a way out is to enlarge the search space, including subsets that are not partial solutions, but can be easily tested. Usually, the way to do this is to *relax* some constraint of the problem.

Examples of search spaces

Most constructive algorithms for the *KP* adopt the feasible solutions as the search space

$$\mathcal{F}_A \equiv X$$

as it is easy to test and satisfies all properties discussed above.

The constructive algorithms for the *MDP* typically consider the collection of partial solutions (that is, all subsets of cardinality $\leq k$)

$$\mathcal{F}_A \equiv \bigcup_{x \in X} 2^x$$

The same occurs in Kruskal's algorithm for the *MSTP*. In fact, this algorithm only forbids cyclic subsets of edges: the search space contains all forests.

Prim's algorithm for the *MSTP* is much more restrictive: it considers only subsets of edges that form trees spanning the starting vertex. Therefore, it does not include ever partial solution, but only special promising partial solutions

$$\mathcal{F}_A \subset \bigcup_{x \in X} 2^x$$

For the *TSP* on general graphs, it is $\mathcal{NP}$-complete to determine whether a given set of arcs belongs to a Hamiltonian circuit, because it is $\mathcal{NP}$-complete even to determine whether a Hamiltonian circuit exists at all. Therefore, the set of partial solutions is not a viable choice for the search space. A possible alternative is to relax the constraints and admit all subsets of arcs that are not clearly making it impossible to complete the solution. Such subsets are those in which some nodes have an ingoing or outgoing degree larger than one and those that include "subtours", that is circuits visiting only a strict subset of the nodes. In this case, the search space is strictly larger than the collection of partial solutions:

$$\mathcal{F}_A \supset \bigcup_{x \in X} 2^x$$

6.1.2 The termination condition

A constructive algorithm terminates when the current subset x can no longer be extended without leaving the search space. In other words, no arc in the construction graph goes out of the node. This condition is formally expressed as

$$\forall i \in B \setminus x^{(t)}\ x^{(t)} \cup \{i\} \notin \mathcal{F}_A$$

or, equivalently

$$\Delta_A^+(x^{(t)}) = \{i \in B \setminus x^{(t)} : x^{(t)} \cup \{i\} \in \mathcal{F}_A\} = \emptyset$$

where $\Delta_A^+(x^{(t)})$ is the set of acceptable extensions of subset $x^{(t)}$, each of which corresponds to an outgoing arc.

As long as $\Delta_A^+(x^{(t)}) \neq \emptyset$, the algorithm chooses one of the acceptable extension, and adds it to x. When this is no longer possible, it returns the best feasible solution visited during the execution. This is usually the last visited subset, because the previous ones are either unfeasible or worse. However, this is not always the case: $x^{(t)}$ could move in and out of X several times during the execution.

6.1.3 The general scheme

A constructive heuristic (for minimization problems) can be described by the pseudocode of Algorithm 1. It starts from an empty subset x, and initialises the best solution found x^* to x and its value f^* to the value $f(x)$ if it is feasible, to $+\infty$ otherwise. The algorithm proceeds until the termination condition is met, selecting at each step an element i and adding it to x. The choice of i is based on a selection criterium $\varphi(i,x)$, that depends on i, but in general also on the current subset x. Therefore, it is a function

$$\varphi_A : B \times \mathcal{F}_A \to \mathbb{R}$$

that returns a real value estimating how good the insertion of i in x is. For minimisation problem, we generally assume that good insertion correspond to small values of φ.

The general scheme of Algorithm 1 can be tailored to a specific problem by defining $\Delta_A^+(x)$ and $\varphi_A(i,x)$ which correspond, respectively, to the construction graph (Δ_A^+ identifies the topology of the graph: nodes and arcs), and to a weight function associated to the arcs of the graph ($\varphi_A(i,x)$ is the weight of the arc from node x to node $x \cup \{i\}$).

6.1.4 Effectiveness and efficiency of constructive algorithms

At the beginning, subset x is contained in any optimal solution. In the luckiest cases, it maintains this property throughout the execution of the algorithm, and therefore terminates returning an optimal solution. The optimality of a constructive algorithm critically depends on the preservation of this property - being included in an optimal solution - which requires some nice property both from the problem and from the algorithm.

A general discussion can be made also about the efficiency of constructive algorithms. Their cost derives from the execution of $|x| \leq n$ iterations (where $n = |B|$) and from the complexity of the operations performed at each iteration. These are the scan of the extension set $\Delta_A^+(x)$, the evaluation of the selection criterium $\varphi_A(i,x)$ for each $i \in \Delta_A^+(x)$, the choice of the minimum value, the update of the current solution x and, possibly, of auxiliary data structures and of the best known solution x^*. Usually, the most expensive terms are the time $T_{\Delta_A^+}(n)$ required to scan

Algorithm 1 Constructive Heuristic Pseudocode

```
1:  procedure GREEDY(I)
2:      x := ∅
3:      x* := ∅
4:      if x ∈ X then
5:          f* := f(x)
6:      else
7:          f* := +∞
8:      end if
9:      while Δ_A^+(x) ≠ ∅ do
10:         i := arg min_{i∈Δ_A^+(x)} φ_A(i,x)
11:         x := x ∪ {i}
12:         if x ∈ X and f(x) < f* then
13:             x* := x
14:             f* := f(x)
15:         end if
16:     end while
17:     return (x*, f*)
18: end procedure
```

the extension set and the time $T_{\varphi_A}(n)$ to compute the selection criterium, so that the resulting complexity is a polynomial of low order:

$$T_A(n) \in O(n(T_{\Delta_A^+}(n) + T_{\varphi_A}(n)))$$

In conclusione, constructive algorithms are really intuitive, simple to analyse and implement and tend to be efficient (unless the two main terms of their complexity are costly). However, they suffer from a strongly variable effectiveness: on some problems they guarantee an optimal solution; on other problems they may provide approximation guarantees; on most problems they provide solutions of extremely variable quality; sometimes, they even fail to return a feasible solution. Studying the problem before implementing any algorithm can determine through an *a priori* analysis whether some degree of effectiveness can be reached or not, and what features an algorithm should have to reach a good performance.

Constructive algorithms are certainly used when they guarantee optimal or near-optimal solutions, but also when the execution time must be very short or when the problem has a huge size or requires heavy computations at each step (for example, to compute the selection criterium). We shall see that they are also used as components of exchange or recombination algorithms, in order to solve auxiliary subproblems.

6.1.5 Using the objective as a selection criterium

As remarked in Section 2.1.3, the objective function can often be trivially extended from X to the whole of 2^B, or at least to an intermediate domain. If the objective function can be defined on the search space $\mathcal{F}_A$, it is a natural idea to adopt it as a selection criterium:

$$\varphi_A(i,x) = f(x \cup \{i\})$$

This extension allows to use the objective as a *guide* in the construction, that is as a metric to decide which element i should be added to the current subset in order to obtain a good solution.

The idea is natural, because the objective is exactly what we aim to minimise in the end. In the following, we consider a number of examples, in which applying this idea produces very different results.

The fractional knapsack problem

The *Fractional Knapsack Problem* (*FKP*) aims to select from a set of objects of identical volume v a subset of maximum value that fits in a knapsack of limited capacity V. It is, of course, a special case of the *KP*, in which the capacity simply corresponds to a cardinality constraint: the feasible solutions, in fact, are those with $|x| \leq \lfloor \frac{V}{v} \rfloor$.

Algorithm 2 Constructive heuristic for the FKP

```
1:  procedure GREEDYFKP(I)
2:      x := ∅
3:      while |x| < ⌊V/v⌋ do
4:          i := arg max_{i∈B\x} φ_i
5:          x := x ∪ {i}
6:      end while
7:      x* := x
8:      f* := f(x)
9:      return (x*, f*)
10: end procedure
```

The pseudocode of Algorithm 2 is an adaptation of Algorithm 1 to the specific problem considered. The termination condition states that subset x can be extended as long as $|x| < \lfloor V/v \rfloor$. Its acceptable extensions are all the element of $B \setminus x$. The selection criterium is the objective function, but can be replaced by an equivalent simpler expression, because it is additive. In fact:

$$f(x \cup \{i\}) = f(x) + \phi_i \Rightarrow \arg\max_{i \in B \setminus x} f(x \cup \{i\}) = \arg\max_{i \in B \setminus x} \phi_i$$

Since all visited subsets are feasible solutions, there is no need to test them for feasibility. Since every addition is an improvement, the last subset visited is the best solution found, and there is no need to update it at every iteration.

For example, consider the following instance: with $v_i = 1$ for all $i \in B$ and $V = 4$.

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1

The algorithm starts from $x := \emptyset$ and performs the following steps:

1. $x := \{a\}$
2. $x := \{a, d\}$
3. $x := \{a, c, d\}$
4. $x := \{a, c, d, e\}$

Then, it terminates, because the required cardinality has been reached. This algorithm always finds the optimal solution. Why?

The knapsack problem

Considering the general*KP*, a very similar algorithm can be designed, as reported in Algorithm 3. The only differences are that the set of possible extensions is in general smaller than $B \setminus x$, because the volume of its elements must not exceed the residual capacity of the knapsack. This also affects the termination condition.

Algorithm 3 Constructive Heuristic Pseudocode for the KP

```
1: procedure GREEDYKP(I)
2:     x := ∅
3:     while ∃i ∈ B \ x; v_i ≤ V − Σ_{j∈x} v_j do
4:         i := arg max_{i∈B\x; v_i ≤ V − Σ_{j∈x} v_j} φ_i
5:         x := x ∪ {i}
6:     end while
7:     x* := ∅
8:     f* := f(x)
9:     return (x*, f*)
10: end procedure
```

Consider the following instance: with capacity $V = 8$.

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1

The algorithm starts from $x := \emptyset$ and follows these steps:

1. $x := \{a\}$
2. $x := \{a,d\}$

terminating, because the capacity is exhausted. The solution obtained is not the optimal solution $x^* = \{a,c,e\}$, even if the algorithm is the same and the problem is extremely similar. Why?

The travelling salesman problem

Consider the *TSP* with the search space that relaxes the set of all partial solutions in order to make the inclusion test polynomial and the objective function as the selection criterium.

Algorithm 4 provides the pseudocode, that is once again an adaptation of the general scheme to this specific problem and search space. Since the objective function is additive, the selection criterium can be expressed by its variation, that is simply the cost of the added arc. In other words, the algorithm selects at each step the cheapest arc among those that do not form subtours and keep a degree ≤ 1 in all nodes. Only the last subset x can be a feasible solution, but it is not guaranteed to be, so the test must be performed, once, at the end.

Consider the example of Figure 6.3, that is a noncomplete graph, missing just arc $(1,3)$. Algorithm 4 starts from $x := \emptyset$ and performs the following steps:

1. since $\Delta_A^+(x) \neq \emptyset$, add $i := (3,5)$ to x
2. since $\Delta_A^+(x) \neq \emptyset$, add $i := (2,4)$ to x, noticing that arc $(5,3) \notin \Delta_A^+(x)$ as it would create a subtour

Algorithm 4 Constructive Heuristic Pseudocode for the TSP

```
1:  procedure GREEDYTSP(I)
2:      x := ∅
3:      x* := ∅
4:      f* := +∞
5:      while Δ_A^+(x) ≠ ∅ do
6:          i := arg min_{i∈Δ_A^+(x)} c_i
7:          x := x ∪ {i}
8:      end while
9:      if x ∈ X then
10:         x* := x
11:         f* := f(x)
12:     end if
13:     return (x*, f*)
14: end procedure
```

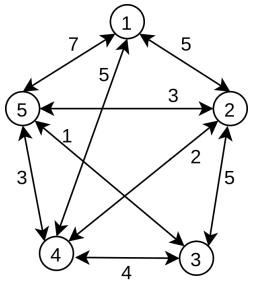


Figure 6.3: Numerical TSP example.

3. since $\Delta_A^+(x) \neq \emptyset$, add $i := (5,2)$ and update x ($(4,2) \notin \Delta_A^+(x)$)
4. since $\Delta_A^+(x) \neq \emptyset$, select $i := (4,1)$ and update x, notice that $(2,5),(4,5),(5,4),(3,4)$ and $(4,3) \notin \Delta_A^+(x)$

Now $\Delta_A^+(x) = \emptyset$, and the algorithm terminates without finding a feasible solution. Yet, feasible solutions exist (for example, a simple tour along the external arcs). In a complete graph, including arc $(1,3)$, the algorithm would certainly find a feasible solution. However, the cost of that arc could be huge (for example, 100), and the resulting solution would be nonoptimal, and possibly worse than the optimum without limit. Why?

The maximum diversity problem

The *MDP* is characterized by a cardinality constraint, as the *FKP*: it requires to select from a set of points a subset of k elements with the maximum sum of pairwise distances.

Algorithm 5 adapts the general scheme to this problem. The termination condition simply checks the cardinality of the current subset x. Any element of $B \setminus x$ extends x in an acceptable way. The objective function $f(\cdot)$ is not additive, but quadratic. When adding a new element i to a subset x, it can be split into the original value of the objective function $f(x)$ plus the distances of the newly added point i from each original point in x and from each point of x to i, plus the distance between i and i, that is 0:

$$f(x \cup \{i\}) = f(x) + 2\sum_{j \in x} d_{ij} + d_{ii} \rightarrow \arg\max_{i \in B \setminus x} f(x \cup \{i\}) = \arg\max_{i \in B \setminus x} \sum_{j \in x} d_{ij}$$

This allows to replace the objective function with a simpler expression as a selection criterium. The last subset visited is the only feasible solution found and can be directly returned.

Algorithm 5 Constructive Heuristic Pseudocode for the MDP

```
1: procedure GREEDYMDP(I)
2:     x := ∅
3:     while |x| < k do
4:         i := arg max_{i∈B\x} Σ_{j∈x} d_ij
5:         x := x ∪ {i}
6:     end while
7:     return (x, f(x))
8: end procedure
```

Unfortunately, this algorithm does not work. Consider the instance represented in Figure 6.4, whose optimal solution is $x^* = \{1,5,6\}$. At the first step, the algorithm takes the point with the maximum distance from an empty solution: that is 0 for every point. So any point is a valid first subset. Suppose that we try all of them, to avoid focusing on one (usually, the first) that can easily be nonoptimal. The second point is just the farthest from the first (in the figure, starting from $\{1\}$ the algorithm generates $\{1,7\}$). The third point is that with the maximum distance from the two original ones: let us take point 3 (but 4 is equivalent). The solution is nonoptimal. Actually, for any possible starting point the algorithm fails to get the optimal solution. Why?

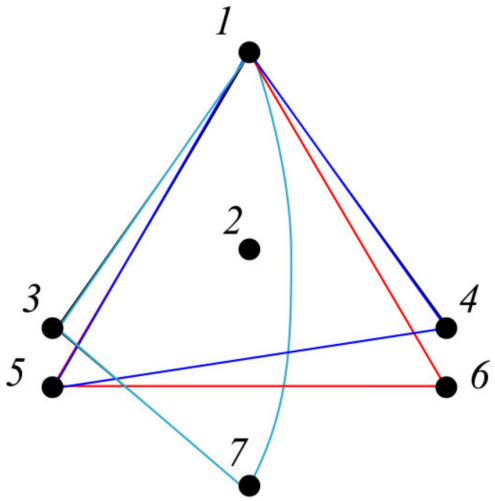


Figure 6.4: An example of Maximum Diversity Problem

6.2 Exact constructive algorithms

What features allow the basic constructive algorithm to find the optimum?

Having a search space coincident with the feasible region ($\mathcal{F}_A = X$) is not enough, because both the *FKP* and the *KP* have it, but the algorithm does not yield an optimal solution.

Having a cardinality-constrained problem would explain the failure on the *KP*, but not on the *MDP* and the *TSP*. Having an additive function does not explain the failure on the *TSP*.

There is **no general characterization** of the problems solved exactly by constructive algorithms, but there are characterization for wide classes of problems.

6.2.1 The additive case: matroids and greedoids

Assume that the objective function is additive, that is, there exists an auxiliary function defined on the ground set such that the value of the objective for a solution is given by the sum of the auxiliary function values on the elements of the solution:

$$\exists \phi : B \to \mathbb{N} : f(x) = \sum_{i \in x} \phi_i$$

Assume also that these solutions are the maximal subsets (**bases**) of the search space:

$$X = \mathcal{B}_F = \{y \in F : \nexists y' \in F : y \subset y'\}$$

meaning that any feasible solution is acceptable ($y \in X \Rightarrow y \in F$) and no acceptable subset ($y' \in F$) is strictly larger than a feasible solution. These assumptions sound abstract, but they are quite simple: many of the Combinatorial Optimization problems considered so far have an additive objective function. In most cases, the feasible solution is the last acceptable subset found: it cannot be enlarged. This is true for the *Max-SAT* and *TSP*: once all n variables are assigned a value, it is impossible to make further assignments; once a Hamiltonian circuit of n arcs is found, it is impossible to add further arcs. The *MDP*, *BPP* and *PMSP* have feasible solutions that are bases, but their objective functions are not additive. The *SCP* has an additive objective function, but the feasible solutions are not maximal subsets: in general, it is always possible to add a column to a subset that already covers all rows (and is therefore a feasible solution).

A separate discussion can be made for the *KP*, the *MCP* and the *MISP*, which have additive objective functions and solutions that are bases of the search space, but also solutions that are not bases, because they can be augmented (think of the empty set, for example). They do not match this assumption, but it is possible to remark that the nonmaximal solutions are by definition nonoptimal, and therefore can in principle be neglected. For these problems, the assumptions can be therefore applied with a slight extension, saying that the bases of the search space coincide with the "relevant" feasible solutions.

Under the assumptions above, the basic constructive algorithm finds an optimal solution for any additive objective function if and only if the pair (B,F) composed by the ground set and the search space is a **matroid embedding**. Since the definition of matroid embedding is rather complex, in the following we focus on some important simpler cases.

Greedoid

A **greedoid** is a pair (B,F) with B finite and $F \subseteq 2^B$ such that

- **trivial axiom**: $\emptyset \in F$, necessary as a greedy algorithm starts from the empty set.
- **accessibility axiom**: if $x \in F$ and $x \neq \emptyset$, then $\exists i \in x : x \setminus \{i\} \in F$
  In words, any acceptable subset can be destroyed removing its elements in a suitable order (and, conversely, built from scratch adding elements in a suitable order). Equivalently, any node of the construction graph can be reached from $x^{(0)} = \emptyset$. Notice that the axiom does not require that *any* element i can be removed from x: only that *at least one* can.

- **exchange axiom**: if $x, y \in F$ with $|x| = |y| + 1$, then $\exists i \in x \setminus y$ such that $y \cup \{i\} \in F$
  In words, any acceptable subset can be extended to another one adding a suitable element taken from any acceptable subset of larger cardinality.

Notice that the exchange axiom implies that all bases have the same cardinality, since any smaller base could be extended taking elements from a larger one, and would not be a base.

Consider the *FKP*. The trivial axiom certainly holds: the empty set is acceptable. The accessibility axiom holds as well: any subset of objects respecting the capacity can be reduced cancelling an element (actually any element) remaining inside the search space. The exchange axiom also holds: given a feasible subset of 3 elements and a feasible subset of 4 elements, one of the latter can be added to the former, and this will remain in the search space.

The axioms do not hold for the general *KP*, the *TSP* and many other problems. They actually hold for the *MDP*, but in this case the objective function is not additive!

The existence of a greedoid is a **necessary condition** for the optimality of the basic constructive algorithm. It is, however, **not sufficient**. The necessity of the trivial axiom is obvious: if the empty set is not acceptable, the constructive algorithm, that starts from it, cannot even be applied. The accessibility axiom is less obvious, but still simple: unreachable nodes could correspond to solutions (potentially optimal) that cannot be obtained by the constructive algorithm. The exchange axiom is crucial in guaranteeing the optimality of the final solution, based on choices taken in the single steps; understanding this would require to investigate the proof.

Notice that the axioms combine properties of the problem (referring to the objective function and the feasible region) with properties of the algorithm (referring to the search space).

Matroid

A **matroid** is a **set system** (B, F) with B finite and $F \subseteq 2^B$ such that

- **trivial axiom**: $\emptyset \in F$
- **hereditarity axiom**: if $x \in F$ and $y \subset x$ then $y \in F$
  Any acceptable subset can be built adding its element in any order (not only a specific chosen element).
- **exchange axiom**: if $x, y \in F$ with $|x| = |y| + 1$ then $\exists i \in x \setminus y$ such that $y \cup \{i\} \in F$
  Any acceptable subset can be extended taking a suitable element from any acceptable subset of larger cardinality.

The hereditarity axiom is a stronger version of accessibility; the other two axiom are the same as for greedoids. The existence of a matroid is a **sufficient condition** for the optimality of the basic constructive algorithm. Once again, this combines properties of the problem and of the algorithm.

Graphic matroid for the MSTP Consider Kruskal's search space for the *MSTP*, that is composed by all acyclic subsets of arcs (spanning forests):

$$F = \{x \subseteq B : x \text{ forms no cycles}\}$$

The trivial axiom holds: the empty set is acceptable because it forms no cycles. The hereditarity axiom holds: if x is acyclic, all of its subsets are also acyclic. The exchange axiom holds: if x and y are acyclic and $|x| = |y| + 1$ then one can always add a suitable edge of x to y without forming any cycle (at least one edge can be always added from a larger forest to a smaller one). See Figure 6.5: x is the set of blue arcs and y is the set of red arcs; edges (A, D), (C, G) and (D, H) belong to x and can be added to y ((B, F) cannot, and (E, F) would not augment y).

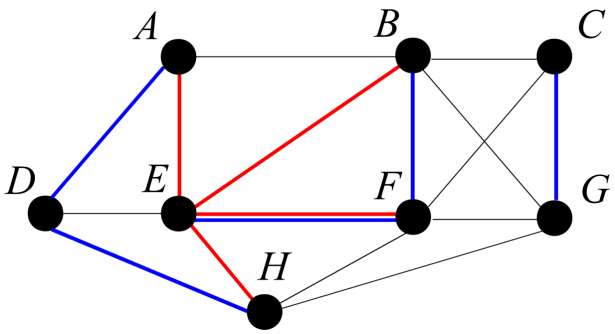


Figure 6.5: A matroid to solve the *MSTP*.

Uniform matroid for the *FKP*, not the *KP* Consider the search space for the *FKP*:

$$F = \{x \subseteq B : |x| \leq \lfloor V/v \rfloor\}$$

The trivial axiom holds: the empty set respects the cardinality constraint. The hereditarity axiom holds: if x respects the cardinality constraint, all of its subsets also respect it. The exchange axiom holds: if x and y respect the cardinality constraint and $|x| = |y| + 1$ one can always add a suitable element of x to y without violating the cardinality (in fact any element of x).

For the *KP*, the first two axioms hold, but the third one does not: for example, let $V = 6$ and $v = [3,3,2,2,1]$; the subsets $x = \{3,4,5\}$ and $y = \{1,2\}$ are in F, but no element of x can be added to y.

Greedoids with the strong exchange axiom

Instead of strengthening the accessibility axiom, one can operate strengthening the exchange axiom. The **greedoids with the strong exchange axiom** satisfy, of course, the trivial an the accessibility axiom, but also the following condition: (**strong exchange axiom**):

$$\begin{cases} x \in F, y \in \mathcal{B}_F \text{ such that } x \subseteq y \\ i \in B \setminus y \text{ such that } x \cup \{i\} \in F \end{cases} \implies \exists j \in y \setminus x : \begin{cases} x \cup \{j\} \in F \\ y \setminus \{j\} \cup \{i\} \in F \end{cases}$$

In words, given a basis and one of its subsets from which the basis is accessible (a subset in F), if there is an element that "leads astray" the subset from the basis (adding i to x makes it so that x will never reach y, which does not contain i), there must be another one which keeps it on the "right path" and it must be feasible to exchange the two element in the basis.

Again, it can be proved that these assumptions are a **sufficient condition** for the optimality of the basic constructive algorithm.

Greedoid with the strong exchange axiom for the *MSTP* A classical example of greedoid with the strong exchange axiom is given by

$$\begin{aligned} B &= \text{ set of edges of a graph} \\ F &= \text{ collection of trees including a given vertex } v_1 \end{aligned}$$

which yields Prim's algorithm for the *MSTP*.

In fact, the trivial axiom is satisfied, as the empty set is a degenerate tree reduced to vertex v_1. The accessibility axiom is satisied, because *some* edges (the leaves) can always be removed from a tree without disconnecting it. Finally, the exchange axiom holds in the strong form.

This is why Prim's algorithm solves the *MSTP* exactly, even if the hereditary axiom is violated (removing any edge from a tree does not always generate a tree).

6.3 Nonexact constructive algorithms

When all the search spaces devised to solve a problem violate the required algebraic properties, it is possible to try and guide the algorithm though the construction graph using a selection criterium more sophisticated than the objective function. In particular, the selection criterium could take into account somehow also the constraints that make the problem hard to solve. While this may fail getting an exact algorithm, quite often it will get *better* results, possibly even an approximation guarantee.

6.3.1 Pure constructive heuristics

Most of the selection criteria φ_A considered so far admitted equivalent expressions depending only on i: for example, in the KP, $\varphi_A(i,x) = f(x \cup \{i\})$ was replaced by ϕ_i; in the TSP, $f(x \cup \{(i,j)\})$ was replaced by c_{ij}. The algorithms in which the selection criterium depends only on the new element i are called **pure** constructive algorithm. In this section, we review other algorithms of this kind.

A constructive heuristic for the *KP*

The main reason why the matroid axioms are violated in the *KP* is that the objects have different volumes, and it is no longer true that a small cardinality subset can be enlarged by adding an element taken from a large cardinality subset. The problem here is the capacity constraint or, better, the different effect of the capacity constraint on different elements : promising objects have a large value, but also a small volume. Following this remark one can replace the naive selection criterium composed by the value with a smarter one, given by the **unitary value** $\varphi_A(i,x) = \frac{\phi_i}{v_i}$ of each object, that is the ratio of the value with the volume. The resulting algorithm can still perform very badly, but is in general much better and with a small modification becomes 2-approximated.

Consider the instance described by the following table with capacity $V = 8$:

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1
ϕ/v	1.4	0.67	2	1.67	4	1

The algorithm performs the following steps, where $V_r = V - \sum_{i \in x} v_i$ is the residual capacity:

1. $x := \emptyset, V_r = 8$
2. $x := \{e\}, V_r = 7$
3. $x := \{c,e\}, V_r = 5$
4. $x := \{c,d,e\}, V_r = 2$
5. $x := \{c,d,e,f\}, V_r = 1$
6. since $\Delta_A^+(x) = \emptyset$, terminate.

The value of the solution found is 14, while the optimal solution is $x^* = \{a, c, e\}$ with $f^* = 15$. The algorithm, therefore, is nonoptimal, but does not perform badly.

There are some critical cases, though. Consider the following table with capacity $V = 10$:

B	a	b
ϕ	10	90
v	1	10
ϕ/v	10	9

The algorithm chooses object a, which has the maximum ratio, and terminates, returning a solution of value 10, while the optimum is 90. There are instances with unlimitedly worse gap! The reason of the mistake is that the first discarded object has a large volume, but also a large value, exceeding that of the objects selected so far.

2-approximation guarantee for the *KP*

In order to obtain a 2-approximation guarantee, the algorithm must be slightly modified, following the lines of Algorithm 6. It still starts from the empty set and chooses at each iteration an element i of maximum unitary value. If the element respects the capacity, it is added to the solution, as in the basic algorithm; if the capacity is violated, on the contrary, instead of rejecting the item and proceeding with the next one, the algorithm terminates. From this point of view, the algorithm is clearly worse. However, before terminating, it considers a degenerate solution including only the discarded element, and returns the better of the two solutions found. This provides a better result in the worst case.

Algorithm 6 2-approximated algorithm for the KP

```
1:  procedure GREEDYKP(I)
2:      x := ∅
3:      while V > Σ_{i∈x} v_i do
4:          i^(t) := arg max_{i∈B\x^(t−1)} φ_i/v_i
5:          if v_{i^(t)} + Σ_{i∈x^(t−1)} ≤ V then
6:              x^(t) := x^(t−1) ∪ {i^(t)}
7:          else
8:              x' := {i^(t)}
9:              break
10:         end if
11:     end while
12:     return max[f(x), f(x')]
13: end procedure
```

In order to prove the approximation guarantee, we have to find an upper bound on the optimum (this is a maximisation problem) and a feasible solution providing a lower bound with a constant ratio from the upper bound. The first step consists in proving that the sum of the values of the two solutions overestimates the optimum:

$$f(x) + f(x') \geq f^*$$

Suppose that the capacity of the knapsack is enlarged so that is can host exactly the discarded object. The new problem is slightly different from the original one, and it surely has a larger

optimal value, since the solutions that are feasible for the original problem are feasible also for this one, but additional solutions, potentially better, become feasible. In particular, the solution obtained combining x and x' is optimal for the new problem, because it includes the objects of maximum unitary value and completely fills the capacity: any modification would introduce objects of smaller unitary value and possibly leave residual capacity.

Then, the lower bound with a fixed ratio to the upper bound is simply the better of the two solutions:

$$f_A = max[f(x), f(x')] \geq \frac{f(x) + f(x')}{2} \geq \frac{1}{2} f^*$$

since the maximum of two numbers is certainly never smaller than their arithmetic mean, which is half of their sum, and consequently at least half the optimum.

It is also clear that both x and x' could potentially be improved by adding other elements, instead of simply terminating the algorithm. However, even without such an improvement, the result is already 2-approximated.

A constructive heuristic for the *TSP*

An alternative to modifying the selection criterium is to modify the search space. This can be done, for example, for the *TSP*. We have already seen that the search space based on partial solutions does not admit a polynomial inclusion test (see Section 6.1.1) and its relaxation that only avoid subtours and vertex degrees larger than 1 provides very bad results. An alternative approach somewhat similar to Prim's algorithm for the *MSTP* consists in restricting the search space to very special partial solutions, that are anyway able to generate any feasible one.

The so called **Nearest Neighbour** (*NN*) algorithm defines $\mathcal{F}_A$ as the collection of all paths that start from a given node (for example node 1), plus the Hamiltonian circuits. These subsets of arcs have a structure that is simple to test and are promising, because any feasible solution includes some: they just have to be completed visiting the remaining nodes and going back to the first one.

Let N_x be the set of nodes visited in a path x. The acceptable extensions of x are all the arcs that go out of the last node and do not close a subtour:

$$\Delta_A^+(x) = \{(h,k) \in A : h = \text{Last}(x) \wedge (k \notin N_x \vee (k = \text{First}(x) \wedge N_x = N))\}$$

In other words, if h is the last node of path x, the acceptable extensions of x are are the arcs from h to any other node k not belonging to x, unless x has visited all nodes ($N_x \equiv N$) and k is the first visited ode.

A quick check to the axioms shows that the trivial axiom obviously holds: the empty set represents a degenerate path starting in node 1 and not reaching any other node. The accessibility axiom holds: removing the last arc still yields a path starting from node 1, so the reduced subset is still in the search space. The hereditarity axiom does not hold because most subsets of a path are not paths. Therefore, there is no matroid. Finally, the exchange axiom is violated, and consequently there is also no greedoid. Consequently, the algorithm cannot be exact.

Algorithm 7 starts with the empty set. Each iteration finds the arc of minimum cost among the possible extensions, that is among the arcs that go from the last node of the current path to a not yet visited node.

It is a very intuitive algorithm, with quadratic complexity: it performs n iterations, and in each one it scans at most n candidate arcs to determine and add the best one. We already know that on general graphs the feasibility of the *TSP* is $\mathcal{NP}$-complete, so no polynomial algoritm can guarantee to find a feasible solution, but on complete graphs this is always possible. If the triangular inequality holds, the algorithm is even $\log(n)$-approximated.

Algorithm 7 Nearest Neighbour heuristic for the *TSP*

```
1:  procedure NNTSP(I)
2:      x := ∅
3:      h := 1
4:      while true do
5:          (i, j) := arg min_{(h,k)∈Δ_A^+(x)} c_hk
6:          x := x ∪ {(i, j)}
7:          if j = 1 then
8:              break
9:          end if
10:     end while
11:     return x
12: end procedure
```

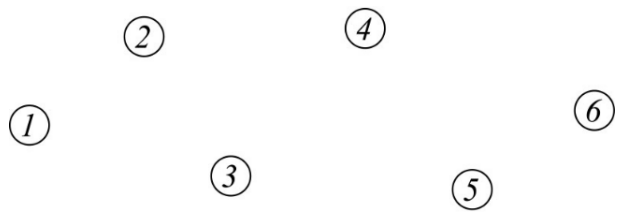
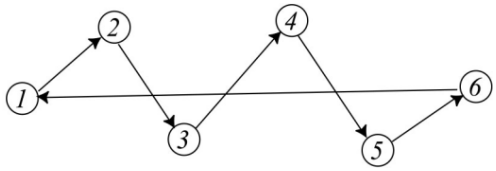
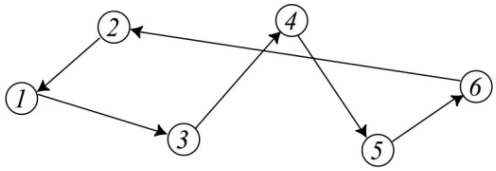


Figure 6.6: A complete graph with arcs not reported for clarity and Euclidean costs.

A numerical example Consider the graph in Figure 6.6. The *NN* heuristic starting from node 1 finds the suboptimal path in Figure 6.7 (a). Starting from node 2, it finds the suboptimal path in Figure 6.7 (b). Starting from any of the nodes yields a suboptimal solution: the optimal solution that corresponds to the convex envelope of the six points cannot be found by this algorithm.



(a) NN heuristic starting from node 1.



(b) NN heuristic starting from node 2.

Figure 6.7: NN heuristics on a complete graph starting from different nodes.

6.3.2 Adaptive constructive algorithms

When the selection criterium φ_A depends not only on the new element i, but also on the current solution x, it is called an **adaptive** algorithm, as it needs to update the value of the selection criterium from iteration to iteration, in order to adapt it to the current subset x. Of course, this makes the procedure more complex from the computational point of view. An example already considered was the basic constructive algorithm for the *MDP*. In the following we consider other examples.

An adaptive constructive algorithm for the *SCP*

Given a binary matrix and a cost vector associated to the columns of the matrix, the *SCP* aims to find a subset of columns that cover all the rows and have a minimum total cost. The objective function is obviously additive, but the solution are not maximal subsets, since any feasible solution can be augmented by additional columns, and actually the minimal feasible subsets are better.

Using the objective function as the selection criterium leads to choose columns that repeatedly cover the same rows. Moreover, the cost of a column is not sufficient to indicate whether it is promising or not: since the fundamental constraint is to cover all rows, a column should have a low cost, but also cover many rows. Finally, the rows to cover should not be always the same, but new ones at each iteration, with respect to rows already covered by the current subset of columns. In summary, the acceptable extension set $\Delta_A^+(x)$ includes only columns that covering additional rows, and the selection criterion $\varphi_A(i,x)$ should be adaptive, taking into account not only the cost c_i, but also the number $a_i(x)$ of "new rows", that is rows that are covered by i but not by x:

$$\varphi_A(i,x) = \frac{c_i}{a_i(x)}$$

This algorithm is not exact, but it is $\log(n)$-approximated. The proof is a bit involved. It requires an auxiliary weight vector defined on the rows of the matrix. At the beginning, this vector is initialised to $\theta_j = 0$ for all $j \in R$. At each iteration t each column i is evaluated with the criterion

$$\varphi_A(i,x^{(t-1)}) = \frac{c_i}{a_i(x^{(t-1)})}$$

and the minimum one is chosen and added to the current subset. Suppose that row j is covered for the first time by column i_j at iteration t_j. When a column is added to x, the weights of each row j newly covered is updated as follows:

$$\theta_j = \frac{c_{i_j}}{a_{i_j(x^{(t_j-1)})}}$$

Therefore, the total weight of the rows increases by c_{i_j} at step t_j, and, correspondingly, the cost of subset x increases by c_{i_j}, as the subset includes column i_j. Consequently, sum of the row weights is always exactly equal to the sum of the selected columns costs

$$f_A(x) = \sum_{i \in x} c_j = \sum_{j \in R} \theta_j$$

Since the the values $a_i(x^{(t-1)})$ decrease iteration after iteration, the row weights increase step by step. Since at each step at least one new row is covered, at step t there are $|R^{(t)}| \leq |R| - t$ uncovered rows. The columns of the optimal solution could cover these rows with cost f^*. This means that at least one of such columns has unitary cost $\leq f^*/|R^{(t)}|$.

However, the column chosen at each step is the one with minimum unitary cost, that is certainly not larger than the average cost given by the optimal solution. Therefore, some columns have a smaller cost and some a larger one. The cost of the final solution will not be larger than the sum of the upper estimates computed in each step of the algorithm

$$\theta_j \leq \frac{f^*}{|R^{(t_j)}|} \rightarrow \sum_{j \in R} \theta_j \leq \sum_{j \in R} \frac{f^*}{|R^{(t_j)}|}$$

In other words, the cost to cover each row j is not larger than the optimum divided by the number of rows uncovered at the step in which j gets covered.

Now, the number $|R^{(t)}|$ strictly decreases at each step and the sum can be overestimated reducing $|R^{(t)}|$ by 1 at each step. This yields a harmonic sum, which implies that the approximation ratio is limited by a logarithmic guarantee:

$$f_A = \sum_{j \in R} \theta_j \leq \sum_{j \in R} \frac{f^*}{|R^{(t_j)}|} \leq \sum_{r=|R|}^{1} \frac{f^*}{r} \leq (ln|R|+1) f^*$$

A numerical example Consider the *SCP* instance of Table 6.1.

c	25	6	8	24	12
A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	1	1	0
	1	0	0	1	0
	1	0	0	0	1

Table 6.1: An SCP instance.

The algorithm takes the following steps:

1. since $\varphi_A(i,x) = [4.16\ 2\ 4\ 12\ 12]$, select $i := 2$ and set $\theta_1 = \theta_2 = \theta_3 = 2$ which is less than $f^*/|R^{(0)}| = 25/6 = 4.16$.
2. since $\varphi_A(i,x) = [8.3\ -\ 8\ 12\ 12]$, select $i := 3$ and set $\theta_4 = 8$ which is less than $f^*/|R^{(1)}| = 8.3$.
3. since $\varphi_A(i,x) = [12.5\ -\ -\ 24\ 12]$, select $i := 5$ and set $\theta_6 = 12$ which is less than $f^*/|R^{(2)}| = 12.5$.
4. since $\varphi_A(i,x) = [25\ -\ -\ 24\ -]$, select $i := 4$ and set $\theta_5 = 24$ which is less than $f^*/|R^{(3)}| = 25$.
5. all the rows are covered, therefore $\Delta_A^+(x) = \emptyset$ and the algorithm terminates.

Hence, $f_A = \sum_{j \in R} \theta_j = 50$ and the approximation holds:

$$f_A \leq (ln|R|+1) f^* \approx 2.79 f^*$$

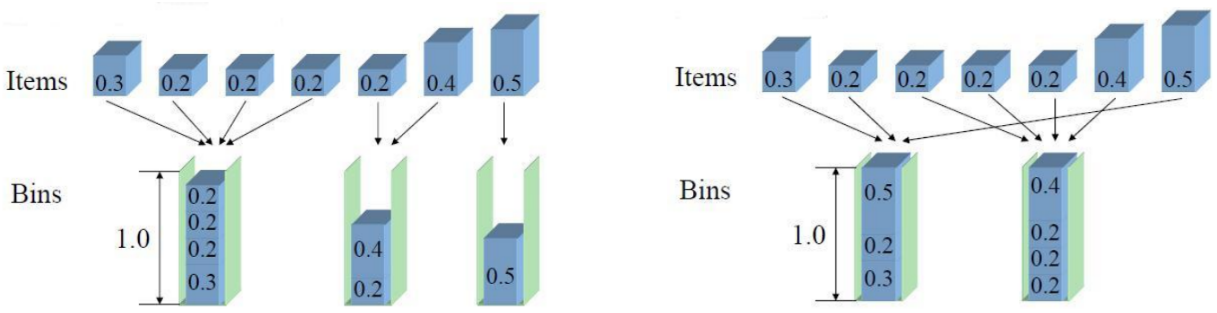
Adaptive heuristics for the *BPP*

The *BPP* requires to divide a set O of voluminous objects into the minimum number of containers of given capacity drawn from a set C. The ground set $B = O \times C$ includes all possible object-container assignments with one container for each object and with the total volume in each container not exceeding its capacity.

We consider two heuristics for this problems, that are adaptive, that is apply a selection criterium that depends on the current subset x. They also show that the selection criterium is a function in its most general sense, that is a relation between data and value, not necessarily expressed with an algebraic formula. Both heuristics define the search space $\mathcal{F}_A$ as the set of partial solutions: not all objects may be inserted in a bin, each object appears at most in one; no volume constraint is violated.

The objective function is certainly not additive, but the basic constructive algorithm may be applied anyway, starting from an empty subset and taking at every iteration the assignment that minimises the value of the objective function. However, this algorithm immediately finds that at the first step all acceptable extensions (assign a new object to a container respecting its capacity) increase the objective from zero to one container. In the following steps, the objective is still rather useless, as there are at most two classes of possible assignments: those that keep the same number of containers and those that increase it by 1. Such a flat objective function is not a reasonable selection criterium.

First-fit heuristic It is however possible to keep the constructive scheme using other, very simple, criteria. The **first-fit heuristic** (*FF*) splits the choice of the pair (object,container) into two subsequent choices. The selection criterium is still a function, even if it does not have a simple algebraic expression. The idea is trivially to choose the first (minimum index) unassigned object i and the first (minimum index) used container j that has enough residual capacity to host object i; if no container has enough residual capacity, j is the first unused container.



(a) First Fit solution. (b) Optimal solution.

Figure 6.8: Example of First-Fit heuristic on an instance.

The general scheme still applies: objects are assigned as long as the solution can be extended, that is as long as there are unassigned objects. The total number of steps is equal to the number of objects. Figure 6.8 (a) shows an example of the application of the First-Fit heuristic on a BPP instance.

The solution obtained is not optimal, as shown in Figure 6.8 (b). It can be proved, however, that the solution has a 2-approximation guarantee[1]. First, we find a lower bound on the optimum: in order to host all objects, at least $f^* \geq \lceil (\sum_{i \in O} v_i)/V \rceil$ containers are necessary (this is the number of containers needed assuming that the objects could be split). Following the rules of the *FF* heuristic, all used containers, apart possibly from the last one, have an occupied volume strictly greater than $V/2$. This can be proved by contradiction. Assume that two containers are less than half filled. This implies that all objects in the second one could fit in the first, but the algorithm always puts an object in the first container that can host it. So, this is impossible.

Consequently, the total volume exceeds that of the $f_A - 1$ "saturated" containers, that is larger than $V/2$:

$$\sum_{i \in O} v_i > (f_A - 1)\frac{V}{2}$$

[1] Actually, a better one, but the proof for the 2-approximation is simple.

which implies

$$(f_A - 1) < 2\frac{\sum_{i \in O} v_i}{V} \leq 2f^* \rightarrow f_a \leq 2f^*$$

Decreasing first-fit heuristic Of course, the approximation ratio $\alpha = 2$ proved above holds for any given permutation of the objects. Intuition would suggest to select first the smallest objects, in order to keep the value of the objective, $f(x \cup \{i\})$, as small as possible. This, however, neglects that all the objects must be assigned. In fact, it is better to select the largest objects first. By construction, each object in a container has a volume strictly larger than the residual capacity of all the previous containers. Keeping the smallest objects in the end guarantees that most containers have a small residual capacity, and therefore their volume is effectively exploited. So, the idea is to apply the First-Fit heuristic on a permutation of the objects to insert by decreasing volumes. This heuristic doesn't yield an optimal solution in general, but it provides a better worst-case approximation ratio:

$$f_A \leq \frac{11}{9} f^* + 1$$

as well as better empirical results on the average.

The remark that all objects must be "serviced" and, therefore, it is better to take into account the most "critical" ones first is a general property that holds for many other problems, as we shall see.

6.4 Extensions to the basic constructive scheme

In addition to designing a suitable search space and a suitable selection criterium, the basic scheme constructive of constructive heuristics can be extended also in more general ways. There are two basic enhancings. The first is to modify the **construction graph** replacing the rule that arcs correspond to adding a single element to an acceptable subset with a more general one: either adding more than one element or even adding elements to x but also removing some (provided that a "net increment" in the cardinality of the subset still occurs).

The second extension focuses on the **selection criterium**, making it dependent not only on the added element i and the current subset x, but on more sophisticated information. We will examine two possible extenstions, one using a *regret-based* function, that estimates the potential future loss associated to the inclusion of a certain element in the current subset and one using a *lookahead* function, that estimates the final value of the objective function implied by the addition of a certain element in subset x.

6.4.1 Extensions of the construction graph with small subsets

The basic constructive algorithm adds an element at a time to the current subset. In order to generalise this scheme, more than one element can be added at each step. In this case, the selection criterium $\varphi_A(B^+, x)$ (instead of $\varphi_A(i, x)$) depends on the current subset x and on each acceptable subset of elements of the ground set that are currently out of the solution

$$B^+ \subseteq B \setminus x$$

Another possibility is to add some elements from outside and remove some elements from the solution. In this case, the selection criterion will be a function $\varphi_A(B^+, B^-, x)$ with

$$B^+ \subseteq B \setminus x \text{ and } B^- \subseteq x \text{ with } |B^+| > |B^-|$$

As in the basic scheme, the construction graph has nodes corresponding to subsets of the ground set (the search space) and arcs corresponding to pairs of subsets such that the second one is obtained from the first one by applying an acceptable extension. The graph is still acyclic. Contrary to the basic scheme, the set of extensions $\Delta^+(x)$ is not composed by single elements of the ground set, but by pairs of subsets (B^+, B^-). This family of pairs of subsets must be designed in such a way that **optimising the selection criterion is a polynomial problem**:

$$\min_{B^+ \subseteq B \setminus x, B^- \subseteq x} \varphi_A(B^+, B^-, x)$$

While previously the set was clearly limited in size, now minimising this selection criterion can be computationally complex, because the subsets B^+ and B^- can be very numerous, possibly generating an unwanted exponential task. The two classical ways to limit this complexity are: i) to limit the size of the subsets B^+ and B^-, e.g. $|B^+| = 2$ and $|B^-| = 1$; ii) to define a specific family of set pairs such that minimising the selection criterion becomes a classic polynomial problem, that can be optimised efficienly (for example, a shortest path problem). In the following, we examine both cases.

A third search space for the TSP

In the previous sections we have discussed two constructive heuristics for the TSP based on different search spaces. The first one considered the subsets of arcs that forbid subtours and impose an outer and an inner degree not larger than one for all nodes. The second is the nearest neighbour heuristic, in which the search space is composed of all the paths starting in a certain node. The acceptable extensions are the arcs from the last visited node to an external one (or, if all nodes have already been visited, to the given starting node). In both algorithms, the selection criterium is the cost of the added arc. The first algorithm has a very bad performance, the second is reasonably good. We introduce now a third search space, and apply to it three different selection criteria.

Let us define the search space $\mathcal{F}_A$ is the set of all the circuits in the graph that include a given node, say node 1 (without loss of generality). A circuit cannot be obtained from another by adding a single arc to another circuit. It can be obtained by removing an arc (i, j) and adding two arcs $(i,k),(k, j)$. In this context, $B^+ = \{(i,k),(k, j)\}$ and $B^- = \{(i, j)\}$.

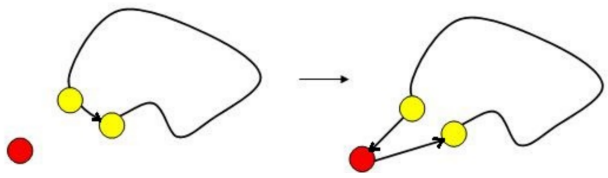


Figure 6.9: Building a new circuit from a given circuit.

The idea is to start from an empty set of arcs $x^0 = \emptyset$ (it would be more correct to think of it as a degenerate circuit composed by a self loop on node 1), and iteratively choose a node different from those spanned by the current solution, and an arc inside the solution, which shall be removed. The node and the arc identify automatically the two arcs that should be added. Therefore, the number of possible extensions in a complete graph is $(n - |x|)|x| \in O(n^2)$, deriving from the $(n - |x|)$ possible nodes and the $|x|$ possible arcs. If the graph is complete, after $n - 1$ iterations the solution will be a Hamiltonian circuit. Otherwise, the number of choices is smaller and the process could terminate in advance without returning any feasible solution. The following three algorithms are based on different selection criteria to determine the added node and the removed arc.

Cheapest insertion algorithm for the TSP The **cheapest insertion** (CI) heuristic uses as a selection criterion the objective function

$$\varphi_A(B^+, B^-, x) = f(x \cup B^+ \setminus B^-)$$

This is additive, hence extensible to the whole search space $\mathcal{F}_A$. Thanks to the additivity of $f(x)$

$$\arg\min_{B^+,B^-} \varphi_A(B^+, B^-, x) = \arg\min_{i,k}(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$$

where $k \in N \setminus N_x$ (the set of nodes spanned by x) ranges on the potential new nodes and $(s_i, s_{i+1}) \in x$ on the potentially removed arcs, as in Figure 6.10. The cost $f(x)$ of the original solution can be neglected, as it is constant for all extensions. Therefore, the computational cost to evaluate φ_A decreases from $\Theta(n)$ to $\Theta(1)$. This is fundamental, as the elementary insertion

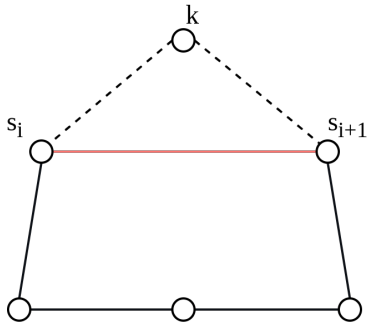


Figure 6.10: Example of an insertion.

step must be as fast as possible. It estimates $O(n^2)$ alternatives (trivially feasible in complete graphs, but checkable in constant time also in general ones), and the objective function can be computed updating it in constant time. The scheme of this algorithm is:

1. start with an the self-loop centered on node 1: $x^{(0)} = \{(1,1)\}$;
2. select the arc $(s_i, s_{i+1}) \in x$ and the node $k \notin N_x$ such that $(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$ is minimum;
3. if the circuit does not visit all nodes go back to point (2), otherwise terminate.

This algoritm is not exact, but it is 2-approximated for graphs that respect the triangle inequality (not on general complete graphs).

The CI algorithm performs $n-1$ steps and at each step t it evaluates $(n-t)t$ pairs $(node, arc)$ in constant time, performs the best addition/removal and decides whether to terminate. So, the overall complexity is $\Theta(n^3)$. However, it can be reduced to $\Theta(n^2 * log(n))$ collecting in a min-heap the best insertion cost for each external node: each of the n steps, then, selects the best insertion in $O(1)$ time, performs it and thus creates two new arcs, that provide insertion points for each external node, and possibly improve the value saved in the heap; each of the $O(n)$ updates of the heap takes $O(log(n))$ time.

Nearest insertion heuristic for the TSP The **nearest insertion** (NI) heuristic has the same search space $\mathcal{F}_A$ as the CI heuristic, but a different selection criterion. The CI algorithm tends to select nodes that are close to the circuit x: minimising $(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$, in fact, implies that $c_{s_i,k}$ and $c_{k,s_{i+1}}$ are small. We have also seen, discussing the FF and the DFF heuristics for the *BPP* that the selection criterion is not always a function that can be expressed analytically.

The NI heuristic splits the selection criterium is split into two subprocedures: the first one applies chooses the node k to add as

$$k = \arg\min_{l \notin N_x} (\min_{s_i \in N_x} c_{s_i,l})$$

while the second one chooses the arc to remove as

$$(s_i, s_{i+1}) = \arg\min_{(s_i,s_{i+1}) \in x} (c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$$

In other words, instead of minimising the cost of the new solution considering all pairs (node,arc), the criterium first choses the closest external node, then choses the "right" arc to break. These two steps repeat until the circuit visits all the nodes. Once again, the algorithms it not exact but 2-approximated under the triangle inequality.

It performs $n-1$ steps: at each step t it evaluates the distance of $(n-t)$ nodes from the circuit, each one in $\Theta(t)$ time; then, it selects the node at minimum distance, evaluates the removal of t arcs (each one in $\Theta(1)$ time), performs the best addition/removal and decides whether to terminate. So, the overall complexity is again $\Theta(n^3)$. It can be reduced to $\Theta(n^2)$ collecting in a vector the closest external node for each internal node: each of the $n-1$ steps selects the closest node in $O(n)$ time, inserts it creating two new insertions for each external node, which possibly improve the value saved in the vector; each of the $O(n)$ improvements takes $O(1)$ time. So, the algorithm has the same approximation guarantee as CI, but is faster.

Farthest insertion heuristic for the TSP The **farthest insertion** (FI) heuristic is very similar to the NI heuristic, but in a sense complementary: the nodes to be inserted are chosen maximising the distance from the current circuit (that is, the minimum distance from its nodes). This guarantees that the most problematic nodes - the farthest ones - are served in the best way. In a way, this is the same philosophy seen in the DFF algorithm for the BPP. Formally, the FI heuristic starts with a degenerate circuit centered on node 1 ($x^{(0)} = \{(1,1)\}$). At each step, it first selects the farthest node k from the cycle x

$$k = \arg\max_{l \notin N_x} (\min_{s_i \in N_x} c_{s_i,l})$$

and then the arc (s_i, s_{i+1}) that minimises the objective function

$$(s_i, s_{i+1}) = \arg\min_{(s_i,s_{i+1}) \in x} (c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$$

If the circuit does not visit all nodes it selects another one, otherwise it terminates. It can be proved that this algorithm is $\log(n)$-approximated under the triangle inequality, hence worse than the previous ones in the worst case. However, often it proves experimentally better. Its complexity is the same as the NI heuristic, $\Theta(n^2)$.

6.4.2 Extension of the construction graph using auxiliary subproblems

A second possible extension of construction graphs consists in adding subsets of element chosen by solving an auxiliary subproblem.

The Steiner tree problem

To illustrate this second possibility, we introduce a new problem called the **Steiner tree** problem (STP). Given an undirected graph $G = (V, E)$, a cost function $c : E \to \mathbb{N}$ on the edges and a subset of special vertices $U \subset V$, the problem requires to find a minimum cost tree connecting all the special vertices. An instance of the *STP* is represented in Figure 6.11, where the special nodes are represented in red. The nonspecial vertices can be included or not, arbitrarily.

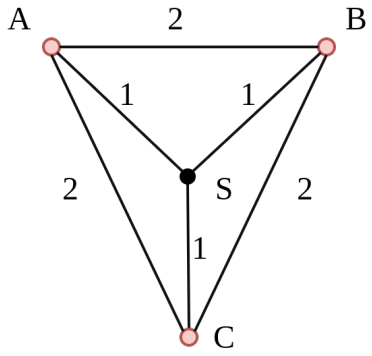


Figure 6.11: An example of STP instance.

At first sight, it seems reasonable to consider only the special vertices, remove the other ones, and find the minimum spanning tree on the former: the example in Figure 6.11 shows that this is not the right tactic in general. Moreover, in general the special vertices do not even induce a connected subgraph.

A basic constructive heuristic for the *STP* would start from an empty set of arcs and iteratively add one edge at a time, possibly with a search space inspired by Kruskal (the set of all forest spanning the special vertices), or a search space inspired by Prim (the set of all trees including a given pivotal vertex). This does not work well because it easily yields solutions with redundant edges: the algorithm has a really hard time trying to discern useful edges from redundant ones. The algorithm "knows" that an edge is cheap, but not whether it is useful or not to reach the special vertices.

Distance heuristic for the STP The critical point is to distinguish useful edges from reduntant ones, besides considering their cost. The **distance heuristic** (DH) focuses on the constraint that all special vertices must be spanned by the solution. Its idea is to start from a special vertex, say 1, and step after step add a new special vertex j connected with a path to the previous ones. In other words, the DH iteratively adds a path B^+ from the solution x (a tree) to a special vertex, instead of a single edge. In this way, x remains a tree, but spans a new special vertex. The minimum cost path can be computed efficiently at each step using Dijkstra's or any other shortest path algorithm.

Consider Figure 6.12; the algorithm works as follows:

1. start from a degenerate tree composed by a single special vertex A: $x^{(0)} := \emptyset$;
2. add the closest special vertex B through path (a,e,d,b): $x := \{(a,e),(e,d),(d,b)\}$.
3. add the closest special vertex G through path (g,h,d): $x := \{(a,e),(e,d),(d,b),(g,h),(h,d)\}$.
4. since all special vertices are in the solution, terminate.

The solution obtained in this example is optimal. In general, the DH algorithm is 2-approximated. It is equivalent to computing a minimum spanning tree on a graph with vertices reduced to the special vertices and edges corresponding to minimum paths.

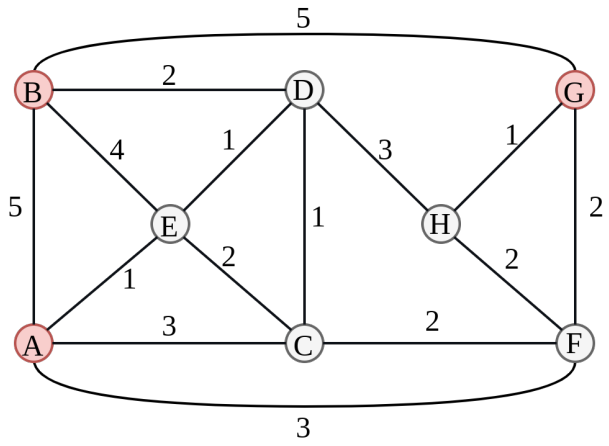


Figure 6.12: A second example of STP instance.

Let us see a **counterexample to optimality** regarding the DH algorithm for the STP. In the complete graph of Figure 6.13 all vertices except vertex 1 are special, so $U = V \setminus \{1\}$. The cost function is

$$c_{uv} = \begin{cases} (1+\varepsilon)M & u = 1 \vee v = 1 \\ 2M & u, v \in U \end{cases}$$

where M is a nonnecessary constant used only to obtain integer cost for any ε.

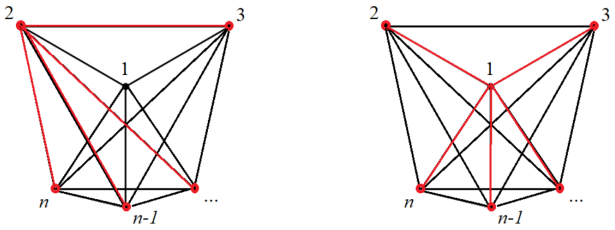


Figure 6.13: Counterexample to DH optimality.

The optimal solution is the spanning star centered in 1, with cost

$$f^* = (n-1) \cdot (1+\varepsilon)M$$

(see Figure 6.13 on the right), while the DH returns a star spanning only the special vertices and costing:

$$f_{DH} = (n-2) \cdot 2M$$

(see Figure 6.13 on the left). The approximation ratio is

$$\rho_{DH} = \frac{f_{DH}}{f^*} = \frac{n-2}{n-1} \cdot \frac{2}{1+\varepsilon} < 2$$

and converges to 2 as n increases and ε decreases.

6.4.3 Extensions of the selection criterion: regret functions

Another way to extend the basic scheme of constructive algorithms, even keeping a standard construction graph with arcs corrisponding to the addition of a single element, is to define more sophisticated selection criteria. The basic limitation of constructive algorithms is that decisions

taken in the first steps strongly influence the rest of the computation. In particular, they can severely restrict the future feasible choices. Consider the insertion of objects into containers in the *BPP*: quite clearly, the first insertions limit the possibility to put other objects in the same containers. As well, in the *TSP* choosing to service certain nodes impose a starting shape to the circuit, and only one arc of that shape can be changed at each step to reach other nodes. In the *CMSTP*, since the capacity of each subtree has to be respected, early links could make completely fill some subtrees and make them unavailable to host vertices not yet added, even if they would be particularly fit.

A selection criterion can be designed to try and take this aspect into account implicitly: we have seen that the *DFF* heuristic for the *BPP* starts assigning the largest objects, to keep many containers available for the smallest ones. As well, the *FI* heuristic for the *TSP* visits the farthest nodes first, in order to obtain a circuit offering good choices also for the last nodes.

A typical **regret-based** heuristic requires to:

1. partition the acceptable extensions $\Delta_A^+(x)$ into disjoint classes of choices **such that at least one must belong to any feasible solution**; for example, since the ground set of the *BPP* is made of pairs (object,container), one can divide the extensions into classes each referring to an object (all the possible assignments of the first object, of the second object, and so on);
2. compute the value of a basic selection criterion for all extensions and divide them according to the classes obtained above;
3. compute for each class the **regret**, defined as the difference between the second-best and the best extension in each class, or, alternatively, the difference between the average (possible weighted) and the best choice; such a function estimates the damage incurred by postponing the best choice.
4. choose the class with maximum regret and the best choice in that class (therefore, in general not the best overall).

A large regret means that not choosing the best option in the class might force in later steps to choose the second-best option (or even a following one). If the regret is large, the quality of these choices will be much worse, suggesting that the best choice must be taken immediately, as long as it is feasible. The basic requirement is that at least one choice for each class must be taken: for example, at least an assignment for each object, at least an outgoing (or ingoing) arc for each node, at least a machine for each task, etc...

Example on the CMSTP Consider the *CMSTP* with ground set $B = V \times T$, that is the pairs (vertex,subtree), with the constraint that each vertex must belong to exactly one subtree. Consider the example of Figure 6.14 with root in r, a uniform vertex weight $\forall v \in V\ w_v = 1$ and a capacity $W = 2$.

The feasible solutions are spanning trees such that all subtrees appended to the root vertex r have a total weight not larger than W. Let the search space F include all partial solutions.

In the first iteration, each vertex can belong to one out of four subtrees: $(1,1)$, $(1,2)$, $(1,3)$, $(1,4)$, $(2,1)$, ..., $(4,4)$ with different costs. The basic constructive heuristic puts vertex 2 in subtree 1 (chosen lexicographically), vertex 3 in subtree 2 (adding 3 to the first subtree would require a cost of 100 to link 2 and 3, the same holds for using subtree 3 or 4); vertex 4 must be added to subtree 1 connecting with vertex 2 at cost 2 (building a new subtree would cost 100); vertex 5 has to go in subtree 2 with 3, since subtree 1 is currently full and building a third

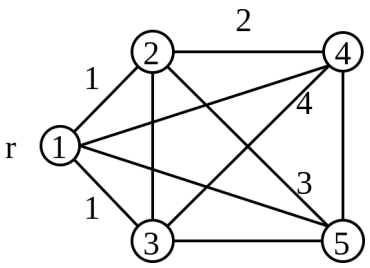


Figure 6.14: An instance of CMSTP problem. The graph is complete: the arcs not reported have have weight 100.

subtree would cost the same. Thus:

$$f(x) = 1+1+2+100 = 104$$

The regret algorithm (using the difference between second-best and best choice) puts vertex 2 in subtree 1 and vertex 3 in subtree 2 (the regrets are all equal to zero); then:

- the regret of vertex 4 is the difference $c(4,2) - c(4,1) = 4-2 = 2$
- the regret of vertex 5 is the difference $c(5,2) - c(5,1) = 100-3 = 97$

So the algorithm puts vertex 5 in subtree 1 with 2 and vertex 4 in subtree 2 with vertex 3 and obtains

$$f(x) = 1+3+1+4 = 9$$

The regret technique works very well when the problem is tightly constrained. In loosely constrained instanes it can lead to ineffective results.

6.4.4 Extensions of the selection criterion: roll-out heuristics

Roll-out heuristics are based on the idea of making a look-ahead, that is asking would happen in the end when a new element is chosen and added to the solution. These heuristics have been proposed in 1997 by Bertsekas and Tsitsiklis. They are *meta*-algorithms, meaning that they depend on a basic constructive algorithm A. They:

- start from an empty subset $x^{(0)} = \emptyset$;
- at each step t
  - extend the subset in each feasible way: $\forall i \in \Delta_A^+(x)\ x^{(t-1)} \cup \{i\}$;
  - apply the basic heuristic to each extended subset and compute the resulting solution $x_A(x^{(t-1)} \cup \{i\})$;
  - use the value of the solution as the selection criterion to choose $i^{(t)}$ $\varphi_A(i,x) = f(x_A(x^{(t-1)} \cup \{i\}))$;
- terminate when $\Delta_A^+(x)$ is empty.

In other words, these heuristics try every feasible move, look at the final result and go back to choose the move with the best final result. So, lots of attempts are done and the basic heuristic is run lots of times. Under very general conditions, the result of the roll-out heuristic dominates

that of the basic heuristic. The complexity remains polynomial, but much larger than that of the basic heuristic: in the worst case, in fact,

$$T_{ro} = |B|^2 T_A$$

because the process takes $O(|B|)$ steps and at each one the basic heuristic is run for each of the $O(|B|)$ acceptable extensions.

Example: roll-out for the SCP Given the *SCP* instance of Table 6.2, the feasible solutions are subsets of columns that cover all the rows. We know a good log-approximated algorithm, based on the ratio between the cost and the number of newly covered rows.

c	25	6	8	24	12
A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	1	1	0
	1	0	0	1	0
	1	0	0	0	1

Table 6.2: An instance of the SCP.

Applying the roll-out heuristic, each of the five columns is taken as part of the solution and the original heuristic algorithm is applied to it:

1. start with the empty subset $x^{(0)} = \emptyset$;
2. for each column i, apply the constructive heuristic starting from subset $x^{(0)} \cup \{i\} = \{i\}$;
   - for $i = 1$, obtain $x_a(\{1\}) = \{1\}$ of cost $f_A(\{1\}) = 25$
   - for $i = 2$, obtain $x_a(\{2\}) = \{2,3,5,4\}$ of cost $f_A(\{2\}) = 50$
   - for $i = 3$, obtain $x_a(\{3\}) = \{3,2,5,4\}$ of cost $f_A(\{3\}) = 50$
   - for $i = 4$, obtain $x_a(\{4\}) = \{4,2,5\}$ of cost $f_A(\{4\}) = 43$
   - for $i = 5$, obtain $x_a(\{5\}) = \{5,2,3,4\}$ of cost $f_A(\{5\}) = 50$
3. the best solution is the first one, therefore $i^{(1)} = 1$;
4. since all rows are covered, the algorithm terminates.

The scheme can be generalised by applying several basic heuristics $A^{[1]}, \cdots, A^{[l]}$ and increasing the number of look-ahead steps, i.e. using $x^{(t-1)} \cup B^+$ with $|B^+| > 1$. Both generalisations increment the computational time, but the overall scheme does not change significantly:

- start with an empty subset $x^{(0)} = \emptyset$
- at each step t
  - for each possible extension $B^+ \in \Delta_A^+(x^{(t-1)})$ apply each basic algorithm $A^{[i]}$ starting from $x^{(t-1)} \cup B^+$.
  - the selection criterion is $\min_i f_{A^{[i]}}(x^{(t-1)} \cup B^+)$
  - use the value of the solution as the selection criterion to choose $i^{(t)}$ $\varphi_A(i,x) = \min_i f_{A^{[i]}}(x^{(t-1)} \cup B^+)$
- terminate when $\Delta_A^+(x)$ is empty

6.5 Destructive heuristics

Destructive heuristics are complementary to constructive ones. Their general scheme is:

- start with the full ground set: $x^{(0)} := B$
- remove a selected element at a time:
  - so as to remain within the search space $\mathcal{F}_A$, in other words $\Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in \mathcal{F}_A\}$
  - so as to maximize a selection criterion $\varphi_A(i,x)$, usually a cost reduction
- terminate when $\Delta_A^+(x) = \emptyset$

A destructive heuristic for minimisation problems can be described by Algorithm 8. This algo-

Algorithm 8 Destructive Heuristic Pseudocode

```
 1: procedure STINGY(I)
 2:     x := B
 3:     x* := B
 4:     if x ∈ X then
 5:         f* := f(x)
 6:     else
 7:         f* := +∞
 8:     end if
 9:     while Δ_A^+(x) ≠ ∅ do
10:         i := arg max_{i∈Δ_A^+(x)} φ_A(i,x)
11:         x := x \ {i}
12:         if x ∈ X and f(x) < f* then
13:             x* := x
14:             f* := f(x)
15:         end if
16:     end while
17:     return (x*, f*)
18: end procedure
```

rithm is often also called *stingy*, as it starts with the entire ground set in x and proceeds throwing away the most "expensive" objects. For example, in the *SCP* one can start with all columns and iteratively remove the most expensive column that covers only rows covered by other columns, until no such column can be found. In the *MST* problem, the stingy algorithm is exact, as Kruskal's or Prim's algorithm: starting from the full set of edges, costly edges get iteratively discarded from the solution provided that the vertices remain connected. The reason why it is rarely taught is that it is computationally more expensive: to test whether the graph remains connected requires a visit, that is more costly than the acyclicity test run by union-find sets in Kruskal or the automatic generation of only feasible solutions in Prim.

6.5.1 Why are they less used than constructive heuristics?

When the solutions are much smaller than the ground set ($|x| << |B|$), a destructive heuristic requires a larger number of steps than a constructive one. It is also more likely to make wrong

decision at an early step, given that there are more possible choices. Sometimes, it requires more time to evaluate $\Delta_A^+(x)$ and $\varphi_A(i,x)$ at each single step.

Nonetheless, whenever a constructive heuristic returns redundant solutions, it is useful to "append" a destructive heuristic at its end as a post-processing phase. This auxiliary destructive heuristic starts from a solution x of the constructive heuristic, instead of B. It typically adopts as a search space the feasible region:

$$\mathcal{F}_A = X \implies \Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in X\}$$

and as a selection criterion the objective function

$$\varphi_A(i,x) = f(x \setminus \{i\})$$

Usually, it terminates after very few steps.

Constructive/destructive heuristic for the SCP Consider the following SCP instance:

c	25	8	24	12
A	1	0	0	0
	1	0	0	0
	1	1	0	0
	0	1	1	0
	0	0	1	0
	0	0	0	1

The constructive heuristic selects, in order, columns $1,2,4$ and 3, which is redundant. The auxiliary destructive heuristic removes column 2 and provides the optimal solution $x^* = \{1,3,4\}$.

6.6 Exercises

6.6.1 Exercise 1

Given an undirected graph $G = (V, E)$ and a cost function defined on its edges $c : E \to \mathbb{N}$, the *Maximum Matching Problem* consists in finding a subset of nonadjacent edges of maximum total cost. Suppose that the ground set B for this problem has been defined as the set of edges E, and the search space $\mathcal{F}$ as the collection of all subsets of nonadjacent edges (i. e., it coincides with the feasible region).

Does $(B, \mathcal{F})$ satisfy the *trival axiom*?

Does $(B, \mathcal{F})$ satisfy the *accessibility axiom*?

Does $(B, \mathcal{F})$ satisfy the *hereditarity axiom*?

Does $(B, \mathcal{F})$ satisfy the *exchange axiom*?

What can be deduced from this?

Solution The trivial axiom is obviously satisfied: an empty set of edges includes no pair of adjacent edges. The hereditarity axiom (and therefore the accessibility one) is also satisfied, because removing edges from a set does not create a new pair of adjacent edges. The exchange axiom is not satisfied, because it is easy to find counterexamples. For example, let $x = \{(2,3)\}$ and $y = \{(1,2),(3,4)\}$: the two subsets are both part of the search space and the latter is larger than the former, but none of the edges in the latter can be added to former remaining in the search space. As a confirm, the bases of the search space, that is the maximal matchings, can have different cardinalities. Since the objective function is additive and the (potentially optimal) feasible solutions are the bases of the search space, the *Maximum Matching Problem* is not in general solved exactly by the greedy algorithm.

6.6.2 Exercise 2

Given an undirected graph $G = (V, E)$ and a weight function defined on its vertices $c : E \to \mathbb{N}$, the *Maximum Clique Problem* consists in finding a subset of pairwise adjacent vertices of maximum total weight. Suppose that the ground set B for this problem has been defined as the set of vertices V, and the search space $\mathcal{F}$ as the collection of all subsets of pairwise adjacent vertices (i. e., it coincides with the feasible region).

Does $(B, \mathcal{F})$ satisfy the *trival axiom*?

Does $(B, \mathcal{F})$ satisfy the *accessibility axiom*?

Does $(B, \mathcal{F})$ satisfy the *hereditarity axiom*? Does $(B, \mathcal{F})$ satisfy the *exchange axiom*?

Solution For the trivial axiom, one can conventionally say that the vertices of an empty set are adjacent or that they are not, as it seems preferable. The hereditarity axiom (and, therefore, the accessibility one) is certainly satisfied, because when vertices are removed from a subset, the remaining ones still are pairwise adjacent. The exchange axiom is not satisfied, because it is easy to find counterexamples. Let $x = \{(1)\}$ and $y = \{2,3\}$: if edges $(1,2)$ and $(1,3)$ do not exist, none of the vertices from y can be added to x and still form a clique. In fact, maximal cliques can have different cardinalities. Since the objective function is additive and the (potentially optimal) feasible solutions are the bases of the search space, the *Maximum Clique Problem* is not in general solved exactly by the greedy algorithm.

6.6.3 Exercise 3

Considering the *BPP*, assume as ground set the collection of all object-container pairs ($B = E \times C$) and as search space the collection of all partial solutions, that is subsets of pairs that assign each object to at most one container without exceeding the capacity of the containers. Discuss whether ground set and search space satisfy the *trivial*, the *hereditarity*, the *accessibility* and the *exchange* axiom. What can be deduced from the discussion?

Solution For the trivial axiom, an empty set is certainly a partial solution: all objects are assigned to zero containers (therefore, not more than 1) and the occupied volume is zero for all containers (therefore, not exceeding the capacity). The hereditarity axiom (and, therefore, the accessibility one) is certainly satisfied, because when objects are removed from the container to which they are assigned, the remaining ones still have at most one assignment and the capacity of the containers still is respected. Considering the exchange axiom, the partition constraint is not a problem: if a subset has one assignment more than a second one, then that assignment can be added to the first. However, the capacity of the containers could be easily violated. A counterexample very similar to the one discussed for the KP can be proposed. Suppose that in subset x an object completely fills the first container, while in subset y two other objects completely fill the first container: none of the two assignments of y can be feasibly added to x. Since the objective function is not additive, this discussion actually does not prove or disprove anything on the optimality of the greedy algorithm, that is however clearly nonexact.

6.6.4 Exercise 4

Given a directed graph $G = (N, A)$, a cost function defined on its edges, $c : A \to \mathbb{N}$, a source node s and a destination node t, the *Shortest Path Problem* requires to find a directed path of minimum total cost from s to t. Assume the set of arcs A as the ground set and the collection of all paths starting from s as the search space and discuss whether ground set and search space satisfy the *trivial*, the *hereditarity*, the *accessibility* and the *exchange* axiom. What can be deduced from the discussion?

Solution The trivial axiom is satisfied, as an empty set of arcs can be seen as a (degenerate) path from s. The hereditarity axiom is violated, because removing arcs from a path usually does not yield a path. The accessibility axiom is satisfied, because removing the last arc leaves a path starting from s. Finally, the exchange axiom is violated, because counterexamples can be easily built (just take two disjoint paths). This proves that in general the greedy algorithm does not solve exactly the *Shortest Path Problem*.

6.6.5 Exercise 5

Given the following instance of the *Knapsack Problem* with capacity $V = 10$:

Objects	a	b	c	d	e	f	g
Prize ϕ	10	3	5	12	7	6	8
Volume v	5	1	2	4	3	1	6

solve it with the trivial constructive algorithm that uses the objective function as a selection criterium, and with the standard constructive algorithm that uses the ratio of object prize to object volume. For the sake of simplicity, break ties with a lexicographic rule on the objects.

Solution In the first case, since the objective function is additive, the selection criterium can be reduced to the prize of the single objects: $\varphi_A(i,x) = \phi_i$. The algorithm proceeds as follows, updating at each step $\Delta_A^+(x) = \{i \in E \setminus x : v(x) + v_i \leq V\}$ and x:

1. start from $x = \emptyset$;
2. select arg $\max_{i \in \Delta_A^+(x)} \phi_i = d$, and obtain $x = \{d\}$;
3. select arg $\max_{i \in \Delta_A^+(x)} \phi_i = a$, and obtain $x = \{d, a\}$;
4. select arg $\max_{i \in \Delta_A^+(x)} \phi_i = f$, and obtain $x = \{d, a, f\}$;
5. stop, because $\Delta_A^+(x) = \emptyset$.

Solution $x = \{a, d, f\}$ has a total value of $f(x) = 28$.

The second algorithm computes the ratios $\varphi_A(i,x) = \phi_i / v_i$.

Objects	a	b	c	d	e	f	g
$\varphi_A(i,x)$	2	3	5/2	3	7/3	6	4/3

and proceeds as follows:

1. start from $x = \emptyset$;
2. select arg $\max_{i \in \Delta_A^+(x)} \phi_i / v_i = f$, and obtain $x = \{f\}$;
3. select arg $\max_{i \in \Delta_A^+(x)} \phi_i / v_i = b$ (applying the lexicographic rule to choose between b and d), and obtain $x = \{f, b\}$;
4. select arg $\max_{i \in \Delta_A^+(x)} \phi_i / v_i = d$, and obtain $x = \{f, b, d\}$;
5. select arg $\max_{i \in \Delta_A^+(x)} \phi_i / v_i = c$, and obtain $x = \{f, b, d, c\}$;
6. stop, because $\Delta_A^+(x) = \emptyset$.

Solution $x = \{b, c, d, f\}$ has a total value of $f(x) = 26$. In this case, the smarter heuristic actually obtained a worse result.

6.6.6 Exercise 6

Given the following instance of the *Parallel Machine Scheduling Problem* (*PMSP*) with 3 machines:

Task	a	b	c	d	e	f	g
d	9	3	7	4	5	4	10

solve it with the basic constructive heuristic that assigns tasks to machines using the objective function as the selection criterium. For the sake of simplicity, break ties with a lexicographic rule first on the machines, then on the tasks.

Solve it again choosing first the task by decreasing durations d_i, and then the machine with the objective function. In both choices, break ties lexicographically.

Solution The set of possible extensions $\Delta_A^+(x)$ includes the pairs $(t,m) \in T \times M$ such that task t has not yet been assigned to a machine in x. The basic constructive heuristic proceeds as follows:

1. start from $x = \emptyset$;
2. select $\arg \min_{(t,m)\in\Delta_A^+(x)} f(x \cup \{(t,m)\}) = (b,1)$, because for all choices the completion time is equal to the duration d_t of the chosen task, b is the task with the shortest duration and machine 1 is chosen lexicographically;
3. select $\arg \min_{(t,m)\in\Delta_A^+(x)} f(x \cup \{(t,m)\}) = (d,2)$, because d is the remaining task with the shortest duration (breaking lexicographically the tie with f) and all machines except for 1 give the minimum completion time, but machine 2 is lexicographically the first;
4. select $\arg \min_{(t,m)\in\Delta_A^+(x)} f(x \cup \{(t,m)\}) = (f,3)$;
5. select $\arg \min_{(t,m)\in\Delta_A^+(x)} f(x \cup \{(t,m)\}) = (e,1)$;
6. select $\arg \min_{(t,m)\in\Delta_A^+(x)} f(x \cup \{(t,m)\}) = (c,2)$;
7. select $\arg \min_{(t,m)\in\Delta_A^+(x)} f(x \cup \{(t,m)\}) = (a,3)$;
8. select $\arg \min_{(t,m)\in\Delta_A^+(x)} f(x \cup \{(t,m)\}) = (g,1)$;

The resulting solution is $x = \{(b,1),(e,1),(g,1),(c,2),(d,2),(a,3),(f,3)\}$ with completion time $f(x) = \max(18,11,13) = 18$.

The constructive heuristic that considers tasks by nonincreasing durations proceeds as follows:

1. start from $x = \emptyset$;
2. select first $\arg\max_{t\in T} d_t = g$ and then $\arg\min_{m\in M} f(x \cup \{(g,m)\}) = 1$ (lexicographically);
3. select first $\arg\max_{t\in T} d_t = a$ and then $\arg\min_{m\in M} f(x \cup \{(a,m)\}) = 2$ (lexicographically);
4. select first $\arg\max_{t\in T} d_t = c$ and then $\arg\min_{m\in M} f(x \cup \{(c,m)\}) = 3$;
5. select first $\arg\max_{t\in T} d_t = e$ and then $\arg\min_{m\in M} f(x \cup \{(e,m)\}) = 3$;
6. select first $\arg\max_{t\in T} d_t = d$ (lexicographically) and then $\arg\min_{m\in M} f(x \cup \{(d,m)\}) = 2$;
7. select first $\arg\max_{t\in T} d_t = f$ and then $\arg\min_{m\in M} f(x \cup \{(f,m)\}) = 1$;
8. select first $\arg\max_{t\in T} d_t = b$ and then $\arg\min_{m\in M} f(x \cup \{(b,m)\}) = 3$.

The resulting solution is $x = \{(f,1),(g,1),(a,2),(d,2),(b,3),(c,3),(e,3)\}$ with completion time $f(x) = \max(14,13,15) = 15$.

6.6.7 Exercise 7

Given the following instance of the *Bin Packing problem* (*BPP*):

Items	a	b	c	d	e	f
Volumes	34	4	10	30	2	12

Bin capacity
40

apply the First-Fit and the Decreasing First-Fit algorithms to solve the problem.

Solution The First-Fit heuristic proceeds as follows:

1. start from $x = \emptyset$;
2. put object a into container 1: the residual capacity becomes $40 - 34 = 6$;
3. put object b into container 1: the residual capacity becomes $6 - 4 = 2$);
4. put object c into container 2: the residual capacity becomes $40 - 10 = 30$);
5. put object d into container 2: the residual capacity becomes $30 - 30 = 0$);
6. put object e into container 1: the residual capacity becomes $2 - 2 = 0$);
7. put object f into container 3: the residual capacity becomes $40 - 12 = 28$);

The resulting solution is $x = \{(a,1),(b,1),(e,1),(c,2),(d,2),(f,3)\}$ and the value of the objective is $f(x) = |x| = 3$. It is optimal, as $\lceil \sum_{i \in E} v_i / V \rceil = \lceil 92/40 \rceil = 3$.

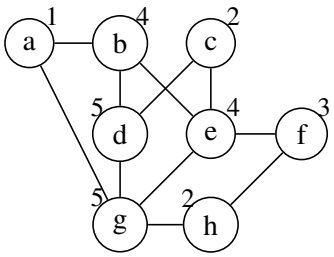
The Decreasing First-Fit heuristic proceeds as follows:

1. start from $x = \emptyset$;
2. put object a into container 1: the residual capacity becomes $40 - 34 = 6$;
3. put object d into container 2: the residual capacity becomes $40 - 30 = 10$);
4. put object f into container 3: the residual capacity becomes $40 - 12 = 28$);
5. put object c into container 2: the residual capacity becomes $10 - 10 = 0$);
6. put object b into container 1: the residual capacity becomes $6 - 4 = 2$);
7. put object e into container 1: the residual capacity becomes $2 - 2 = 0$);

The resulting solution and value of the objective function are the same.

6.6.8 Exercise 8

Given the following undirected graph, solve the *Weighted Vertex Cover Problem* with a constructive algorithm using the objective as the selection criterium. Break ties using the lexicographic order.



Solve it with an adaptive selection criterium that computes for each vertex the ratio of the weight and the number of not yet covered incident edges. Break ties using the lexicographic order.

Solution The basic constructive heuristic proceeds as follows:

1. start from $x = \emptyset$;
2. select $\arg\min_{i \in V \setminus x} w_i = a$, and obtain $x = \{a\}$;
3. select $\arg\min_{i \in V \setminus x} w_i = c$, and obtain $x = \{a, c\}$;
4. select $\arg\min_{i \in V \setminus x} w_i = h$, and obtain $x = \{a, c, h\}$;
5. select $\arg\min_{i \in V \setminus x} w_i = f$, and obtain $x = \{a, c, h, f\}$;
6. select $\arg\min_{i \in V \setminus x} w_i = b$, and obtain $x = \{a, c, h, f, b\}$;
7. select $\arg\min_{i \in V \setminus x} w_i = e$, and obtain $x = \{a, c, h, f, b, e\}$;
8. select $\arg\min_{i \in V \setminus x} w_i = d$, and obtain $x = \{a, c, h, f, b, e, d\}$;
9. stop, because all edges are covered by the vertices in x.

The objective value of the resulting solution $x = \{a, c, h, f, b, e, d\}$ is $f(x) = 1+2+2+3+4+4+5 = 21$. There are many redundant vertices.

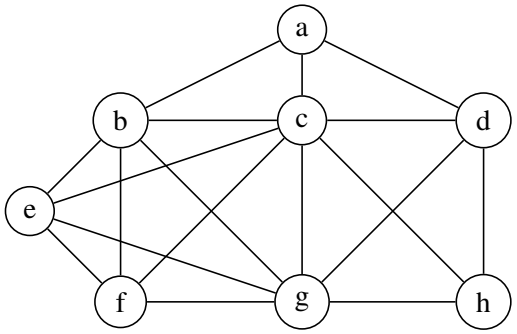
The adaptive constructive heuristic proceeds as follows:

1. start from $x = \emptyset$;
2. select $\arg\min_{i \in V \setminus x} w_i / \delta_i(x) = a$, and obtain $x = \{a\}$;
3. select $\arg\min_{i \in V \setminus x} w_i / \delta_i(x) = c$, and obtain $x = \{a, c\}$;
4. select $\arg\min_{i \in V \setminus x} w_i / \delta_i(x) = h$, and obtain $x = \{a, c, h\}$;
5. select $\arg\min_{i \in V \setminus x} w_i / \delta_i(x) = e$, and obtain $x = \{a, c, h, e\}$;
6. select $\arg\min_{i \in V \setminus x} w_i / \delta_i(x) = d$, and obtain $x = \{a, c, h, e, d\}$;
7. stop, because all edges are covered by the vertices in x.

The objective value of the resulting solution $x = \{a, c, h, e, d\}$ is $f(x) = 1+2+2+4+5 = 14$.

6.6.9 Exercise 9

Given the following instance of the *Maximum Clique Problem* (*MCP*):



apply a constructive heuristic that defines the selection criterium as the degree δ_i of each vertex $i \in V$. This must be maximised, breaking ties with the alphabetic order.

Solution The selection criterium δ_i assumes the following values:

i	a	b	c	d	e	f	g	h
δ_i	3	5	7	4	4	4	6	3

Its aim is clearly to favour vertices with many neighbours, which are more likely to belong to large cliques. Notice that this specific criterium is not particularly smart, because it is not adaptively updated as the current subset x is augmented: the degree counts also neighbours that cannot be added to the current subset. This is done for the sake of simplicity.

At the first step, the set of possible extensions includes all vertices; in the following steps, it includes only the vertices adjacent to all vertices of the current subset x:

$$\Delta^+(x) = \begin{cases} V & \text{when } x = \emptyset \\ \{i \in V : \exists j \in x : (i,j) \in E\} & \text{when } x \neq \emptyset \end{cases}$$

The basic greedy algorithm therefore starts from $x := \emptyset$ and:

1. it selects $c = \arg\max_{i \in \Delta^+(x)} \delta_i$ and updates $x := \{c\}$ and $\Delta^+(x) = \{a,b,d,e,f,g,h\}$;
2. it selects $g = \arg\max_{i \in \Delta^+(x)} \delta_i$ and updates $x := \{c,g\}$ and $\Delta^+(x) = \{b,d,e,f,h\}$;
3. it selects $b = \arg\max_{i \in \Delta^+(x)} \delta_i$ and updates $x := \{b,c,g\}$ and $\Delta^+(x) = \{e,f\}$;
4. it selects $e = \arg\max_{i \in \Delta^+(x)} \delta_i$ (lexicographically) and updates $x := \{b,c,e,g\}$ and $\Delta^+(x) = \{f\}$;
5. it selects $f = \arg\max_{i \in \Delta^+(x)} \delta_i$ and updates $x := \{b,c,e,f,g\}$ and $\Delta^+(x) = \emptyset$;
6. it terminates, because $\Delta^+(x) = \emptyset$.

6.6.10 Exercise 10

Consider the constructive algorithm for the *Capacitated Minimum Spanning Tree Problem* (*CMSTP*) inspired by Kruskal's algorithm for the *MST* problem: the ground set is the edge set E; the search space $\mathcal{F}$ includes all forest (acyclic subsets of edges) such that the trees obtained removing the root have weight $\leq W$; the selection criterium is the objective function. Apply this algorithm to the instance with root in vertex a, weight function $w_v = 1$ for all $v \in V \setminus \{a\}$ and $w_a = 0$, capacity $W = 2$ and cost function:

Cost	a	b	c	d	e	f
a	0	10	5	4	9	6
b	10	0	9	8	11	7
c	5	9	0	12	15	3
d	4	8	12	0	2	13
e	9	11	15	2	0	7
f	6	7	3	13	7	0

Solution The heuristic proceeds as follows:

1. start from $x = \emptyset$;
2. select (d,e);
3. select (c,f);
4. select (a,d);
5. select (a,c);
6. reject (a,f) (it forms a cycle);
7. reject (b,d) (it forms a subtree of excessive weight);
8. reject (a,e) (it forms a cycle);
9. select (a,b);
10. terminate, because the result is a spanning tree and any other edge would form a cycle.

The resulting solution $x = \{(d,e),(c,f),(a,d),(a,c),(a,b)\}$ has objective value $f(x) = 24$.

6.6.11 Exercise 11

Solve the following instance of the *Set Covering problem* (*SCP*):

a	b	c	d	e	f	g	h	i	l
1	3	6	4	3	4	18	19	7	2
1	1	0	0	1	0	0	1	1	0
1	0	1	0	0	0	1	0	0	1
0	0	1	1	0	0	0	1	1	0
0	0	0	0	1	0	1	0	0	0
0	0	1	0	1	1	0	1	0	1
0	0	0	1	0	1	1	0	0	0
0	1	0	1	0	1	0	1	0	0

with the basic constructive heuristic and the adaptive greedy heuristic whose selection criterium is the ratio of the column cost to the number of newly covered rows.

Solution The basic constructive heuristic proceeds as follows:

1. start from $x = \emptyset$;
2. select $\arg\min_{i \in C \setminus x} c_i = a$;
3. select $\arg\min_{i \in C \setminus x} c_i = l$;
4. select $\arg\min_{i \in C \setminus x} c_i = b$ (applying the lexicographic rule);
5. select $\arg\min_{i \in C \setminus x} c_i = e$;
6. select $\arg\min_{i \in C \setminus x} c_i = d$ (applying the lexicographic rule);
7. terminate, because all rows are covered.

The resulting solution $x = \{a, b, d, e, l\}$ has objective value $f(x) = 13$.

The adaptive constructive heuristic proceeds as follows:

1. start from $x = \emptyset$ and compute the ratios $\rho_i(x) = c_i / a_i(x)$:
$$\begin{bmatrix} 0.5 & 1.5 & 2 & 1.\bar{3} & 1 & 1.\bar{3} & 6 & 4.75 & 3.5 & 1 \end{bmatrix}$$
2. select $\arg\min_{i \in C \setminus x} \rho_i(x) = a$ and update the ratios:
$$\begin{bmatrix} - & 3 & 3 & 1.\bar{3} & 1.5 & 1.\bar{3} & 9 & 6.\bar{3} & 7 & 2 \end{bmatrix}$$
3. select $\arg\min_{i \in C \setminus x} \rho_i(x) = d$ (applying the lexicographic rule) and update the ratios:
$$\begin{bmatrix} - & \infty & 6 & - & 1.5 & 4 & 18 & 19 & \infty & 2 \end{bmatrix}$$
4. select $\arg\min_{i \in C \setminus x} \rho_i(x) = e$
5. terminate, because all rows are covered.

The resulting solution $x = \{a, d, e\}$ has objective value $f(x) = 8$.

6.6.12 Exercise 12

Consider the instance of the *Travelling Salesman Problem* on a complete graph with the following cost function:

	a	b	c	d	e
a	0	10	6	13	5
b	8	0	7	5	8
c	11	6	0	7	1
d	4	8	2	0	3
e	13	8	13	11	0

Solve it with the Nearest Neigbour, the Cheapest Insertion, the Nearest Insertion and the Farthest Insertion heuristics, starting from node a. For the sake of simplicity, break all ties with a lexicographic rule.

Solution In the following, it is strongly advisable to draw pictures of the solutions, in order to help visualise the progress of the algorithm and the moves that are available at each step.

The Nearest Neighbour heuristic simply moves from the last visited node to the nearest node not yet visited, until all nodes are visited. Then, it returns to the starting node:

1. start from a;
2. go from a to e;
3. go from e to b;
4. go from b to d;
5. go from d to c;
6. go from c back to a.

The resulting solution $x = \{(a,e),(e,b),(b,d),(d,c),(c,a)\}$ has a cost equal to $f(x) = 5+8+5+2+11 = 31$.

The Cheapest Insertion heuristic starts from a self-loop around node a and proceeds breaking one of the arcs in the current circuit and adding two arcs to reach a nonvisited node. At each step, it requires to maintain for each external node ($i \in N \setminus N_x$) the $|x|$ possible insertions in the current solution (to be efficient, they should form a min-heap, but this is impractical in small exercises solved by hand). Starting from a self-loop around a:

1. the possible insertions (add a node from $N \setminus N_x$ and remove an arc from x) are:

δf	x
$N \setminus N_x$	(a,a)
b	$10+8-0=18$
c	$6+11-0=17$
d	$13+4-0=17$
e	$5+13-0=18$

   Choose c (lexicographically), and obtain $x = \{(a,c),(c,a)\}$.

2. now there are three external nodes and two internal arcs:

δf	x	
$N \setminus N_x$	(a,c)	(c,a)
b	$10+7-6=11$	$6+8-11=3$
d	$13+2-6=9$	$7+4-11=0$
e	$5+13-6=12$	$1+13-11=3$

   Choose d and (c,a), and obtain $x = \{(a,c),(c,d),(d,a)\}$.

3. now there are two external nodes and three internal arcs (notice that column (a,c) is the same as above, whereas the other two correspond to the newly added arcs):

δf	x		
$N \setminus N_x$	(a,c)	(c,d)	(d,a)
b	$10+7-6=11$	$6+5-7=4$	$8+8-4=12$
e	$5+13-6=12$	$1+11-7=5$	$3+13-4=12$

Choose b and (c,d), and obtain $x = \{(a,c),(c,b),(b,d),(d,a)\}$.

4. now there is a single external nodes and four internal arcs (notice that columns (a,c) and (d,a) are the same as above, whereas the other two correspond to the newly added arcs):

δf	x			
$N \setminus N_x$	(a,c)	(c,b)	(b,d)	(d,a)
e	$5+13-6=12$	$1+8-6=3$	$8+11-5=14$	$3+13-4=12$

Choose e (necessarily) and (c,b).

5. stop, because no external node remains.

The resulting solution $x = \{(a,c),(c,e),(e,b),(b,d),(d,a)\}$ has a cost equal to $f(x) = 6+1+8+5+4 = 24$. We can also notice that the sum of the variations chosen during the process is, obviously, the same: $\sum \delta f = 17+0+4+3 = 24$.

The Nearest Insertion heuristic is very similar, but it splits the computation in two independent phases: the first one computes for each external node the distance from the circuit in order to choose one to add; the second computes the variation of the objective function for each internal arc in order to choose one to remove. Starting from a self-loop around a:

1. the distances c_{ai} of the external nodes from the circuit are:

	$N \setminus N_x$			
	b	c	d	e
Distance	10	6	13	5

Choose e and the only existing arc (a,a), with variation $\delta f = 5+13-0 = 18$, to obtain $x = \{(a,e),(e,a)\}$ with cost $\delta c = 5+13 = 18$.

2. now the external nodes are only three, and their distances from the circuit can be updated by simply comparing the previous ones with the distance from the newly added node c_{ei}:

	$N \setminus N_x$		
	b	c	d
Distance	$\min(10, c_{eb}) = 8$	$\min(6, c_{ec}) = 6$	$\min(13, c_{ed}) = 11$

Choose c and consider the two internal arcs that can be removed:

	x	
	(a,e)	(e,a)
δf	$6+1-5=2$	$13+11-13=11$

Therefore, break (a,e), and obtain $x = \{(a,c),(c,e),(e,a)\}$.

3. now the external nodes are only two, and their distances from the circuit can be updated comparing the previous ones with c_{ci}:

	$N \setminus N_x$	
	b	d
Distance	$\min(8, c_{cb}) = 6$	$\min(11, c_{cd}) = 7$

Choose b and consider the three internal arcs that can be removed:

	x		
	(a,c)	(c,e)	(e,a)
δf	$10+7-6=11$	$6+8-1=13$	$8+8-13=3$

Therefore, break (e,a), and obtain $x=\{(a,c),(c,e),(e,b),(b,a)\}$.

4. finally, there is only one remaining external node: $N\setminus N_x=\{d\}$ and four internal arcs can be removed:

	x			
	(a,c)	(c,e)	(e,b)	(b,a)
δf	$13+2-6=9$	$7+3-1=9$	$11+8-8=11$	$5+4-8=1$

Therefore, break (b,a).

5. stop, because no external node remains.

The resulting solution $x=\{(a,c),(c,e),(e,b),(b,d),(d,a)\}$ has a cost equal to $f(x)=6+1+8+5+4=24$, which is, obviously, equal to the sum of the variations obtained during the process: $\sum\delta f=18+2+3+1=24$. The solution is also the same obtained with the *CI* heuristic, but the single steps are different and in general also the result will be different.

Finally, the Farthest Insertion heuristic has the same structure of the Nearest Insertion heuristic, but it replaces the choice of the external node at minimum distance from the current circuit with that of the external node at maximum distance. Starting from a self-loop around a:

1. the distances c_{ai} of the external nodes from the circuit are:

	$N\setminus N_x$			
	b	c	d	e
Distance	10	6	13	5

Choose d and the only existing arc (a,a), with variation $\delta f=13+4-0=17$, to obtain $x=\{(a,d),(d,a)\}$ with cost $\delta c=13+4=17$.

2. now the external nodes are only three, and their distances from the circuit can be updated comparing the previous ones with c_{di}:

	$N\setminus N_x$		
	b	c	e
Distance	$\min(10,c_{db})=8$	$\min(6,c_{dc})=2$	$\min(13,c_{de})=3$

Choose b and consider the two internal arcs that can be removed:

	x	
	(a,d)	(d,a)
δf	$10+5-13=2$	$8+8-4=12$

Therefore, break (a,d), and obtain $x=\{(a,b),(b,d),(d,a)\}$.

3. now the external nodes are only two, and their distances from the circuit can be updated comparing the previous ones with c_{ci}:

	$N \setminus N_x$	
	c	e
Distance	$\min(2, c_{bc}) = 2$	$\min(3, c_{be}) = 3$

Choose e and consider the three internal arcs that can be removed:

	x		
	(a,b)	(b,d)	(d,a)
δf	$5+8-10=3$	$8+11-5=14$	$3+13-4=12$

Therefore, break (a,b), and obtain $x = \{(a,e),(e,b),(b,d),(d,a)\}$.

4. finally, there is only one remaining external node: $N \setminus N_x = \{c\}$ and four internal arcs can be removed:

	x			
	(a,e)	(e,b)	(b,d)	(d,a)
δf	$6+1-5=2$	$13+6-8=11$	$7+7-5=9$	$2+11-4=9$

Therefore, break (a,e).

5. finally, the only remaining external node: $N_x = \{c\}$ can be inserted in the solution in four possible ways

   - break (a,e) and obtain $\delta c = 6+1-5=2$;
   - break (e,b) and obtain $\delta c = 13+6-8=11$;
   - break (b,d) and obtain $\delta c = 7+7-5=9$;
   - break (d,a) and obtain $\delta c = 2+11-4=9$;

   Therefore, break (a,e).

6. stop, because no external node remains.

The resulting solution $x = \{(a,c),(c,e),(e,b),(b,d),(d,a)\}$ has a cost equal to $f(x) = 6+1+8+5+4 = 24$, obviously coinciding with the sum of the variations obtained during the process. The solution is the same obtained with the *CI* and *FI* heuristics, but the single steps performed are completely different, and in general also the final solution will be different.

6.6.13 Exercise 13

Given the instance of the *Vehicle Routing Problem* (*VRP*) in which node a is the depot, the other nodes have weight $w_i = 1$ ($i \in \{b,c,d,e\}$), the vehicles have capacity $W = 2$, and the arcs of the graph have the following costs:

Cost	a	b	c	d	e
a	0	10	6	3	7
b	8	0	7	4	6
c	7	5	0	8	7
d	4	2	10	0	2
e	7	8	5	1	0

apply the *Nearest Neighbour* constructive heuristic, breaking ties with a lexicographic rule.

Solution The heuristic makes the following steps: $a \to d \to b \to a \to c \to e$ with cost $f(x) = 3+2+8+6+7+7 = 33$. Notice that sometimes (for example, when the last node visited is c) the depot node a could be a viable destination for the next step. One must indicate whether this is acceptable or not. The solution above considers it as nonacceptable, as obviously the remaining nodes must anyway be visited, and returning to the depot does not seem reasonable, unless this is forced by the capacity constraint.

6.6.14 Exercise 14

Solve the following instance of the *Maximum Diversity Problem* with $k = 3$ with the basic destructive heuristic:

Distance	a	b	c	d	e	f	g
a	0	14	2	3	18	2	5
b	14	0	18	1	15	10	8
c	2	18	0	19	5	10	9
d	3	1	19	0	1	12	4
e	18	15	5	1	0	8	12
f	2	10	10	12	8	0	3
g	5	8	9	4	12	3	0

breaking ties with a lexicographic rule.

Solution The destructive heuristic starts from the whole ground set $B = P$ and iteratively removes the point that yields the maximum value of the objective function $f(x \setminus \{i\})$. It is easy to prove that, when an element is removed from a solution, the objective function decreases by twice the total distance of that element from the remaining ones. Therefore, instead of recomputing from scratch the objective function, one can simply minimise the total distance of the removed element from the solution (since the distance of the element from itself is zero).

At first, the vector of the total distances is:

$$[\ 44 \quad 66 \quad 63 \quad 40 \quad 59 \quad 45 \quad 41\ ]$$

Therefore, we remove d and update the total distances, subtracting from each point i the value of d_{di}:

$$[\ 41 \quad 65 \quad 44 \quad - \quad 58 \quad 33 \quad 37\ ]$$

Now, we remove f and update again the distances subtracting d_{fi}:

$$[\ 39 \quad 55 \quad 34 \quad - \quad 50 \quad - \quad 34\ ]$$

Now, we remove c and update the distances subtracting d_{ci}:

$$[\ 37 \quad 37 \quad - \quad - \quad 45 \quad - \quad 25\ ]$$

Finally, we remove g and we stop, because we have reached the desired cardinality: solution $x = \{a,b,e\}$ has a value of $f(x) = 14+18+14+15+18+15 = 94$. If one also updates the total distances subtracting d_{gi}, they become:

$$[\ 32 \quad 29 \quad - \quad - \quad 33 \quad - \quad -\ ]$$

and their sum coincides with the objective function ($32+29+33 = 94$), because it is the sum of all pairwise distances.

CHAPTER 7

Constructive metaheuristics

After extending the basic scheme of constructive algorithms, that is modifying the construction graph and the selection criterion, but keeping the basic concept of iteratively enlarging a subset initialised as the empty set until it reaches a final solution, we can now move to **constructive metaheuristics**, that iterate the basic scheme, introducing random steps or memory-based steps that allow to generate different solutions.

7.1 Introduction to constructive metaheuristics

Constructive algorithms, apart from the few provably exact, usually have strong limitations. The idea of the constructive metaheuristics, then, is to generate many different solutions, improving the effectiveness of the algorithm, since at least some of these solutions will be better than that generated by the simple heuristic. Of course, this will also decrease the efficiency, as the computational time will be given by the sum of every execution of the "basic" constructive algorithm. The tradeoff between efficiency and effectiveness must be tuned depending on the available resources.

We will first consider the simplest metaheuristic approach, called *multistart* or *restart*, that consists in running several constructive algorithms, based on different search spaces and different selection criteria. Then, we will introduce the two main mechanisms featured in metaheuristics, that is the use of **randomisation**, as in the case of the semigreedy algorithms, the *GRASP* (*Greedy Randomized Adaptive Search Procedure*) and the Ant Systems and the use of **memory** - that is the exploitation of solutions found in the previous iterations - as in the case of the *ART* (Adaptive Research Technique), the cost perturbation methods and, again, the Ant Systems.

7.1.1 Multistart

Multistart is a classical, very simple and natural approach: it is based on the idea of having several different heuristics and running all of them in sequence, returning the best solution found overall:

- define different search spaces $F_{A^{[l]}}$ and selection criteria $\varphi_{A^{[l]}}(i,x)$
- apply each resulting algorithm $A^{[l]}$ to obtain $x^{[l]}$
- return the best solution $x = \arg\min_{l=1,\ldots,\ell} f(x^{[l]})$ (for minimization problems, max for maximization)

Each algorithm $A^{[l]}$ has its own search space and its selection criterium: they can be completely different or the same algorithm with numerical parameters set to different values.

A technical remark is required if we want to extend the concept of construction graph from the constructive heuristics to the constructive metaheuristics. It is possible to use a single graph by simply including all the nodes and arcs defined by at least one algorithm $A^{[l]}$:

$$F_A = \bigcup_{l=1}^{\ell} F_{A^{[l]}}$$

but defining arc weights that depend on l: $\varphi_A(i,x,l) = \varphi_{A^{[l]}}(i,x)$, with infinite value for the arcs that are forbidden in a specific algorithm $A^{[l]}$: $\varphi_A(i,x,l) = +\infty$ if $i \notin \Delta^+_{A^{[l]}}(x)$. In other words, the different algorithms are modelled as different iterations (with an index l) of a single algorithm, the search spaces of the single algorithms are "merged", considering all the subsets that can be visited by at least one of them, and the weights of the arcs are tuned using the value of the specific algorithm applied in the current iteration, with infinite values for the arcs that are not available.

Example

Consider the three constructive heuristics for the *TSP* whose search space is composed by circuits starting and ending in some node 1: the cheapest insertion, the nearest insertion and the farthest insertion heuristic. Even without introducing new selection criteria φ, it is possible to generate a whole family of constructive heuristics combining them with suitable numerical parameters. The basic structure remains the same: first choose the new node to insert (*insertion* step), then choose the arc to break (*selection* step).

Consider the latter, assuming that in the first step node k has been selected. Instead of using the variation of the objective function, as in all three considered heuristics, we may modify it as

$$i_k^* = \arg\min_{i \in \{1,\cdots,|x|\}} \gamma_{i,k} = \mu_1(c_{s_i,k} + c_{k,s_{i+1}}) - (1-\mu_1)c_{s_i,s_{i+1}}$$

where $\mu_1 \in [0,1]$ tunes the relative strength of the two components of the variation δf: the increase in cost due to the added node k and the decrease in cost due to the removed edge (s_i, s_{i+1}).

Notice that, even if the components are two, we do not need two different parameters, because multiplying them for any positive number yields an equivalent criterium, that has the same optimal solutions. The number of parameters must always be kept as little as possible using normalisations. A typical normalisation is to impose that the coefficients of a linear combination have nonnegative coefficients with a sum equal to 1: this is known as a **convex combination**. Of course, when the parameters are two, setting their sum to 1 means that they can be expressed, respectively, as μ_1 and $1-\mu_1$ and that $0 \leq \mu \leq 1$.

Let us discuss the extreme cases: when $\mu_1 = 0$, only the cost of the removed arc is considered; when $\mu_1 = 1$, only the cost of the two added arcs is considered. Intermediate values *tune* the weights favouring one of the other terms. In particular, $\mu_1 = 1/2$ yields a function $\gamma_{i,k}$ equivalent to the variation of the objective function.

The rationale of this function might be (it is just an example, not a common strategy) that the removed arc measures a saving that is certainly gained, whereas the two added arcs might be removed in any future step. Therefore, the first component of $\gamma_{i,k}$ might be less "informative" than the second. This might even be tuned from step to step.

Considering the *insertion step*, a parametric selection criterion could be:

$$k^* = \arg\min_{k \in N \setminus N_x} \varphi_A(k,x) = \mu_2 d(x,k) - \mu_3 d(x,k) + (1-\mu_2-\mu_3)\gamma_{i_k^*,k}$$

where $\mu_2, \mu_3 \in [0,1]$ tune the relative strength and sign of the distance of the added node k from the current circuit x and the increase in cost due to its insertion. The result is a convex combination, with three nonnegative parameters summing to 1, that is μ_2, μ_3 and $1-\mu_2-\mu_3$. Tuning the weights yields different heuristics: when $\mu_2 = \mu_3 = 0$, one obtains the CI algorithm (of course, if $\mu_1 = 1/2$ in the selection step); when $\mu_2 = 0$ and $\mu_3 = 1$, one obtains the NI algorithm; finally, when $\mu_2 = 1$ and $\mu_3 = 0$, one obtains the FI algorithm. Notice that this specific combination is quite uncommon, because the same function ($d(x,k)$) appears in two terms with opposite signs. In general, this should be avoided, because it confuses the tuning of the parameters (for example, all combinations with $\mu_2 = \mu_3 \in [0,1/2)$ actually yield the CI heuristic and the combination with $\mu_2 = \mu_3 = 1/2$ yields a degenerate zero function, that makes no sense. Once again, it is just an example for a general concept.

7.2 Adaptive research technique

The Adaptive Research Technique (*ART*) was proposed by Patterson et al. in 1998 to solve the *CMSTP*. After that, it was never taken into account in the literature; so, in a sense, it is a failure. Nonetheless, it is interesting from several points of view and, possibly, it was not exploited at its best. The starting point is that a constructive heuristic often obtains bad results due to mistakes committed in the first steps of the algorithm, where elements that are deceivengly good are added but, in the long run, force bad results. We have seen an example for the *KP* in Section 6.3.1, where an object with a very good unitary cost was added to the solution, forbidding the addition of other good elements and ending in a bad result. A way out it to reject the deceivengly good elements: the problem, of course, is to decide which elements are deceptive. The *roll-out* approach tries to answer this question making a look-ahead, that is, trying one by one all the possibilities and comuting the final solution obtained. This approach is clearly very expensive. The approach of the *ART* is to *forbid* an element, hoping that in that way the element chosen in alternative "drive" the flow of the algorithm on the right path of the construction graph. Due to this basic idea, it could be also called a *tabu greedy* method (this name is not standard, but certainly much more expressive). Formally, we can represent the idea as a restriction of the search space in iteration l from F to $F^{[l]}$, based on the solutions obtained by the algorithm in the previous iterations:

$$\min_{i: x \cup \{i\} \in F^{[l]}} \varphi_A(i,x) \text{ with } F^{[l]} = F^{[l]}(x_A^{[1]}, \cdots, x_A^{[l-1]})$$

Of course, this is a memory mechanism, which makes the *ART* a metaheuristic approach. Forbidding elements has also an important effect: the algorithm can be repeated, obtaining different final results. A fundamental point is that the prohibitions introduced must be temporary, because otherwise in subsequent repetitions of the algorithm, they progressively forbid the search space, making it impossible to find optimal or even feasible solutions, and thus terminating the algorithm. Therefore, the *ART* is ruled by a parameter known as the **expiration time** L, that is the number of iterations during which an element remains forbidden after having been selected for prohibition.

Basic scheme of the *ART*

In detail, the *ART* starts by defining a basic constructive heuristic A, so that, in a sense, the *ART* is a meta-algorithm. The tabu mechanism, that handles the prohibitions, employs a vector defined on the ground set: T_i is the iteration at which the algorithm chooses to forbid element $i \in B$. Since at first no element is forbidden, we set $\forall i \in B\ T_i = -\infty$. At each iteration $l \in \{1, \cdots, \ell\}$, where ℓ is the maximum number of iterations imposed by the user, the algorithm follows these steps:

1. apply heuristic A forbidding all elements such that $l \leq T_i + L$; in this way, automatically, all prohibitions older than L expire; let $x^{[l]}$ be the resulting solution
2. if $x^{[l]}$ improves the best solution found until now, save it and save all values $T_i - l$, as they will tell which solutions were forbidden and which solutions were allowed; in this way, it is possible to know how such a solution was obtained, so as to intensify the search around that solution
3. decide, with probability π, whether to forbid or not each $i \in x^{[l]}$ and set $T_i = l$ for each forbidden element
4. (possibly) make auxiliary changes to L, π or T_i

In the end, the best solution found will be returned.

c	25	6	8	24	12
A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	1	1	0
	1	0	0	1	0
	1	0	0	0	1

Table 7.1: An instance of the SCP.

Example: *ART* for the SCP Consider the *SCP* instance in Table 7.1. Its optimal solution consists of the first column, while the basic (adaptive) greedy algorithm would choose columns $\{2,3,5,4\}$, obtaining a bad solution. Let us apply the *ART* metaheuristic, with the following iterations:

1. the basic heuristic with no prohibition finds the solution $x^{[1]} = \{2,3,5,4\}$ of cost $f(x^{[1]}) = 50$; suppose that, at random, column 2 is forbidden;
2. the basic heuristic now finds the solution $x^{[2]} = \{3,1\}$ of cost $f(x^{[2]}) = 33$; suppose that, at random, column 3 is forbidden;
3. the basic heuristic now finds solution $x^{[3]} = \{1\}$ of cost $f(x^{[3]}) = 25$, that is optimal;
4. ...

Of course, this was a lucky case: an unlucky random choice could forbid column 1 at step 2 and delay the identification of the optimal solution, but the prohibition would expire and, sooner or later, a combination of prohibitions allowing to find the optimum could be generated.

Parameters of the *ART*

The *ART* has three basic parameters: the total number of iterations ℓ, the length L of the prohibition and the probability π of the prohibition. The number of iterations is mainly tuned by the available time. The other two are technical parameters that influence the performance of the algorithm in the single iteration. They can be tuned by finding the correct value with experimental campaigns. These tend to be very long, because one must consider all possible values of each parameter. This is a reason why minimising the number of parameters and their range is a good idea. An additional problem is that the experimental evaluation is performed on a specific benchmark, and can lead to **overfitting**, that is labelling as good in an absolute sense values that are good only on the benchmark instances considerated. The excess of parameters is an undesirable aspect, which often reveals an insufficient study of the problem and of the algorithm.

Diversification and intensification

Forbidding elements is a diversification mechanism. **Diversification** is a general concept, which consists in trying to find solutions that are different from those previously found: this drives the search in the solution space from the region in which the algorithm currently operates to different regions. It is also known in textbooks with the alternative name of *exploration*.

A complementary concept is the concept of **intensification**: when a "promising" solution is found, meaning that for some reason we suspect that better and very similar solutions could exist, the search should be driven to look *around* this solution, looking for similar ones. This concept is also known with the alternative name of *exploitation*.

The two concepts are complementary: we must not *choose* one of them, but combine them, with the right tradeoff, that can be different in different steps of the algorithm. In the *ART*, the tuning is ruled by the number of elements forbidden in each iteration and the expiration time. Forbidding many elements means that algorithm must look for very different solutions, thus diversifying; forbidding few elements means that the algorithm must look for something only slightly different, intensifying the research. A long expiration time keeps a large number of forbidden elements used in the past, therefore diversifying the search; a short expiration time allows to go back to solutions similar to the ones visited in the past, thus intensifying.

The values of the parameters, and therefore the balance between diversification and intensification is not necessarily constant throughout the execution. In particular, the parameters that rule it (π and L) could depend on features of the **data**: π_i and L_i could be smaller for good elements i (e.g., large value or small cost). On the other hand, they could also be tuned based on **memory**, that is "learning" parameter values: π_i and L_i could be smaller for elements that belong to the best solutions found so far. Alternatively, they could change from iteration to iteration, and therefore be defined as $\pi^{(l)}$ and $L^{(l)}$, depending on the fact that the best solution found so far is improving (therefore, it could make sense to intensify the search) or the search is stagnating (therefore, it could be better to diversify it). An other way to intensify the search could be to restart the algorithm reinitialising T_i to obtain the same $T_i - l$ values associated with the best solution, so to obtain exactly the same solution and then move in a region close to it.

7.3 The semi-greedy algorithm

The **semigreedy algorithm**, proposed by Hart and Shogan in 1987, tackles the basic problem of the constructive scheme (taking the wrong choice at some step) with randomisation. While the

greedy algorithm chooses at each step the best acceptable extension according to the selection criterion

$$i^* = \arg \min_{i \in \Delta_A^+(x)} \varphi_A(i,x)$$

the semigreedy algorithm assumes that, even if $\varphi_A(i,x)$ is incorrect, it is not completely wrong. In other words, that the elements belonging to an optimal solution are very good for $\varphi_A(i,x)$, even if not strictly optimal. How to know which element to choose?

The semigreedy algorithm defines a suitable distribution function on the elements of $\Delta_A^+(x)$, so as to favour the elements with the best values of $\varphi_A(i,x)$. Then, the new element is selected through a random extraction:

$$\min_{i \in \Delta_A^+(x)} \varphi_A^{[l]}(i,x,\omega)$$

Formally, the idea is to assign a probability $\pi_A(i,x)$ to each arc $(x, x \cup \{i\})$ of the construction graph in such a way that

$$\forall x \in F_A : \left(\Delta_A^+(x) \neq \emptyset \implies \sum_{i \in \Delta_A^+(x)} \pi_A(i,x) = 1 \right)$$

so that the probabilities of the arcs going out of each node sum to 1 and better arcs have larger probabilities:

$$\varphi_A(i,x) \leq \varphi_A(j,x) \iff \pi_A(i,x) \geq \pi_A(j,x)$$

for each $i, j \in \Delta_A^+(x), x \in F_A$ (of course, for minimisation problems).

This heuristic approach can be reapplied several times, in general obtaining different solutions.

7.3.1 Convergence to the optimum

The semigreedy have interesting properties concerning the possibility to reach an optimal solution. In short, it has a nonzero probability to obtain it under the rather obvious technical condition that there exists a path from $\emptyset$ to X^* in the construction graph (accessibility of at least an optimal solutions). Moreover, the probability to obtain the optimum grows gradually with the number of iterations, because each one strictly decreases the probability of *not* reaching the optimum, that is always taking a wrong path.

In fact, given a specific possible run of a semigreedy algorithm, that is a maximal path γ in the construction graph, from the empty set to a terminal subset, the probability of following it is the product of the probabilities on the arcs

$$\prod_{(y, y \cup \{i\}) \in \gamma} \pi_A(i,y)$$

due to the law of conditional probabilities. The probability of reaching a specific node x at the end of the path is the sum of the probabilities on the set Γ_x of all paths leading to x

$$\sum_{\gamma \in \Gamma_x} \prod_{(y, y \cup \{i\}) \in \gamma} \pi_A(i,y)$$

as they are incompatible events (the algorithm cannot follow two paths in Γ_x in a single run). Therefore, the probability of reaching a solution x in the optimal subset is strictly positive if and

only if the construction graph admits a path of nonzero probability from the empty set to the optimal solutions.

Under this condition, the probability not to reach any optimal solution is strictly lower than 1 and strictly decreases, converging to zero as the number of iterations grows up to infinity ($\ell \to +\infty$). The complementary probability of reaching an optimal solution correspondingly converges to 1. In other words, under this condition the semigreedy algorithm is probabilistically approximatively complete. We shall see that more sophisticated algorithms, in which the probabilities are not constant from iteration to iteration, have a more complex behaviour.

Examples of stochastic construction graphs

A first example of an algorithm that uses a stochastic construction graph is the **random walk**, which in the context of heuristics is a constructive metaheuristic where all the arcs going out of the same node have the same probability. This algorithm finds a path to the optimum with probability 1, if such a path exists in the graph. The time required to reach a predefined probability can be extremely long, possibly longer than the time required to solve exactly the problem with the exhaustive algorithm.

A deterministic constructive heuristic sets all probabilities on the graph to zero except for those on the arcs of a single path, namely the arcs that satisfy the selection criterium. Randomised metaheuristics are in general midway between these two extreme cases: the probabilities of the arcs going out of a node are neither perfectly uniform nor concentrated on a single arc. Focusing the probabilities on the most promising arcs and reducing them on the other ones accelerates the average convergence time, weakening the guarantee of convergence to the optimum in the worst case. An algorithm that sets to zero the probability on some arcs can even bar the way to the optimum. An algorithm that modifies the probabilities from iteration to iteration (which the semigreedy algorithm does not do) can reduce the overall probability of finding the optimum from 1 to a strictly lower value, even if a path to the optimum is not strictly removed.

7.4 Greedy Randomized Adaptive Search Procedure

The **Greedy Randomized Adaptive Search Procedure** (*GRASP*), proposed by Feo and Resende in 1989, is a sophisticated variant of the semigreedy heuristic and one of the most popular heuristics in the literature.

It is *greedy* because it uses a greedy basic constructive heuristic; it is *randomised* because it takes random steps; it is *adaptive* because the constructive heuristic typically uses an adaptive selection criterion $\varphi_A(i,x)$ (but this is not strictly necessary). Finally, it is a *search* algorithm because it alternates the constructive heuristic and an exchange heuristic, that allows strongly better results. This aspect will be ignore in this chapter, and investigated in the following ones.

As the semigreedy algorithm, *GRASP* defines a probability function such that better values of the selection criterion correspond to larger probabilities; for minimisation problems:

$$\varphi_A(i,x) \leq \varphi_A(j,x) \iff \pi_A(i,x) \geq \pi_A(j,x)$$

A possible way to define such a function, commonly adopted in genetic algorithms, is to express π directly as a strictly decreasing function of φ. *GRASP* adopts a different approach because (as we shall see in Chapter 10.4.4) experiments on genetic algorithms prove that this method tends to be rather biased and ineffective. Building on the results of those experiments, *ranking schemes* have been proposed and adopted also in *GRASP* and semigreedy algorithms.

The two most popular schemes are the **Heuristic Based Stochastic Sampling** (*HBSS*) and the **Restricted Candidate List** (*RCL*). Both are based on ranking: the elements of the acceptable extension set $\Delta_A^+(x)$ are sorted by nonincreasing values of $\varphi_A(i,x)$ and are assigned probabilities according to their position (or rank) in the ordered sequence. The probability, then, is not given as a function of $\varphi_A(i,x)$, but as a function of the rank, that is an ordinal value. This allows a much better control of the probability values used. The *HBSS* scheme assigns a decreasing probability according to a simple profile (linear, exponential, etc...). The *RCL* scheme assigns a uniform probability to the first k arcs and zero to the remaining ones. See Figure 7.1 for examples of both. Notice that, setting the probability of some arcs to zero, the *RCL* strategy potentially destroys the property of global convergence to the optimum.

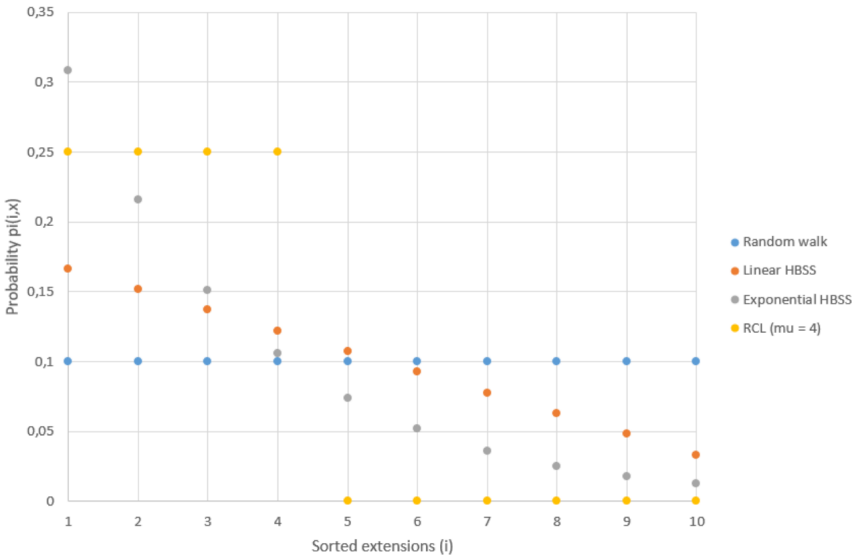


Figure 7.1: Ranking Schemes at a step where $|\Delta_A^+(x)| = 10$ extensions are available

7.4.1 Definition of the RCL

The *RCL* strategy is more common than the *HBSS* strategy. It admits two main variants. The first one directly defines the **cardinality** of the subset of arcs: the *RCL*, in fact, includes the best μ elements of $\Delta_A^+(x)$, where $\mu \in \{1,\cdots,|\Delta_A^+(x)|\}$ is a parameter explicitly fixed by the user. Setting $\mu = 1$ yields the constructive basic heuristic, while setting $\mu = |B|$ yields a random walk. The second strategy is based on **value** of the different extensions. The algorithm first computes the minimum and maximum values of the selection criterium in $\Delta_A^+(x)$

$$\varphi_{min}(x) = \min_{i \in \Delta_A^+(x)} (i,x) \text{ and } \varphi_{max}(x) = \max_{i \in \Delta_A^+(x)} (i,x)$$

Then, it includes in the *RCL* all the elements of $\Delta_A^+(x)$ whose value is between φ_{min} and $(1-\mu)\varphi_{min} + \mu\varphi_{max}$, where parameter $\mu \in [0,1]$ is by the user. Setting $\mu = 0$ yields something similar to the basic constructive heuristic (actually, several equivalent extensions with the minimum value of the selection criterium could be included in the *RCL* and one would be chosen at random with uniform probability). Setting $\mu = 1$ yields a random walk. Clearly, this second definition imposes an adaptive cardinality to the *RCL*, based on the distribution of the values of the selection criterium between the best and the worst choice.

Algorithm 9 provides a pseudocode of the method. In practice, the values of $\varphi_A(i,x)$ are not actually sorted: first one extracts a random numer; then, the position in the ranking of

Algorithm 9 GRASP Pseudocode (TO BE UPDATED)

```
1:  procedure GRASP(I)
2:      x* := ∅
3:      f* := +∞
4:      for l = 1 to ℓ do
5:          while Δ_A^+(x) ≠ ∅ do
6:              φ_i := φ_A(i,x) i ∈ Δ_A^+(x)
7:              L := Sort(Δ_A^+(x), φ)
8:              π := AssignProbabilities(L, μ)
9:              i := RandomExtract(L, π)
10:             x := x ∪ {i}
11:         end while
12:         x := Search(x)
13:         if x ∈ X and f(x) < f* then
14:             x* := x
15:             f* := f(x)
16:         end if
17:     end for
18:     return (x*, f*)
19: end procedure
```

the selected element is derived from the random numerb; finally, the corresponding element is found, without necessarily sorting the set.

GRASP for the SCP Considering the *SCP* instance reported in Table 7.1, let us apply a *GRASP* algorithm using a *RCL* with $\mu = 2$ candidates. A possible execution is:

1. start with the empty subset $x^{(0)} = \emptyset$;
2. compute the rank of the elements in $\Delta^+(x)$. Choose the two best candidates to build a RCL with the μ constratint: column 2, which has $\varphi_2 = 2$ and column 3, which has $\varphi_3 = 4$; select (at random) column 3
3. build a RCL with 2 candidates: column 2 and column 1, which has $\varphi_1 = 6.25$; select (at random) column 1
4. the solution obtained is $x = \{3,1\}$ of cost $f(x) = 33$

Notice that with $\mu = 2$ the optimal solution cannot be obtained (unless a destructive post-processing phase is added), whereas it can be reached with $\mu = 3$.

7.4.2 The reactive tuning of parameters

All randomisation schemes for semigreedy and *GRASP* algorithms are tuned by suitable parameters. For the *HBSS* scheme, this is the slope of the probability profile (be it linear or exponential); for the *RCL* scheme, it is the cardinality or the threshold μ. Instead of fixing the value of μ *a priori*, one can tune it during the execution of the algorithm, learning from the previous results. This is a way to exploit memory, in addition to randomisation.

A classical way to do that is the following **reactive** method, that performs batches of ℓ iterations, divided into phases in which different configurations are applied; after each batch, the length of the single phases are updated:

1. select m configurations of parameters $\mu_1, \cdots, \mu_m$ and set $\ell_r = \ell/m$
2. run each configuration μ_r for ℓ_r iterations
3. evaluate the mean $\bar{f}(\mu_r)$ of the results obtained with μ_r
4. update the number of iterations ℓ_r for each μ_r based on $\bar{f}(\mu_r)$

$$\ell_r = \frac{\frac{1}{\bar{f}(\mu_r)}}{\sum_{s=1}^{m} \frac{1}{\bar{f}(\mu_s)}} \ell \text{ for } r = 1, \cdots, m$$

5. repeat the whole process, going back to point (2), for R times

The idea is to experiment with different values of μ, at first using them for the same number of iterations, and then updating their relative weight, conceding more iterations to the configurations μ_i that yield the best objective values on the average. This scheme is based on the specific values of the objective. Other schemes can be based on scores that count the number of best known results obtained by each configuration.

7.5 Cost perturbation methods and Ant System

Another family of constructive metaheuristics is composed by the **cost perturbation methods**. Their basic idea is, instead of forbidding some of the acceptable extensions or tuning the probability to select each of them, to modify the *appeal* of each choice, that is the value of the selection criterion, based on the information provided by previous iterations. Therefore, it is a memory mechanism.

Given a basic constructive heuristic A, at each step of iteration l the selection criterion $\varphi_A(i,x)$ is tuned by a multiplying factor $\tau_A^{[l]}(i,x)$ before taking choices, so that

$$\psi_A^{[l]}(i,x) = \frac{\varphi_A(i,x)}{\tau_A^{[l]}(i,x)}$$

becomes the actual selection criterium. When a full solution has been built, the multiplying factor $\tau_A^{[l]}(i,x)$ is updated based on the previous solutions $x^{[1]}, \cdots, x^{[l-1]}$. In this way, the elements with better values of $\varphi_A(i,x)$ tend to be favoured, but $\tau_A^{[l]}(i,x)$ tunes this effect. In particular, it can promote **intensification** by increasing for the elements that appear most frequently in good solutions, so as to reduce the selection criterium and generate solutions similar to the previous ones. Alternatively, one can induce **diversification** by decreasing the multiplying factor for the most frequent elements, in order to increase the selection criterium and obtain solutions different from the previous ones. Enforcing intensification or diversification will depend on whether one thinks that the previous solutions were promising or not.

The most famous and important family of cost perturbation methods is the **Ant System**, which admits a number of variants, and combines cost perturbation with randomisation. This method was proposed by Dorigo, Maniezzo and Colorni in 1991, drawing inspiration from the

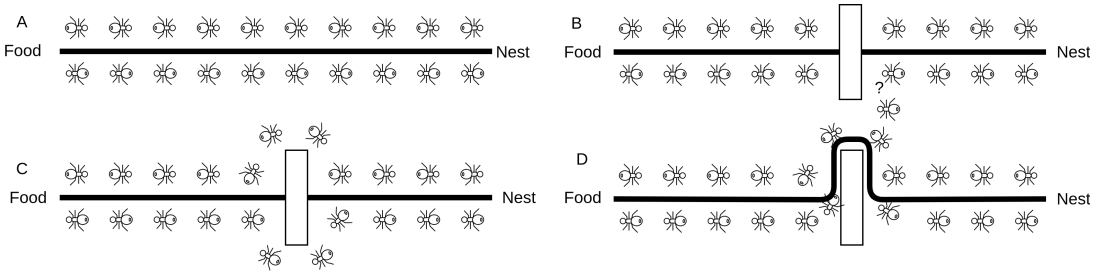


Figure 7.2: Ant behaviour.

social behaviour of ants. The ants go from food to the nest at random and when an ant finds food it goes back to the nest, leaving some **chemical trail** on the ground, so that other ants will be induced to follow it and find the source of food (see Figure 7.2). Since there are different paths to the source of food, less ants will get there by longer paths and the chemical trail on the ground will be weaker, attract less ants and progressively disappear; after a while, only the *best* path will be followed, as more ants follow it. The process of indirect communication among different agents who are influenced by the results of the actions of all agents is called *stigmergy*.

The natural analogy is not important; in fact, it is adviseable to forget it as soon as possible (among many other reasons, because the shortest path problem admits polynomial exact algorithms, and it is not reasonable to solve it in this way), keeping only the basic point that is relevant for optimisation: the use of a trail to mark the data and influence the elementary choices taken to build a solution.

Since the choice is not only influenced by memory, but also randomised, the basic idea of the Ant System can be summarised as follows:

$$\min_{i:x\cup\{i\}\in F} \varphi_A^{[l]}(i,x,\omega,x_A^{[1]},\cdots,x_A^{[l-1]})$$

7.5.1 The role of the trail

In the abstract model, an agent is an application of a basic constructive heuristic, and the trail plays the role of function τ for cost perturbation methods. Each execution of the heuristic leaves a trail on the data, depending on the solution generated: on the other hand, its choices are influenced by the trails left by other executions. The process also has random components.

Differently from the semigreedy heuristic, at each iteration l the basic heuristic A runs f times, instead of one, building a **population** of solutions. At each step of A, all elements of $\Delta_A^+(x)$ can be chosen (there is no *RCL*) and the probability $\pi_A(i,x)$ depends both on the selection criterion $\varphi_A(i,x)$ and the auxiliary information provided by the trail $\tau_A(i,x)$, instead of only on the selection criterium. In the basic version of the Ant System, the trail is produced in previous iterations, in some versions also by other agents during the same iteration.

The trail is uniform at first ($\tau_A(i,x)=\tau_0$) and later tuned increasing or decreasing it, in order to favour promising choices or to avoid repetitive choices, respectively. The trail $\tau_A(i,x)$ is not associated to each arc $(x,x\cup\{i\})$ of the construction graph, because that would require a data structure occupying a huge amount of memory. Typically, the trail values are associated to the elements of the ground set $i\in B$ ($\tau_A(i)$, instead of $\tau_A(i,x)$), implying that all the arcs of the construction graph that correspond to adding the same element i to any possible $x\in\mathcal{F}_A$ have the same trail value. In any case, for the sake of efficiency the trail must be the same for large blocks of arcs with an easy characterisation.

Randomised choice

Instead of selecting the best element according to the selection criterion $\varphi_A(i,x)$, i is extracted from $\Delta_A^+(x)$ with probability

$$\pi_A(i,x) = \frac{\tau_A(i,x)^{\mu_\tau}\eta_A(i,x)^{\mu_\eta}}{\sum_{j\in\Delta_A^+(x)}\tau_A(j,x)^{\mu_\tau}\eta_A(i,x)^{\mu_\eta}}$$

where τ is the trail and η is the **visibility** function

$$\eta_A(i,x) = \begin{cases} \varphi_A(i,x) \text{ for maximisation problems} \\ \frac{1}{\varphi_A(i,x)} \text{ for minimisation problems} \end{cases}$$

that simply assigns larger values to the more promising choices[1]. Differently from the ranking scheme used in *GRASP*, the probability depends on the specific values of the selection criterium. The expression of the probability is normalised by the denominator.

The two parameters μ_τ and μ_η tune the strength of the randomisation and the relative weight of the two terms in the numerator. If $\mu_\eta \approx 0$ and $\mu_\tau \approx 0$ the probability distribution becomes uniform (a random walk). If they are both very large, the probability concentrates on the larger value of $\tau_A(i,x)\eta_A(i,x)$[2]. Setting $\mu_\eta \gg \mu_\tau$ favours the data, simulating the basic constructive heuristic (this makes sense when the known solutions are not very significant, that is not much was learnt from the previous iterations). Setting $\mu_\eta \ll \mu_\tau$ favours memory, keeping close to the previous solutions (this makes sense when the known solutions are very significant).

Ant Colony System

There is a plethora of variants of the Ant System. The **Ant Colony System** distinguishes the tuning of randomisation and the relative tuning between data and memory by splitting each step into two phases. The first phase makes a randomised choice between two alternative: either (with probability q) to select an acceptable extension deterministically, or (with probability $1-q$) to make a randomised choice. Setting $q \approx 1$ favours determinism, while setting $q \approx 0$ favours randomicity. In the second phase, if the deterministic selection prevailed, the algorithm maximises $\tau_A(i,x)\eta_A(i,x)$; if the randomised one prevailed, the algorithm applies a random choice with probabilities

$$\pi_A(i,x) = \frac{\eta_A(i,x)\tau_A(i,x)^{\mu_\tau}}{\sum_{j\in\Delta_A^+(x)}\eta_A(i,x)\tau_A(j,x)^{\mu_\tau}}$$

where parameter μ_{tau} tunes the weight of memory with respect to the data: $\mu_\tau \gg 1$ favours memory , $\mu_\tau \ll 1$ favours the data. This is a form of normalisation, different from the classical convex combination of coefficients, but perfecty equivalent[3].

7.5.2 Trail update

At each iteration l, h instances of the basic heuristic A are run, producing a population $X^{[l]}$ of solutions. A subset $\tilde{X}^{[l]}$ of the obtained solutions is selected to update the trail. In particular,

[1] It should be noted that using values such as $1/\varphi_A(i,x)$ could create numerical problems when $\varphi_A(i,x)$ is large. In particular, different values of $\varphi_A(i,x)$ can yield very similar values of $1/\varphi_A(i,x)$. This is only the classical definition, that could be replaced by more sensible expressions.

[2] Of course, if the product is larger than 1; otherwise, it does the opposite.

[3] Just raise the product to a suitable power and take its logarithm.

their elements are favoured by increasing the associated trail in the following iterations. In order to do this, the trail is updated as follows:

$$\tau_A(i,x) := (1-\rho)\tau_A(i,x) + \rho \sum_{y \in \tilde{X}^{[l]}: i \in y} F_A(y)$$

where $\rho \in [0,1]$ is an **oblivion parameter**, used to reduce the current trail and leave space to the new contributions, whereas $F_A(y)$ is a **fitness function** that measures the quality of solution y. The purpose of the update is to increase the trail on the elements of the solutions $y \in \tilde{X}^{[l]}$ and decrease it on the other elements. Therefore, one must impose that $F_A(y) > \tau$.

7.5.2.0.1 The oblivion parameter When $\rho \approx 1$, the update cancels the current trail and keeps only the new contribution. This is reasonable when the previous solutions are not trustworthy and different ones should be explored, diversifying the search. When $\rho \approx 0$, the update preserves the current trail and completely ignores the new contribution. This is reasonable when the previous solutions are trustworthy and similar ones should be explored, intensifying the search.

7.5.2.0.2 The élite solutions The solutions to be collected in $\tilde{X}^{[l]}$ describe the most promising region of the solution space, since they try and attract the search in the following iterations. The classical Ant System simply used all solutions obtained $\tilde{X}^{[l]} = X^{[l]}$. Recently, elitist methods are preferred, in which only good solutions are taken into account. Often, $\tilde{X}^{[l]}$ is reduced to a single solution, that is the best one found in iteration l, or even the best solution found overall. The elitist methods find better results in shorter time, but they require additional mechanisms to avoid *premature convergence*, that is getting "stuck" around previous solutions.

7.5.2.1 Variants of the trail update mechanism

It should be clear by several notes that the Ant System runs risks of "number-crunching", if the numerical values of the trail are not managed correctly. Several variants have tackled this problem in different ways.

The $MAX-MIN$ Ant System, for example, forces the trail to be confined in a limited range of values $[\tau_{min}, \tau_{max}]$ experimentally tuned. The HyperCube Ant Colony Optimization (HC-ACO) pushes this concept further normalising the values of the trail between 0 and 1.

The Ant Colony System (already mentioned for the separation between the tuning of randomisation and the balance between memory and data) performs two kinds of trail updates:

- the global update - already seen - modifies it at each iteration ℓ, with the purpose to intensify the search on promising solutions;
- the local update modifies the trail at each application g of the basic heuristic, in order to discourage different applications in the same iteration to take identical choices

  $$\tau_A(i,x) := (1-\rho)\tau_A(i,x) \forall i \in x_A^{[l,g]}$$

  The purpose is to diversify the search; in fact, in this case the trail is strictly reduced on the elements that belong to the solution found by the previous "ant" g of the current iteration l.

Algorithm 10 reports a pseudocode of the Ant Colony System, emphasising the position and role of the two update mechanisms.

Algorithm 10 Ant Colony System Pseudocode

```
1: procedure ANTSYSTEM(I)
2:     x* := ∅
3:     f* := +∞
4:     for l = 1 to ℓ do
5:         for g = 1 to h do
6:             x := A(l, τ_A)
7:             x := Search(x)                          ▷ Improvement heuristic
8:             if f(x) < f* then
9:                 x* := x
10:                f* := f(x)
11:            end if
12:            τ_A := LocalUpdate(τ_A, x)
13:        end for
14:        X̃^[l] := Update(X̃^[l], x)
15:        τ_A := GlobalUpdate(τ_A, X̃^[l])
16:    end for
17:    return (x*, f*)
18: end procedure
```

7.5.3 Convergence properties of the Ant System

The Ant System converges to the optimum with probability 1 if suitable assumptions are made on the management of the trail function. The analysis is interesting because it sheds light on the learning mechanism, even if the technical assumptions are actually far from being satisfied in practice.

The analysis is based on the idea that the trail $\tau_A(i,x)$ is a weight function defined on the arcs $(x, x\cup\{i\})$ of the construction graph. It assumes a visibility function uniformly fixed to 1, that is $\eta_A(i,x) = 1$. In other words, the basic heuristic should only decide based on the trail, ignoring the data of the problem. This is a quite strange assumption, but it is useful to guarantee general results, independent from the features of the specific problem, and it can be proved that this restriction can be removed, and the results holds also for basic heuristics considering the data, as well.

Suppose that at the beginning of iteration l, the trail is $\tau^{[l]}$. Then, the basic heuristic(s) are run and $\gamma^{[l]}$ is the best known path on the construction graph from the empty set to a final solution at the end of iteration l. The pair $(\tau^{[l]}, \gamma^{[l-1]})$ can be seen as the state of the computation at the beginning of iteration l. It is also the state of a **nonhomogeneous Markov process**, that is a stochastic mechanism in which the probability of each state depends only on the previous iteration. The process is nonhomogeneous because the dependency on the previous iteration varies with l .

The proof concludes that for $l \to +\infty$, that is after an unlimited number of iterations, at least one run follows an optimum path in F with probability 1 and the trail τ converges to a maximum along one of the optimal paths and to zero on all other arcs of the construction graph.

The main point in the proof is that the trail is being updated with a variable coefficient of oblivion ρ:

$$\tau^{[l]}(i,x) := \begin{cases} (1-\rho^{[l-1]})\tau^{[l-1]}(i,x) + \rho^{[l-1]}\frac{1}{|\gamma^{[l-1]}|} & \text{if } (x, x\cup\{i\}) \in \gamma^{[l-1]} \\ (1-\rho^{[l-1]})\tau^{[l-1]}(i,x) & \text{otherwise} \end{cases}$$

where $|\gamma^{[l-1]}|$ is the number of arcs of path $\gamma^{[l-1]}$, used to normalise the trail. If the oblivion decreases slowly enough

$$\rho^{[l]} \geq 1 - \frac{log(l)}{log(l+1)} \text{ and } \sum_{l=0}^{+\infty} \rho^{[l]} = +\infty$$

then with probability 1 the state converges to (τ^*, γ^*) where γ^* is an optimal path in the construction graph and $\tau^*(i,x) = (|\gamma^*|)^{-1}$ for arcs $(x, x \cup \{i\}) \in \gamma^*$ and 0 otherwise. The conditions on the oblivion are that the sum diverges, so as to forget the trail on the wrong paths quickly enough, but that ρ decreases slowly enough not to get stuck on a path with premature convergence.

A second variant of the Ant System also admits global convergence and maintains a constant oblivion ρ. In this case, however, the trail is slowly forced to decrease to a minimum threshold

$$\tau(i,x) \geq \frac{c_l}{log(l+1)} \text{ and } \lim_{l \to +\infty} c_{[l]} \in (0,1)$$

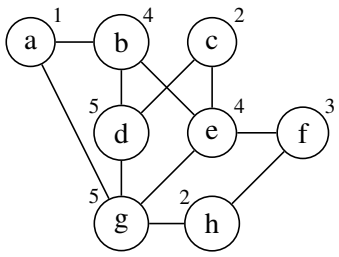
Also in this case, the state converges to (τ^*, γ^*) with probability 1. The intuition is that the trail must *decrease fast but not too fast*.

In practice, all algorithms proposed so far in the literature associate the trail to groups of arcs $(x, x \cup \{i\})$ (e.g. to a single element of the ground set $i \in B$) for efficiency reasons. Moreover, they use constant values for parameters ρ and τ_{min}. Therefore, the convergence guarantee is not proved for practical algorithms. However, one should notice that the aim of a heuristic is not to converge until it finds the optimal solution in each application: it is only to find it once during the execution. Therefore, the result of the theorem is far stronger than needed, and the intuition it gives on the tuning of the process is precious.

7.6 Exercises

7.6.1 Exercise 1

Given the following instance of the *Vertex Cover Problem* (*VCP*)



apply iteration $t = 15$ of an *ART* with tenure $L = 2$, a tabu attribute vector defined on the vertices

t	a	b	c	d	e	f	g	h
T_i	14	10	4	8	5	13	9	11

using the basic constructive heuristic whose selection criterium is the value of the objective ($\phi(i,x) = f(x \cup \{i\})$). Break ties with the alphabetic order.

Update the tabu attribute vector assuming probability $\pi = 0.15$ and the following pseudorandom number sequence: 0.6, 0.8, 0.2, 0.9, 0.6, 0.1...

Solve again the problem at iteration $t = 15$ using the adaptive constructive heuristic whose selection criterium is the ratio of the weight of each vertex i and the number of *newly covered* edges incident in i ($\phi(i,x) = w_i/|\Delta(\{i\}, V \setminus x)|$). Also in this case, break ties with the alphabetic order.

As before, update the tabu attribute vector assuming probability $\pi = 0.15$ and the following pseudorandom number sequence: 0.6, 0.8, 0.2, 0.9, 0.6, 0.1...

Solution The *ART* runs a specified constructive heuristic skipping the forbidden elements of the ground set, that are those such that $t \leq T_i + L$, that is $T_i \geq t - L = 13$. The meaning is trivially that, if $L = 2$, a prohibition imposed at the end of iteration 13 holds during iterations 14 and 15, and then expires. In the present case, therefore, vertices a and f are forbidden; all other vertices are allowed.

The basic heuristic skips a and chooses c and h (in alphabetic order), it skips f, chooses b and e (in alphabetic order), and finally d and g (again, in alphabetic order). The final solution $x = \{c,h,b,e,d,g\}$ has a cost equal to $f(x) = 2+2+4+4+5+5 = 22$.

A trivial remark: forbidding vertices in the *VCP* automatically forces vertices in the solution: if a is forbidden, in fact, in order to cover edges (a,b) and (a,g) necessarily the solution must contain vertices b and g. This could have been applied at the beginning to avoid starting from an empty set. It is also a condition that should be kept into account when applying prohibitions, to avoid generating unfeasible situations.

The solution is also clearly redundant, given that the algorithm completely ignored the fundamental constraint of covering edges. Also from this point of view, using the starting information discussed above would have improved the results.

In the pseudorandom number sequence, only the sixth value falls within the range $[0, \pi]$, and should therefore be forbidden in the following. The text did not specify how we should scan

the elements of the solution to update their tabu attribute. If we scan them in the order in which they have been selected (that is, if we represent the solution as a list or table), g is updated and forbidden:

t	a	b	c	d	e	f	g	h
T_i	14	10	4	8	5	13	**15**	11

If we scan the elements in alphabetic order, considering only those that belong to the solution (that is, if we represent the solution with an incidence vector), h is updated and forbidden:

t	a	b	c	d	e	f	g	h
T_i	14	10	4	8	5	13	9	**15**

The adaptive constructive heuristic starts from $x = \emptyset$, computing and updating at each iteration the number of additional covered edges $\delta_i(x)$ and the ratios $w_i/\delta_i(x)$, still forbidding a and f:

1. since

V	a	b	c	d	e	f	g	h
$\delta_i(x)$	-	3	2	3	4	-	4	2
$w_i/\delta_i(x)$	-	4/3	2/2	5/3	4/4	-	5/4	2/2

   select c (in alphabetic order);

2. since

V	a	b	c	d	e	f	g	h
$\delta_i(x)$	-	3	-	2	3	-	4	2
$w_i/\delta_i(x)$	-	4/3	-	5/2	4/3	-	5/4	2/2

   select h;

3. since

V	a	b	c	d	e	f	g	h
$\delta_i(x)$	-	3	-	2	3	-	3	-
$w_i/\delta_i(x)$	-	4/3	-	5/2	4/3	-	5/3	-

   select b;

4. since

V	a	b	c	d	e	f	g	h
$\delta_i(x)$	-	-	-	1	2	-	3	-
$w_i/\delta_i(x)$	-	-	-	5/1	4/2	-	5/3	-

   select g;

5. since

V	a	b	c	d	e	f	g	h
$\delta_i(x)$	-	-	-	0	1	-	-	-
$w_i/\delta_i(x)$	-	-	-	$+\infty$	4/1	-	-	-

select e;

6. since all edges are covered, stop.

The final solution $x' = \{c,h,b,g,e\}$ has a cost equal to $f(\bar{x}) = 2+2+4+5+4 = 17$. It is better than the previous one, but still redundant.

In the pseudorandom number sequence, none of the first five values falls within the range $[0,\pi]$. Therefore, no element is forbidden and T remains unchanged:

t	a	b	c	d	e	f	g	h
T_i	14	10	4	8	5	13	9	11

This creates the risk to obtain in iteration $t+1 = 16$ the same solution as in iteration $t = 15$. Of course, the expiration of the prohibition for element f could avoid it.

7.6.2 Exercise 2

Consider the instance of the *Travelling Salesman Problem* on a complete graph with the following cost function:

	a	b	c	d	e
a	0	10	6	3	5
b	8	0	7	5	8
c	11	6	0	7	1
d	4	8	2	0	3
e	13	9	13	11	0

and solve it with the *ART* assuming that the current iteration index is $t = 10$, $L = 4$, $\pi = 0.15$, that the constructive heuristic applied is the Nearest Neighbour heuristic starting from node a and breaking ties with the lexicographic rule. Also assume the following tabu attribute matrix:

	a	b	c	d	e
a	$-\infty$	$-\infty$	8	2	1
b	$-\infty$	$-\infty$	$-\infty$	3	$-\infty$
c	2	$-\infty$	$-\infty$	$-\infty$	$-\infty$
d	7	$-\infty$	9	$-\infty$	5
e	$-\infty$	$-\infty$	$-\infty$	6	$-\infty$

and update it in the end assuming the following sequence of pseudorandom numbers: 0.8, 0.1, 0.3, 0.7, 0.2.

Repeat the exercise with the same data, parameters and tabu attribute matrix, except for $T_{ca} = 6$ and $T_{ed} = -\infty$. Then, repeat it again with the original data, parameters and tabu attribute matrix, but setting $T_{bc} = 6$ and $T_{ed} = -\infty$.

Solution Since $t = 10$ and $L = 4$, the forbidden arcs are those with $T_i \geq t - L = 6$, that is (a,c), (d,a) and (d,c) and (e,d).

The *NN* heuristic first selects (a,d). Then, since (d,c) is forbidden, it selects (d,e), followed by (e,b), (b,c) and, finally, (c,a). The final solution is $x = \{(a,d),(d,e),(e,b),(b,c),(c,a)\}$ and its total cost is $f(x) = 3+3+9+7+11 = 33$.

In the pseudorandom number sequence, only the second term is $\leq \pi$. If the solution x is represented as a sequence of arcs (it is unreasonable to use a binary matrix), the second arc, that is (d,e), is now forbidden and the tabu attribute matrix becomes:

	a	b	c	d	e
a	$-\infty$	$-\infty$	8	2	1
b	$-\infty$	$-\infty$	$-\infty$	3	$-\infty$
c	2	$-\infty$	$-\infty$	$-\infty$	$-\infty$
d	7	$-\infty$	9	$-\infty$	**10**
e	$-\infty$	$-\infty$	$-\infty$	6	$-\infty$

Going back to the original tabu attribute matrix with the modified settings $T_{ca} = 6$ and $T_{ed} = -\infty$ meaning that the forbidden arcs are (a,c), (d,a) and (d,c) and (c,a). The *NN* heuristic starts as in the previous example: it first selects (a,d), then skips (d,c) and selects (d,e), followed by (e,b) and (b,c). At this point, the only feasible move is to select (c,a), but this arc is forbidden.

The same happens with $T_{bc} = 6$ and $T_{ed} = -\infty$, because the forbidden arcs are (a,c), (d,a) and (d,c) and (b,c). The *NN* heuristic selects (a,d), skips (d,c) and selects (d,e) and (e,b). At this point, (b,c) is forbidden by the tabu and all other arcs going out of b by the prohibition of subtours.

7.6.3 Exercise 3

Given the following instance of the *Parallel Machine Scheduling Problem* (*PMSP*) with 3 machines:

t	a	b	c	d	e	f
d_t	27	12	15	25	11	16

apply a basic greedy heuristic in which the selection criterium is split into two steps:

1. select the task with the maximum time length: $i^* := \arg\max_{t \in T} d_t$;
2. select the machine that minimises the objective function: $m^* := \arg\min_{m \in M} f(x \cup \{t,m\})$.

In case of ties, in both steps choose the item of minimum index.

Apply a *GRASP* metaheuristic which still deterministically selects the task of maximum time length, but chooses the machine with a *Restricted Candidate List* (*RCL*) of two elements, assuming the following sequence of pseudorandom numbers: 0.2, 0.4, 0.3, 0.8, 0.6, 0.1.

Solution The basic greedy heuristic performs the following operations:

1. start with $x := \emptyset$;
2. select task a, compute the processing time on the three machines for each possible choice (respectively, $(27,0,0)$ for m_1, $(0,27,0)$ for m_2 and $(0,0,27)$ for m_3) and select the one implying the minimum completion time, that is m_1, (based on the minimum index rule for ties);
3. select task d, compute the processing time on the three machines for each possible choice (respectively,$(52,0,0)$ for m_1, $(27,25,0)$ for m_2 and $(27,0,25)$ for m_3) and select the one implying the minimum completion time, that is m_2;
4. select task f, compute the processing time on the three machines for each possible choice (respectively $(43,25,0)$, $(27,41,0)$, $(27,25,16)$) and choose the minimum one, that is m_3;

5. select task c, compute the processing time on the three machines for each possible choice (respectively $(42,25,16)$, $(27,40,16)$, $(27,25,31)$) and choose the minimum one, that is m_3;

6. select task b, compute the processing time on the three machines for each possible choice (respectively $(39,25,31)$, $(27,37,31)$, $(27,25,43)$) and choose the minimum one, that is m_2;

7. select task e, compute the processing time on the three machines for each possible choice (respectively $(38,37,31)$, $(27,48,31)$, $(27,37,43)$) and choose the minimum one, that is m_1;

8. terminate, because there is no possible augmentation.

The final solution assigns tasks a and e to m_1, tasks b and d to m_2 tasks c and f to m_3, with total processing times equal to $(38,37,31)$ and a completion time $f(x) = 38$.

The *GRASP* metaheuristic finds at each step the two best alternatives and selects the first when $r \leq 0.5$, the second otherwise:

1. start with $x := \emptyset$;

2. select task a, compute the processing time on the three machines for each possible choice (respectively, $(27,0,0)$ for m_1, $(0,27,0)$ for m_2 and $(0,0,27)$ for m_3), put m_1 and m_2 in the *RCL* (based on the minimum index rule for ties) and choose m_1 because $r = 0.2$;

3. select task d, compute the processing time on the three machines for each possible choice (respectively,$(52,0,0)$ for m_1, $(27,25,0)$ for m_2 and $(27,0,25)$ for m_3), put m_2 and m_3 in the *RCL* (both with a completion time of 27) and choose m_2 because $r = 0.2$;

4. select task f, compute the processing time on the three machines for each possible choice (respectively $(43,25,0)$, $(27,41,0)$, $(27,25,16)$), put m_3 (competion time 27) and m_2 (completion time 41) in the *RCL* and choose m_3 because $r = 0.3$;

5. select task c, compute the processing time on the three machines for each possible choice (respectively $(42,25,16)$, $(27,40,16)$, $(27,25,31)$) put m_3 (competion time 31) and m_2 (completion time 40) in the *RCL* and choose m_2 because $r = 0.8$;

6. select task b, compute the processing time on the three machines for each possible choice (respectively $(39,40,16)$, $(27,52,16)$, $(27,40,28)$), put m_1 (competion time 40) and m_3 (completion time 40) in the *RCL* and choose m_3 because $r = 0.6$;

7. select task e, compute the processing time on the three machines for each possible choice (respectively $(38,40,28)$, $(27,51,28)$, $(27,40,39)$), put m_1 (competion time 40) and m_3 (completion time 40) in the *RCL* and choose m_1 because $r = 0.1$;

8. terminate, because there is no possible augmentation.

The final solution assigns tasks a and e to m_1, tasks c and d to m_2 tasks b and f to m_3, with total processing times equal to $(38,40,28)$ and a completion time $f(x) = 40$.

7.6.4 Exercise 4

Given the following instance of the *Knapsack Problem* with capacity $V = 11$:

Objects	a	b	c	d	e	f	g
Prize ϕ	10	3	5	12	7	6	8
Volume v	5	1	2	4	3	1	7

solve it with a *GRASP* algorithm that applies the trivial constructive heuristic defining the selection criterium as the objective function. Randomise the choice building a value-based restricted candidate list (*RCL*) with parameter $\mu = 0.4$ and pseudorandom number sequence: 0.4, 0.1, 0.7...

Solution The algorithm starts from the empty set ($x = \emptyset$) and:

1. since $\Delta_A^+(x) = \{i \in E \setminus x : v(x) + v_i \leq V\} = E$ and

Objects	a	b	c	d	e	f	g
ϕ	10	3	5	12	7	6	8

   the best and worst values are $\phi_{\max} = 12$ and $\phi_{\min} = 3$, and the threshold value is $\phi_\mu = \phi_{\max} - \mu(\phi_{\max} - \phi_{\min}) = (1-\mu)\phi_{\max} + \mu\phi_{\min} = 8.4$. Therefore, $RCL = \{a, d\}$ and each of its elements has probability $1/2$. The pseudorandom number 0.4 suggests to select a[4]

2. since $\Delta_A^+(x) = \{b, c, d, e, f\}$ ($a \in x$ and $v_g + v(x) > V$) and

Objects	a	b	c	d	e	f	g
ϕ	-	3	5	12	7	6	-

   the best and worst values and the threshold are unchanged. Therefore, $RCL = \{d\}$ and the choice is deterministic.

3. since $\Delta_A^+(x) = \{b, c, f\}$ and

Objects	a	b	c	d	e	f	g
ϕ	-	3	5	-	-	6	-

   the best and worst values are $\phi_{\max} = 6$ and $\phi_{\min} = 3$, and the threshold value is $\phi_\mu = (1-\mu)\phi_{\max} + \mu\phi_{\min} = 4.8$. Therefore, $RCL = \{c, f\}$ and each of its elements has probability $1/2$. The pseudorandom number 0.1 suggests to select c[5]

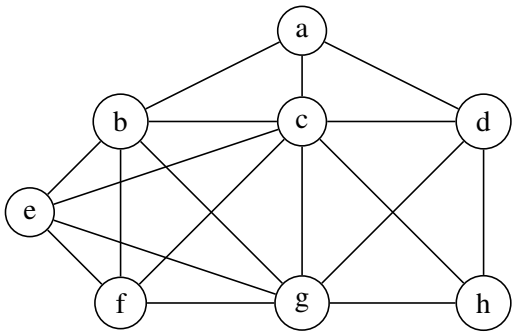
Solution $x = \{a, d, c\}$ has a total value of $f(x) = 27$.

[4] Of course, if the elements are sorted, the answer could be different, but we assume that they are not, for the sake of simplicity.

[5] Once again, the order of the elements is relevant. Moreover, if the previous choice was performed stochastically, even though with probability 1, the pseudorandom number 0.1 would have been "consumed" and we would apply 0.7, selecting f.

7.6.5 Exercise 5

Given the following instance of the *Maximum Clique Problem* (*MCP*):



apply the *Ant System* metaheuristic, assuming that the basic constructive heuristic defines the selection criterium as the degree δ_i of each vertex $i \in V$, to be maximised, breaking ties with the alphabetic order. Suppose that the number of ants is $g = 1$ (for the sake of simplicity), that the visibility function is $\eta_i = \delta_i$, the *trail* function at the current iteration is defined on the vertices of the graph and assumes the following values:

i	a	b	c	d	e	f	g	h
τ_i	1	1	2	1	1	2	1	2

the probability function is

$$\pi(i,x) = \frac{\tau_i \eta_i}{\sum_{j \in \Delta_j^+} \tau_j \eta_j}$$

and the following sequence of pseudorandom numbers is given: $r = 0.4, 0.6, 0.2, 0.8, 0.9, 0.1 \ldots$

Based on the solution obtained, update the trail function combining the current trail and the visibility, with an oblivion coefficient $\rho = 0.1$ and a conversion coefficient $Q = 1$ for the visibility.

Solution The visibility function $\eta_i = \delta_i$ assumes the following values:

i	a	b	c	d	e	f	g	h
η_i	3	5	7	4	4	4	6	3

Its aim is clearly to favour vertices with many neighbours, which are more likely to belong to large cliques. Notice that this specific criterium is not particularly smart, because it is not adaptively updated as the current subset x is augmented: the degree counts also neighbours that cannot be added to the current subset. This is done for the sake of simplicity.

At the first step, the set of possible extensions includes all vertices; in the following steps, it includes only the vertices adjacent to all vertices of the current subset x:

$$\Delta^+(x) = \begin{cases} V & \text{when } x = \emptyset \\ \{i \in V : \exists j \in x : (i,j) \in E\} & \text{when } x \neq \emptyset \end{cases}$$

The *Ant System* metaheuristic starts from $x := \emptyset$ and performs randomised moves based on the following values:

i	a	b	c	d	e	f	g	h
τ_i	1	1	2	1	1	2	1	2
η_i	3	5	7	4	4	4	6	3
$\tau_i\eta_i$	3	5	14	4	4	8	6	6

where the last line can be normalized dividing all values by their sum, that is 50, to obtain the probability function. For the sake of simplicity, however, we deal directly with the nonnormalised values.

The algorithm performs the following steps[6]:

1. $r = 0.4$ with a sum of 50 implies 20; considering the partial sums of $\tau_i\eta_i$ on the elements of $\Delta^+(x)$:

i	a	b	c	d	e	f	g	h
$\tau_i\eta_i$	3	5	14	4	4	8	6	6
Partial sum	3	8	22	26	30	38	44	50

   we select c and update $x := \{c\}$ and $\Delta^+(x) = \{a,b,d,e,f,g,h\}$;

2. $r = 0.6$ with a sum of 36 implies 21.6; considering the partial sums of $\tau_i\eta_i$ on the elements of $\Delta^+(x)$:

i	a	b	d	e	f	g	h
$\tau_i\eta_i$	3	5	4	4	8	6	6
Partial sum	3	8	12	16	24	30	36

   we select f and update $x := \{c,f\}$ and $\Delta^+(x) = \{b,e,g\}$;

3. $r = 0.2$ with a sum of 15 implies 3; considering the partial sums of $\tau_i\eta_i$ on the elements of $\Delta^+(x)$:

i	b	e	g
$\tau_i\eta_i$	5	4	6
Partial sum	5	9	15

   we select b and update $x := \{b,c,f\}$ and $\Delta^+(x) = \{e,g\}$;

4. $r = 0.8$ with a sum of 10 implies 8; considering the partial sums of $\tau_i\eta_i$ on the elements of $\Delta^+(x)$:

i	e	g
$\tau_i\eta_i$	4	6
Partial sum	4	10

   we select g and update $x := \{b,c,f,g\}$ and $\Delta^+(x) = \{e\}$;

[6]This exercise updates the set of possible augmentations and correspondingly the probability function. This is the correct and standard way to operate. In other exercises, for the sake of simplicity, I keep the same probability function throughout the exercise, allowing the extraction of elements that are not acceptable or that have already been selected and suggesting to simply neglect them and go on with the following pseudorandom number. I have still to decide which is the most sensible approach.

5. of course, we select e and update $x := \{b,c,e,f,g\}$ and $\Delta^+(x) = \emptyset$;

6. the algorithm terminates.

The final solution is $x := \{b,c,e,f,g\}$ and its value is $f(x) = |x| = 5$.

Based on solution $x := \{b,c,e,f,g\}$, the trail function can be updated to favour the elements of x and disfavour the other ones, depending also on the quality of x (*intensifying update*). The update formula is:

$$\tau_i = \begin{cases} (1-\rho)\,\tau_i & \text{for } i \notin x \\ (1-\rho)\,\tau_i + \rho Q f(x) & \text{for } i \in x \end{cases}$$

where $Q = 1$ is a conversion factor to combine visibility and trail values, and $\rho = 0.1$ is the oblivion coefficient. In the present case, therefore, the trail on all vertices is multiplied by $1-\rho = 0.9$ and that on the vertices of x is increased by $\rho Q f(x) = 0.1 \cdot 1 \cdot 5 = 0.5$:

i	a	b	c	d	e	f	g	h
$\tau_i^{(\text{old})}$	1	1	2	1	1	2	1	2
$\tau_i^{(\text{new})}$	0.9	1.4	2.3	0.9	1.4	2.3	1.4	1.8

7.6.6 Exercise 6

Solve the following instance of the *Maximum Diversity Problem* with $k = 3$

Cost	a	b	c	d	e
a	0	10	8	7	5
b	10	0	3	7	1
c	8	3	0	5	2
d	4	7	5	0	3
e	5	1	2	3	0

applying the *Ant System* in which the constructive heuristic adopts the objective function as the selection criterium (also at the first step) and breaks ties with the alphabetic order, the number of ants is $g = 1$ (for the sake of simplicity), the visibility function is defined as the objective function, but with a minimum value of 1 to account for the first step ($\eta(i,x) = \min(f(x \cup \{i\}), 1)$), the *trail* function at the current iteration is defined on the points and assumes the following values:

i	a	b	c	d	e
τ_i	1	2	1.5	1	1

the probability function is

$$\pi(i,x) = \frac{\tau_i \eta_i}{\sum\limits_{j \in \Delta_j^+} \tau_j \eta_j}$$

and the following sequence of pseudorandom numbers is given: $r = 0.9, 0.1, 0.3, 0.4\ldots$

Based on the solution obtained, update the trail function combining the current trail and the visibility, with an oblivion coefficient $\rho = 0.2$ and a conversion coefficient $Q = 0.1$ for the visibility.

Solution At the first step, the visibility is equal to 1 for all points. As in the previous exercise, for the sake of simplicity we operate on the values of $\tau_i \eta_i$ without normalising them, exploiting the partial sums to perform the random selections:

i	a	b	c	d	e
η_i	1	1	1	1	1
τ_i	1	2	1.5	1	1
$\tau_i \eta_i$	1	2	1.5	1	1
Partial sum	1	3	4.5	5.5	6.5

Since $r = 0.9$ and the total sum is 6.5, the selected element corresponds to 5.85 and is e.

At the second step, the visibility is the distance from e, $\eta(i,x) = d_{ie}$ (or twice that distance, if one prefers: the choice remains the same). The updated values are:

i	a	b	c	d	e
η_i	5	1	2	3	-
τ_i	1	2	1.5	1	-
$\tau_i \eta_i$	5	2	3	3	-
Partial sum	5	7	10	13	-

Since $r = 0.1$ and the total sum is 13, the selected element corresponds to 1.3 and is a.

At the third step, that is the last one, the visibility is the sum of the distances from e and a, $\eta(i,x) = d_{ie} + d_{ia}$. The updated values are:

i	a	b	c	d	e
η_i	-	11	10	10	-
τ_i	-	2	1.5	1	-
$\tau_i \eta_i$	-	22	15	10	-
Partial sum	-	22	37	47	-

Since $r = 0.3$ and the total sum is 47, the selected element corresponds to 14.1 and is b. The final solution is, therefore, $x = \{a, b, e\}$ and its value is $f(x) = 16$ (or 32, considering the distances in both directions).

Since the problem is of maximisation, the update formula is:

$$\tau_i = \begin{cases} \rho \tau_i & \text{for } i \notin x \\ \rho \tau_i + (1-\rho) Q f(x) & \text{for } i \in x \end{cases}$$

and, therefore, the trail on all points is multiplied by $1 - \rho = 0.8$ and that on a, b and e is increased by $\rho Q f(x) = 0.2 \cdot 0.1 \cdot 16 = 0.32$:

i	a	b	c	d	e
$\tau_i^{(\text{old})}$	1	2	1.5	1	1
$\tau_i^{(\text{new})}$	1.12	2.32	1.2	0.8	1.12

Part IV

Exchange algorithms

CHAPTER 8

Exchange heuristics

As all other algorithms considered in this course, exchange algorithms are based on the concept of *manipulating* subsets of the ground set. An exchange heuristic does it updating a current subset x step by step, exchanging elements in and out of the solutions: some elements taken from $B \setminus x$ are inserted in x, other elements of x are removed.

8.1 The general scheme of exchange algorithms

The general scheme of exchange algorithms is

1. start from a feasible solution $x^{(0)} \in X$ found somehow, often by a constructive heuristic
2. generate a family of feasible solutions by exchanging elements. For example, add a subset A external to $x^{(t)}$ and drop a subset D belonging to $x^{(t)}$:

$$x'_{A,D} = x \cup A \setminus D \text{ with } A \subseteq B \setminus x \text{ and } D \subseteq x$$

3. use a selection criterion $\varphi(x,A,D)$ to choose the subsets to exchange

$$(A^*,D^*) = \arg\min_{(A,D)} \varphi(x,A,D)$$

4. perform the chosen exchange to generate the new current solution

$$x^{(t+1)} := x^{(t)} \cup A^* \setminus D^*$$

5. if a termination condition holds, terminate; otherwise, go back to point (2).

The execution of an exchange heuristic requires a preexisting feasible solution. This is often achieved by applying a constructive heuristic, which ideally takes a small time. This means that the starting point is not given *a priori*, as in the case of constructive heuristics. Different starting solutions usually imply different final ones. Depending on the case, however, the final result could be more or less depending on the initialisation. Discussing constructive heuristics, we have seen that they include extension in which the solution is updated using both added and deleted sets. However, the extension required a strict increase in the cardinality of the solution. Exchange heuristics do not require this. On the contrary, they usually require that both the starting and modified subset are feasible solutions. The graph that results from this mechanism is not acyclic. On the contrary, the arcs are usually symmetric, though this is not strictly required. The algorithm proceeds, iteration after iteration, replacing the current

solution with a new one - called the *incumbent* solution - that is used as the starting point for the next iteration. The process ends when some specific termination condition is satisfied. As for the constructive algorithms, the algorithm is determined by the set of acceptable modifications and by the selection criterion, but additionally also the termination condition must be specified, given that there all solutions have possible successors.

8.1.1 Neighbourhood

An exchange heuristic is defined by the family of pairs of subsets (A,D) that can be exchanged for every solution $x^{(t)}$. Equivalently, it is defined by the collection of solutions generated by these exchanges. The **neighbourhood** is a function that associates to each feasible solution $x \in X$ a subset of feasible solutions $N(x) \subseteq X$:

$$N : X \to 2^X$$

The situation can be formally described with a **search graph**, whose nodes represent the feasible solutions $x \in X$ and whose arcs connect each solution x with the solutions in its neighbourhood $N(x)$. An arc can be seen as a *move*, that represent moving some elements in and/or out of a solution x to obtain a different solution $x' \in N(x)$. So, the triplets (x,A,D) and the arcs of the search graph represent exactly the same thing. The main difference with respect to the construction graph is that the former is acyclic, has a very specific starting node $x^{(0)} = \emptyset$ and terminal nodes, whereas the search graph is cyclic and has no starting and ending node, and all nodes correspond to feasible solutions. In both cases, however, a run of the algorithm corresponds to a path in the graph.

Exactly as several construction graph, several search graphs can be defined on the same problem. The basic difference is that usually all search graphs have the same nodes (the feasible solutions), whereas the construction graph can also have different nodes. Technically, an exchange heuristic *could* visit unfeasible solutions, extending the standard search graph. This implies a set of problems that will be described in more detail when presenting the recombination algorithms in Section 10.5. In the following, we discuss how neighbourhoods are defined.

Neighbourhood based on distance

There are two main approaches to define a neighbourhood. The first one, related to the definition of neighbourhood adopted in continuous mathematics, is based on the concept of **distance**. Let us remind that every subset of a finite set, and therefore every feasible solution $x \in X$, can be represented by its incidence vector

$$x_i = \begin{cases} 1 & i \in X \\ 0 & i \in B \setminus x \end{cases}$$

The distance between two solutions x and x' represented as incidence vectors can be defined in several ways. For example, the class of L_p-distances defines for all $p \geq 0$

$$d_p(x,x') = \sqrt[p]{\sum_{i \in B} |x_i - x'_i|^p}$$

Since the incidence vectors are binary, all such distances are equivalent. In particular, they coincide with the **Euclidean distance** (L_2)

$$d_2(x,x') = \sqrt{\sum_{i \in B} |x_i - x'_i|^2}$$

the **Manhattan distance** (L_1)

$$d_1(x,x') = \sum_{i \in B} |x_i - x'_i|$$

and the **Hamming distance** (L_0)[1]

$$d_H(x,x') = \left|\{i \in B : x_i \neq x'_i\}\right|$$

All such distances trivially count the number of elements that belong to one of the two subsets, and not the other one:

$$d(x,x') = \left|x \setminus x'\right| + \left|x' \setminus x\right|$$

A typical definition of neighbourhood is the set of all solutions with a Hamming distance from x not larger than k for a suitable integer k:

$$N_{H_k}(x) = \{x' \in X : d_H(x,x') \leq k\}$$

This is a parametric definition.

Example: the *KP* The *KP* instance with $B = \{1,2,3,4\}$, $v = [5\ 4\ 3\ 2]$ and $V = 10$ (see Figure 8.1) has 13 feasible solutions out of 16 possible subsets of the ground set: subsets $\{1,2,3,4\}$, $\{1,2,3\}$ and $\{1,2,4\}$, in fact, are unfeasible. Solution $x = \{1,3,4\}$, marked in blue has a a neighbourhood $N_{H_2}(x)$ consisting of 7 elements, marked in red. The subsets in black do not belong to the neighbourhood, because their Hamming distance from x is strictly greater than 2.

(0111)	~~(1111)~~	(0001)	(0000)
(0011)	(1011)	(1010)	(0110)
~~(1110)~~	(1001)	(1000)	(0101)
(1100)	~~(1101)~~	(0010)	(0100)

Figure 8.1: Neighbourhood of a *KP* solution with Hamming distance and $k = 2$.

Neighbourhood based on operations

A second way to define a neighbourhood is based on **operations**. An operation is a function applied to a solution, that generates another solution by adding and/or removing suitable elements. A family of operations $\mathcal{O}$, consequently, defines a family of solutions

$$N_{\mathcal{O}}(x) = \{x' \in X : \exists o \in \mathcal{O} : o(x) = x'\}$$

Consider, for example, the operations that add to x an element of $B \setminus x$, those that remove from x at most an element (including the case of not modifying x) and the operations that exchange one element of x with one of $B \setminus x$. The neighbourhood $N_{\mathcal{O}}$ resulting from the union of all these operations is related to those defined by the Hamming distance, but does not coincide with any of them. More precisely:

$$N_{H_1} \subset N_{\mathcal{O}} \subset N_{H_2}$$

[1]The Hamming distance corresponds to $p = 0$, because all nonzero differences contribute to the sum with a term equal to 1; the zero differences generate 0^0 indeterminate forms, which are conventionally set to 0.

In fact, the subsets of Hamming distance 1 from x have incidence vectors with at most one different value from that of x, therefore at most one element can be added or one can be removed: this neighbourhood does not admit any exchange operation. On the other hand, the subsets of Hamming distance ≤ 2 allow a single swap between an internal and an external element, but also two additions or two removals. The former operations are included in $\mathcal{O}$, whereas the latter are not.

As the distance-based ones, these neighbourhoods can be parametrised considering sequnces of k operations of $\mathcal{O}$ instead of a single one

$$N_{\mathcal{O}_k} = \{x' \in X : \exists o_1, \cdots, o_k \in \mathcal{O} : o_k(o_{k-1}(\cdots o_1(x))) = x'\}$$

where $\mathcal{O}$ should include the identity operation (that does not modify its argument) in order to keep the starting solution in the neigbourhood and to guarantee that each neighbourhood $N_{\mathcal{O}_k}$ includes the ones with smaller values of k.

Different distance and operation-based neighbourhoods

In general, an operation-based neighbourhood includes solutions with different values of the Hamming distance from x. A classical neighbourhood for the *TSP* is the single node swap neighbourhood N_{S_1}, which includes the solutions obtained swapping two nodes in their visit order.

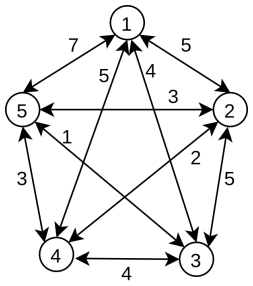


Figure 8.2: TSP Problem

Consider the *TSP* instance in Figure 8.2: the neighbourhood of solution $x = \{(3,1),(1,4),(4,5),(5,2),(2,3)\}$, that corresponds to the node sequence $(3,1,4,5,2)$, is

$$\begin{aligned} N_{S_1}(x) = \{&(\mathbf{1},\mathbf{3},4,5,2),(\mathbf{4},1,\mathbf{3},5,2),(\mathbf{5},1,4,\mathbf{3},2),(\mathbf{2},1,4,5,\mathbf{3}),(3,\mathbf{4},\mathbf{1},5,2),\\ &(3,\mathbf{5},4,\mathbf{1},2),(3,\mathbf{2},4,5,\mathbf{1}),(3,1,\mathbf{5},\mathbf{4},2),(3,1,\mathbf{2},5,\mathbf{4}),(3,1,4,\mathbf{2},\mathbf{5})\} \end{aligned}$$

This neighbourhood cannot be defined using Hamming distances. Swapping two adjacent nodes, in fact, as in the first case, removes three of the five original arcs $((3,1),(1,4),(2,3))$ and adds three new arcs $((1,3),(3,4),(2,1))$, so that the Hamming distance is $3+3=6$. On the other hand, swapping two nonadjacent nodes, as in the second neighbour solution, four arcs are removed $((1,4),(4,5),(2,3),(3,1))$ and, obviously, four arcs have been added, so that the Hamming distance is $4+4=8$. However, N_{S_1} coincides neither with N_{H_6} nor with N_{H_8}: in fact, other exchanges of three or four arcs are possible.

Coinciding distance and operation-based neighbourhoods

In other cases, operation and distance based neighbourhoods are actually equivalent. This is particularly common when the feasible solution of the problem have a fixed number of elements. Consider for example the *MDP*, whose solutions contain k elements: when adding an

element, another element must be removed; all such moves belong to the family of single swaps. Considering the incidence vector, a single swap flips a bit from 1 to 0 and another bit from 0 to 1. The resulting solution belongs to N_{H_2}. Since only feasible solutions in N_{H_2} have two opposite flips, $N_{S_1} \equiv N_{H_2}$, and this can be easily extended to $N_{S_k} \equiv N_{H_{2k}}$ for all k.

Something similar happens for the *BPP*, whose ground set B collects the pairs (object,container), and the *PMSP* with the pairs (task,machine). In both problems the neighbourhood N_{T_1} includes solutions obtained with a single *transfer* (of an object to a container, or a task on a machine). The neighbourhood N_{H_2} includes the solutions at Hamming distance not larger than 2, that are exactly the same, once again because only two opposite flips are feasible and they must concern pairs with the same object. So, $N_{T_k} \equiv N_{H_{2k}}$ for all k.

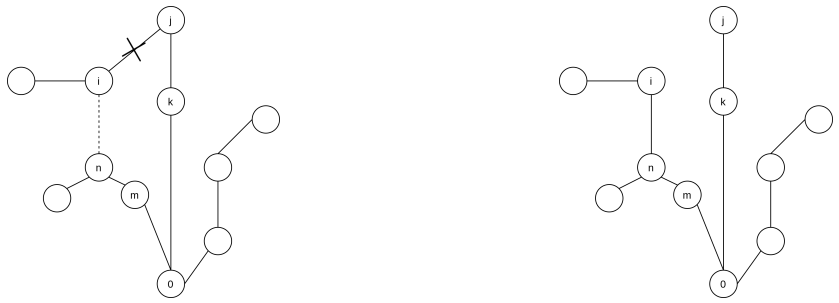
Another example is provided by the Max-SAT problem, where neighbourhood N_{F_k}, defined by at most k *flips* of variables (invertions of their truth assignment) coincides with neighbourhood N_{H_k} for the ground set $B = V \times \mathbb{B}$ introduced in Section 2.3.1.

In summary, operation-based neighbourhood can correspond to distance-based neighbourhood or not, according to the case.

Different neighbourhoods for the same problem

In general, different definitions of neighbourhood can be given for the same problem. Graph problems, in particular, often allow to describe operations in terms of edges (arcs) or in terms of vertices (nodes).

Edge swap neighbourhood for the *CMSTP* Considering the *CMSTP*, a natural neighbourhood can be defined with respect to edges, saying that an edge is deleted and another one is added (in short a single edge swap). Figure 8.3 shows the swap of edge (i, j) with edge (i, n). The consequence is that the whole subtree rooted in node i moves to a different branch (subtree rooted in r) of the solution. This can be interpreted as the transfer or some vertices from branch to branch. In general, a single edge swap can transfer an arbitrary number of vertices: there is no relation between the two number.

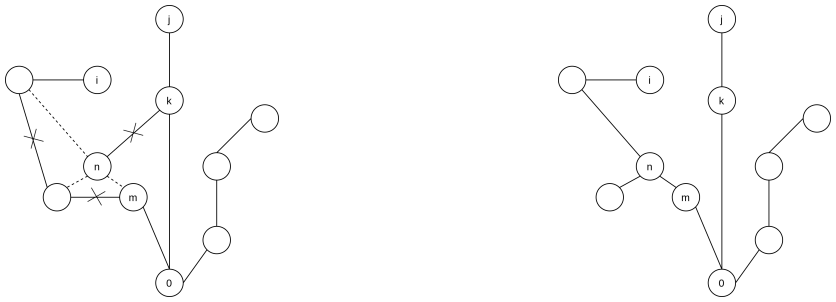


(a) Before exchange (b) After exchange

Figure 8.3: A single edge swap in the *CMSTP*

Vertex transfer neighbourhood for the *CMSTP* Conversely, let us define a neighbourhood in terms of vertex transfers. Figure 8.4 shows the transfer of vertex n from branch 2 to branch 1. We remind that, when the ground set is assumed to consist of assignments of vertices to branches, the objective function must be computed solving a *MSTP* for each branch. Therefore,

after a vertex transfer, the two minimum trees spanning the modified branches must be recomputed. While doing this, in general, it is not known how many of the original edges will be removed and how many new edges will be added, even if a single vertex is transferred.



(a) Before exchange (b) After exchange

Figure 8.4: A single vertex transfer in the *CMSTP*

In summary, two completely heterogeneous ground sets can be considered for the *CMSTP* (one based on edges and one based on assignment of vertices to branches), and two corresponding neighbourhoods (one based on edge swaps and one based on vertex transfers) can be defined. They are both equivalent to distance-based neighbourhoods N_{H_2}, but they are quite different from each other, as the ground set is not the same.

This is very similar to the use of different search spaces in constructive algorithms: in fact, it provides different solutions.

Transfer neighbourhood for the *PMSP* As already observed, the *PMSP* allows to define a single *transfer* neighbourhood N_{T_1}, based on the transfer of a task on a machine (see the example represented in Figure 8.5). This coincides with the Hamming distance neighbourhood N_{H_2}, as in the incidence vector a bit flips from 0 to 1 and another bit flips from 1 to 0, assuming a ground set composed by assignments of tasks to machines.

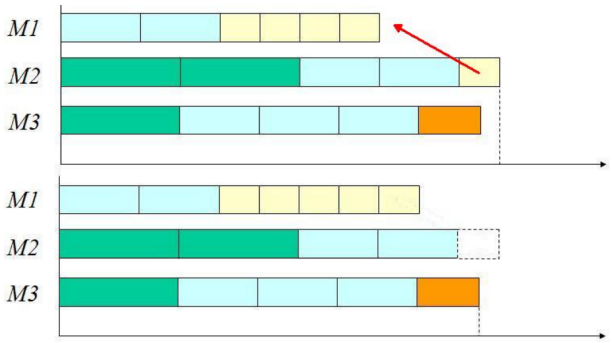


Figure 8.5: A single transfer for the *PMSP*

Swap neighbourhood for the *PMSP* Instead of transferring a single task, one can move two, exchanging their respective positions. Let N_{S_1} be the neighbourhood based on a single *swap* between tasks, as represented in Figure 8.6.

The relationship between task *swaps* and task *transfers* is similar to the relationship between Hamming distance and operations in the case of the *KP* discussed previously: they do not coincide, but a swap is a specific case of two transfers:

$$N_{S_1} \subset N_{T_2}$$

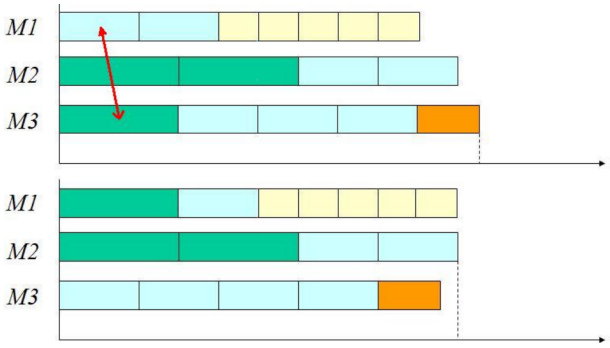


Figure 8.6: A single task swap for the *PMSP*

8.1.2 Connectivity of the search graph

Choosing a neighbourhood is not trivial: the properties of the neighbourhood critically affect the performance of exchange algorithms. A set of important properties are related to the connectivity of the search graph.

Weak connectivity

Exactly as for constructive heuristics with the construction graph, an exchange algorithm is able to reach an optimal solution only if the search graph contains a path from the starting solution to X°. Additionally, exchange heuristic have not a fixed starting solution: their general scheme starts from an arbitrary feasible solution. Therefore, every feasible solution should be able to reach at least one optimal solution, that is there should be a path from x to X^* for every $x \in X$

$$\forall x \in X \; x \rightsquigarrow x^* : x^* \in X^*$$

A search graph having such a property is denoted as *weakly connected* to the optimum. Proving this property for a given neighbourhood is quite difficult, given that the optimal solution set X^* is unknown.

Strong connectivity

Often, a stronger condition is used, requiring that any feasible solution should be reachable from any feasible solution. This is *strong connectivity*, that is the existence of a path in the search graph from x to y for every $x, y \in X$:

$$\forall x, y \in X \; x \rightsquigarrow y$$

As done for constructive metaheuristics, we can associate to each arc of the search graph a probability of being followed, so the possibility to reach an optimal solution with nonzero probability can be expressed in terms of the existence of paths composed by arcs with nonzero probability.

Connectivity has an immediate impact on the quality of a neighbourhood. Consider the single swap neighbourhood N_{S_1} for the *MDP*. It guarantees a way to go from *any* feasible solution to *any* feasible solution, because there is a path made of at most k swaps between them. Finding this path is another matter, far from trivial, but even random walks have a probability to succeed. Considering the single swap neighbourhood for the *KP* or the *SCP*: the corresponding search graphs are not strongly connected. For example, a solution with j elements will never be able to reach a solution with $i \neq j$ elements, even with an arbitrary number of swaps. In

order to make the search graph strongly connected, swaps must be accompanied by removals and additions.

Example on the connectivity of the solution space

Even if a problem has solutions with a fixed number of elements, additional constraints could forbid to move from a solution to any other, destroying strong connectivity. Consider the instance of the *CMSTP* in Figure 8.7: the weights are uniform and equal to 1, except for vertex b which has weight 2.

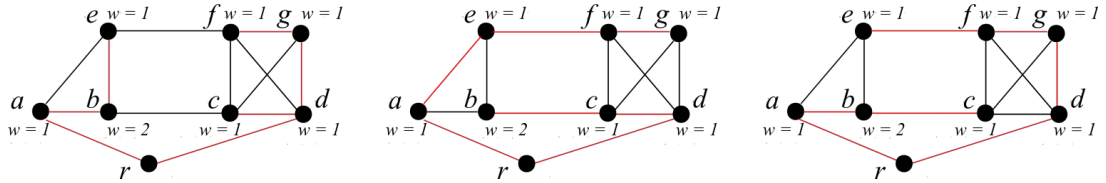


Figure 8.7: Three solutions for a *CMSTP* instance.

Given $V = 4$, only three solutions are feasible, all with two subtrees:

$$x = \{(r,a),(a,b),(b,e),(r,d),(c,d),(d,g),(f,g)\}$$
$$x' = \{(r,a),(a,e),(e,f),(f,g),(r,d),(c,d),(b,c)\}$$
$$x'' = \{(r,a),(a,b),(b,c),(r,d),(d,g),(f,g),(e,f)\}$$

All solutions are $n-1 = 7$ edges. These are mutually reachable only swapping at least two edges (that is, removing two and adding two at a time). The single edge swap neighbourhood does not allow to move from one of these feasible solutions to any other. This is due to the capacity constraint, that forbids to transfer vertices only in one direction (as all single edge swaps do by construction).

The lesson taught by this example is that, even if the general structure of the problem seems to allow connectivity, additional complicating constraints can make some subsets unfeasible and therefore cancel the paths in the search graph that lead from a feasible solution to another one.

8.2 The steepest descent algorithm

8.2.1 The selection criterium

The selection criterion guides the algorithm through the search graph from each solution to a neighbour one, aiming to an optimal solutions, or at least to promising ones. As in constructive algorithms, the most natural choice is to use the objective function as a selection criterion

$$\varphi(x,A,D) = f(x \cup A \setminus D)$$

This is even truer for exchange algorithms because the process visits feasible solutions, for which the objective function is a correct measure of quality, whereas the constructive algorithms visited subsets to which the objective function has to be extended and for which it could easily provide a miopic estimate of quality.

Adopting the objective as a selection criterium, the algorithm moves from $x^{(t)}$ to the best solution in $N(x^{(t)})$. As in constructive algorithms, replacing the objective with the *variation*

$\delta f(x,A,D) = f(x \cup A \setminus D) - f(x)$ usually allows to reduce the computational time. To avoid cyclic behaviours, only strictly improving solutions are accepted at each iteration. The resulting algorithm is called *steepest descent*, because at each step it decreases the objective as much as possible. For maximisation problems, it is called steepest ascent or hill climbing.

Algorithm 11 Exchange Algorithm Pseudocode

```
1:  procedure STEEPESTDESCENT(I)
2:      x := x^(0)
3:      Stop := false
4:      while Stop = false do
5:          x̄ := arg min_{x'∈N(x)} f(x')
6:          if f(x̄) ≥ f(x) then
7:              Stop := true
8:          else
9:              x := x̄
10:         end if
11:     end while
12:     return (x, f(x))
13: end procedure
```

The pseudocode of steepest descent is presented in Algorithm 11. Since the current solution changes only when outperformed by the best neighbour one ($f(\bar{x}) < f(x)$), the last solution obtained is the best and it is directly returned, without managing a data structure *ad hoc*.

Local and global optimality

By definition, the steepest descent algorithm terminates in a *locally optimal* solution, that is a solution $\bar{x} \in X$ not worse than any neighbour one:

$$\forall x \in N(x)\ f(\bar{x}) \leq f(x)$$

A globally optimum solution is always also locally optimal, but clearly the opposite is not true in general:

$$X^* \subseteq \bar{X}_N \subseteq X$$

where X^* is the set of globally optimal solutions and $\bar{X}_N$ the set of locally optimal solutions, that depend on the specific neighbourhood N chosen. Figure 8.8 illustrates the two concepts for a continuous function, where the idea is very similar, even if it is not the case of our analysis.

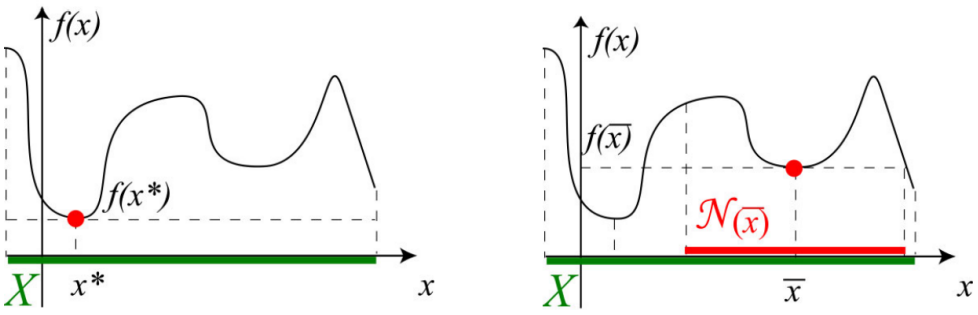


Figure 8.8: Global optimum and local optimum.

8.2.2 Exact neighbourhood

A neighbourhood function $N : X \to 2^X$ such that each local optimum is also a global optimum

$$\bar{X}_N = X^*$$

is called an **exact neighbourhood** (function). The steepest descent algorithm is exact for such neighbourhoods, since it will terminate in $\bar{X}_N$, and consequently in the region of optimal solutions. A trivial case is given by the whole feasible region:

$$\forall x \in X \; N(x) = X$$

This is, however, a useless definition, as it turns the steepest descent algorithm into the exhaustive algorithm: the selection criterion checks every feasible solution.

Sadly, exact neighbourhoods are extremely rare. An important example is the single edge swap neighbourhood for the MSTP: given any spanning tree, one can consider each edge out of the tree, add it closing a cycle, remove an edge of the cycle so that the result is still a spanning tree (thus remaining in the set of feasible solutions) and check whether in this way the objective function value can be reduced. Performing repeatedly the best swap at every iteration (or, actually, any strictly improving swap), it can be proved that the final result is a minimum spanning tree. This is another exact algorithm besides Prim's and Kruskal's. It is less used because its complexity is higher. However, if the starting solution is sufficiently good (for example, because it was optimal before some edge cost was slightly modified), the number of iterations will be sufficiently small to make it an efficient algorithm.

Another example is provided by the single swap between basic and nonbasic variables used by the simplex algorithm for Linear Programming. This can be seen as a combinatorial optimisation problem if one considers it as the problem of identifying an optimal subset of m basic variables (one for each constraint) out of the overall n variables. Of course, the objective function is not additive: it is not a sum of fixed costs associated to the basic variables; it must be computed manipulating the coefficient matrix, the cost vector and the right-hand-side vector. Anyway, the simplex algorithm consists in iteratively swapping a basic variable with a nonbasic one so that the objective function improves: step by step, the algorithm terminates in a local optimum, that is actually a global optimum.

8.3 Properties of the search graph

The effectiveness of steepest descent, and of exchange algorithms in general, depends on the properties of the search graph and of the objective function. Some relevant properties are:

- the size of the search space $|X|$: a larger number of solutions makes the problem harder;
- the connectivity of the search graph: weak connectivity is necessary, more paths to the optimal solutions make it easier to reach them
- the diameter of the search graph, that is the number of arcs of the minimum path between the two farthest solutions: a smaller diameter makes reaching the required solution faster and more likely; a richer neighbourhood produces denser graphs, which usually have a smaller diameter, but other factors may affect this property: the *smallworld* effect allows even sparse graphs to have a small diameter.

Consider, for example, the neighbourhood N_{S_1} (swap of two nodes) for the symmetric *TSP* on complete graphs. The search space consists of $|X| = (n-1)!$ solutions. The neighbourhood of each one includes $N_{S_1}(x) = \binom{n}{2} = \frac{n(n-1)}{2}$ solutions. The search graph is strongly connected and has diameter $n-2$, as the first node of the sequence is always 1 and the last one is automatically in place when the others have been fixed. For example, $x = (1,5,4,2,3)$ can be trasformed into $x' = (1,2,3,4,5)$ in 3 steps:

$$x = (1,5,4,2,3) \rightarrow (1,2,4,5,3) \rightarrow (1,2,3,5,4) \rightarrow (1,2,3,4,5) = x'$$

More complex relevant properties are

- the density of global optima $|X^*|/|X|$ and local optima $|\bar{X}_N|/|X|$: if the local optima are numerous, it is harder to find the global ones;
- the distribution of the quality $\delta(\bar{x})$ of local optima: if the local optima are good enough, it is less important to find a global one;
- the distribution of the locally optimal solutions in the search space: if the local optima are close to each other, it is not necessary to explore the whole space. Once a local optima is reached, intensification allows to reach the other ones. On the contrary, if the local optima are spread, diversification is more appropriate.

Computing these indices requires an exhaustive exploration of the search graph for each single instance. In practice, one performs a sampling of the instances and of the solutions of each instance (which can be a problem in and of itself). These analyses require a very long time and can even be misleading if the global optima are unknown, and therefore part of the required information is only approximately estimated.

Example: the *TSP*

Considering the *TSP* on a complete symmetric graph with Euclidean costs, such analyses typically show that the Hamming distance between two local optima is on average way smaller than n, that is the local optima concentrate in a small region of X. This means that, once a local optimum is found, the search should be intensified. The Hamming distance between local optima on average exceeds that between local and global optima. This suggests that the global optima tend to concentrate in the middle of local optima, another information that supports intensification. The **fitness-distance correlation** diagram of Figure 8.9 reports the percent gap $\delta(\bar{x})$ of several local optima with respect to the global optimum versus their (Hamming) distance from the globally optimal solutions $d_H(\bar{x}, X^*)$.

In the case of the *TSP*, they appear to be correlated: better local optima are closer to the global optima than worse ones. Moving towards better local optima (that is, smaller percentage deviation from the optimum) brings the solution closer to the global optimum. It is still possible to improve the gap (going down) and move away from the global optimum (move right), but it is more unlikely. So, applying an intensification mechanism that moves the search in the surroundings of the best known solutions is probably useful to find even better ones.

Example: the QAP

The Linear Assignment Problem (LAP) is a Combinatorial Optimization problem in which a set of tasks must be assigned to a set of machines so that each task correspond to a single machine and each machine to a single task, with a given cost per each possible pair (task,machine),

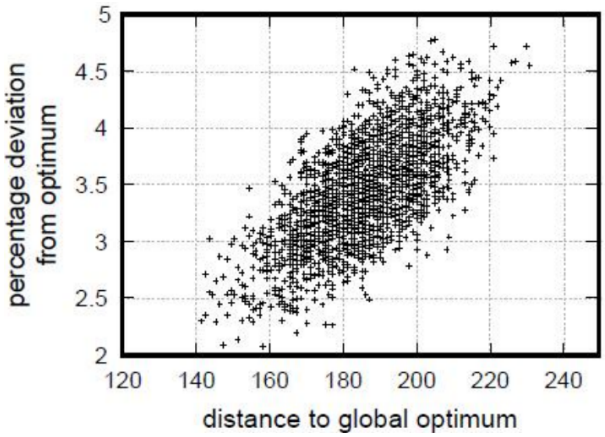


Figure 8.9: Fitness-distance correlation diagram for the *TSP*.

representing the fact that a machine can perform the task better or worse than other ones: the aim is to minimise the sum of all costs for the assignments selected. It is clearly additive.

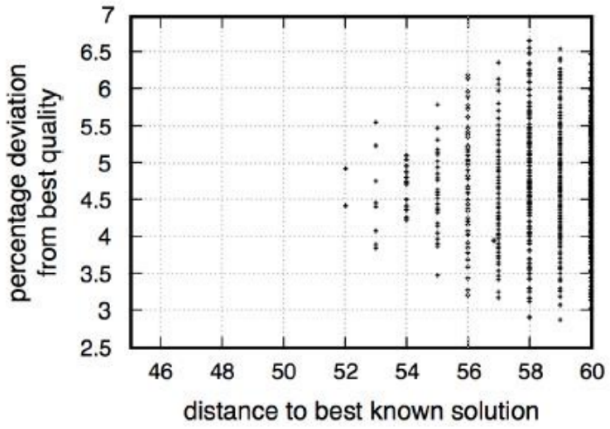


Figure 8.10: Fitness-distance correlation diagram for the QAP.

The Quadratic Assignment Problem (QAP) is a variant in which the total cost of the assignments also includes terms describing the *interferences* between pairs of different assignments. Besides the linear cost of assigning task t_1 to machine m_1 and task t_2 to machine m_2, there is also a cost that is paid when both assignments are selected. The typical FDCD for this problem, represented in Figure 8.10, shows that even if the solution is improved in term of deviation from the best quality (fitness), it does not necessarily get closer to the global optimum: there is no correlation between the two indices. In this case, the search should be diversified, more than intensified.

Such a diagram also gives suggestions about the importance of good starting solutions for an exchange algorithm: if good solutions tend to be closer to the optimum, it makes sense to spend time finding a good initial solution; if, on the contrary, there is no correlation, spending less time on the initialization and more on the exchanges is more reasonable.

The following sections survey a number of descriptive concepts and tools to describe the behaviour of exchange heuristics.

8.3.1 Landscape

The **landscape** is a triplet (X, N, f), where X is the search space (i.e., the set of feasible solutions), $N : X \to 2^X$ is the neighbourhood function and $f : X \to \mathbb{N}$ is the objective function. The first two objects correspond to the search graph (respectively, nodes and arcs), the third to the objective function, that can be seen as a weight function on the nodes. The landscape depends both on the problem (nodes and node weights) and on an exchange algorithm (arcs). The effectiveness of the algorithm strongly depends on its landscape.

Consider the two landscapes in Figure 8.11, where an increase or decrase in height stands for an increment or decrement in cost. We strive to get to the *lowest* nodes in the landscape. If the landscape is "smooth" and has few local optima, than the steepest descent algorithm will usually lead to a very good solution with a number of steps. If the landscape is very rugged, it will quickly lead to a bad local optimum. The first situation is more appropriate for steepest descent than the second.

Figure 8.11: Two different landscapes.

There can be very different type of landscape, such as those represented in Figure 8.12. The one in the upper left corner is an exact neighbourhood. In the upper right corner there is a good global optimum with other very bad local optima: steepest descent will perform poorly. In the lower left corner, the landscape has pretty much similar local optima, distributed everywhere. The lower right corner is a combination of the previous situations, in which the steepest descent will perform very badly, because many local optima are bad, but a mechanism to get out of local optima and reach better ones could drive the search in the correct direction.

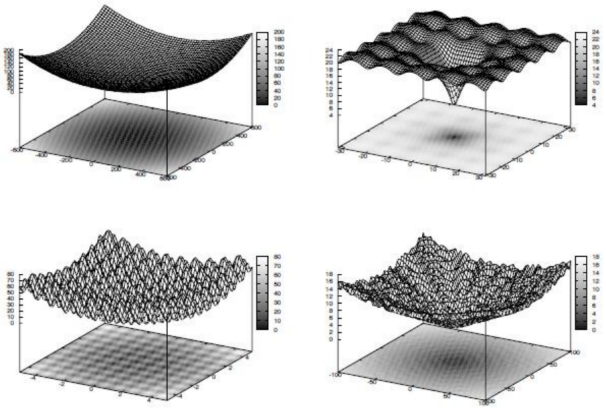


Figure 8.12: Different 3D landscapes.

8.3.2 Autocorrelation coefficient

The aim is to give a quantitative measure able to distinguish the situations depicted in Figure 8.12. The basic idea is to do it empirically, performing a random walk on the search graph for t_{max} iterations, so as to determine a sequence of values of the objective $f^{(1)}, \cdots, f^{(t_{max})}$ and

then combining them in an index of "ruggedness". First, compute the sample mean:

$$\bar{f} = \frac{\sum_{t=1}^{t_{max}} f^{(t)}}{t_{max}}$$

Then, compute the **empirical autocorrelation coefficient**:

$$r(i) = \frac{\frac{\sum_{t=1}^{t_{max}-i}(f^{(t)}-\bar{f})(f^{(t+1)}-\bar{f})}{t_{max}-i}}{\frac{\sum_{t=1}^{t_{max}}(f^{(t)}-\bar{f})^2}{t_{max}}}$$

Let us explain the denominator first. We compute the difference between each value of the sequence and the mean value, raise it to the square and average it over t_{max} iterations. This expression, called the *square mean error* measures how close the sequence of values is to the mean, telling whether the value of the objective function remains pretty much the same or if it varies steeply. The numerator is very similar, but the difference from the mean is computed in two different points and the two differences are multiplied. The aim is to understand not only whether the difference is large or small, but also whether it changes its sign or keeps the same. When $i = 0$, numerator and denominator are the same, so that $r(0) = 1$: each value of the sequence is perfectly correlated with itself. When $i > 0$, the numerator measures whether going from a solution to a solution at i steps of distance the value of the objective changes much or only little, and whether it is above the mean in one solution and below in the other, or it keeps on the same side in both solutions. If the two factors summed at the numerator have the same sign, the autocorrelation coefficient is positive, which suggests that the landscape is smooth. If they are opposite in sign, the autocorrelation coefficient is negative and this means that the landscape is rugged at distance i. In summary:

- if $r(i) \approx 1$ in a large range of distances the landscape is smooth:
  - the neighbour solutions have values close to the current one
  - there are few local optima
  - the steepest descent heuristic is effective
- if $r(i)$ varies steeply the landscape is rugged:
  - the neighbour solutions have values far from the current one
  - there are many local optima
  - the steepst descent heuristic is ineffective

In general, as i increases, the correlation coefficient tends to converge to 0, because solutions far away from each other have values of the objective that are not correlated to each other.

8.3.3 Plateau

A **plateau** is a set of connected solutions in which the objective function has exactly the same value.

Take, for example, the instance of *PSMP* represented in Figure 8.13, moving one of the tasks from machine $M1$ to machine $M3$, the objective function does not change. The same happens transferring any of the two tasks on machine $M3$ to machine $M1$: manydifferent solutions in the single transfer neighbourhood are perfectly equivalent. In this case, the selection criterion

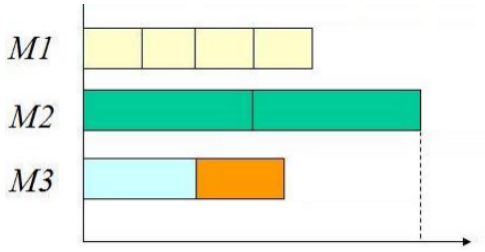


Figure 8.13: Instance of *PMSP* with a plateau.

given by the objective function is nearly ineffective, as it gives no indication on the direction in which to move. This complicates very much the problem. A landscape with large plateaus is particularly smooth, but, contrary to the typical behaviour of smooth landscapes, this is not a desirable property.

8.3.4 Attraction basins

Starting from any feasible solution, the steepest descent leads deterministically to a locally optimal solution. The set of all starting solutions that end up in the same locally optimal solution is known as its **attraction basin**.

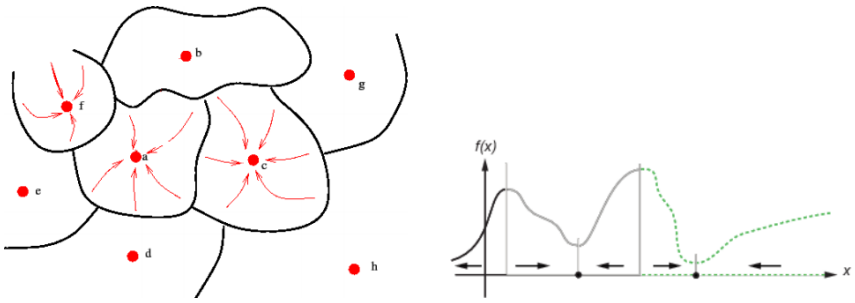


Figure 8.14: Attraction basins.

The attraction basins are separated by arcs, which have on one side a solution "attracted" by a local optimum and on the other side an adjacent solution attracted by a different local optimum. The number of attraction basins in a search space is of interest, as the steepest descent algorithm is effective if the attraction basins are few and large, while it is ineffective if the attraction basins are many and small.

8.4 Efficiency of the exchange algorithms

We now discuss the computational complexity of exchange algorithms and, in particular, the problem of improving as much as possible their efficiency. This is a crucial point as, when metaheuristics will be introduced, the basic scheme of exchange algorithms will be repeated, allowing to also improve the effectiveness of the overall process, and reducing the time required by the basic scheme makes it possible to extend it more strongly.

Exchange algorithms are composed by a simple cycle, exactly as constructive algorithms. The number of iterations of this cycle, however, is not known *a priori* as a function of the size of the problem or some other parameter. This value, t_{max}, depends on the structure of the search graph and of the objective function, and on the details of the exchange algorithm. Considering

the steepest descent algorithm, $t_{\max}$ depends on the starting point $x^{(0)}$, and on the width of the attraction basins. There is an active field of research dedicated to the theoretical estimation of the complexity of exchange algorithms (also known as local search algorithms), but the results are not very strong or general. We therefore ignore this point and focus on the analysis of the single iterations of this cycle, that is on the exploration of the neighbourhood of the current solution. The number of iterations, and consequently the complexity of the overall algorithm will be measured *a posteriori*, in order to determine some empirical average-case information.

8.4.1 The exploration of the neighbourhood

The exploration of the neighbourhood, line (5) in Algorithm 11, has an interesting property: it is a Combinatorial Optimization problem, as it aims to find the best solution in a given finite set. This auxiliary problem is usually solved in two possible ways: either with an *exhaustive search* or with a specialised algorithm. Of course, the second way is better, but only possible for neighbourhoods having very specific structures. The former way is general and, since a neighbourhood is smaller than the whole feasible region, the exhaustive algorithm can actually be a viable approach.

Exhaustive visit of the neighbourhood

For the exhaustive visit of the neighbourhood, the operation

$$\tilde{x} := \arg\min_{x' \in N(x)} f(x')$$

can be expanded as a loop, that generates one by one all the neighbour solutions, computes the corresponding value of the objective function, and updates when necessary the best solution found.

Apparently, the complexity is given by the cardinality of the neighbourhood $|N(x)|$ multiplied by the cost of evaluating each solution, expressed as $\gamma_f(|B|,|x|)$, since in general it depends on the size of the problem and, possibly, of the solution. For example, computing the value of a solution for the *MDP* takes $O\left(k^2\right)$ time, if it is done from scratch. This estimate, however, neglects the fact that it is not always possible to generate only neighbour solutions starting from x: sometimes, the operations that define the neighbourhood generate also unfeasible subsets. For example, given a solution of the *KP*, adding or swapping elements can violate the capacity constraint. In general, therefore, the loop generates all solutions of a superset $N'(x)$ of the actual neighbourhood ($N(x) \subset N'(x)$), made up of candidate subsets, evaluates the feasibility of each subset, spending time time $\gamma_x(|B|,|x|)$ and, finally, evaluates the cost of each feasible solution using time $\gamma_f(|B|,|x|)$.

The resulting complexity for the exploration of the neighbourhood combines three terms: the number of subsets visited ($|N'(x)|$), the time required to evaluate their feasibility (γ_x) and, finally, the time required to evaluate the objective function for each feasible solution (γ_f):

$$T\left(|B|,|x|\right) = |N'(|B|,|x|)|\left(\gamma_x(|B|,|x|) + \gamma_f(|B|,|x|)\right)$$

Since the number of elements N' is an intrinsic feature of the neighbourhood definition, the main effort in reducing the computational cost of the exploration consists in limiting the time required to compute the objective function and to check the feasibility of a subset.

The following sections describe a number of cases in which these computations are performed more efficiently than a full recomputation from scratch.

Updating an additive objective function

It has already been suggested that updating the value of the objective function is more efficient than recomputing it from scratch. Additive objective functions, in particular, offer a simple and powerful example of this concept.

If the objective function is a sum of terms of an auxiliary function defined on the elements of the solutions, the variation associated to adding the elements of a subset A and removing those of a subset D can be expressed as

$$\delta f(x,A,D) = f(x \cup A \setminus D) - f(x) = \sum_{i \in A} \phi_i - \sum_{j \in D} \phi_j$$

Examples of this kind of moves are the swap of objects in the *KP*, of columns in the *SCP*, of edges in the *CMSTP*, etc... Since $f(x)$ is the same for all pairs (A,D), in order to select the best move one can replace the minimisation of $f(x \cup A \setminus D)$ with the minimisation of $\delta f(x,A,D)$. This has two advantages: i) the time required to compute $\delta f(x,A,D)$ is $O(|A|+|D|)$, that is often a small constant, instead of $O(|x|)$, that is typically $O(n)$; ii) the value of $\delta f(x,A,D)$ is independent from x. The first advantage will be discussed now, the second later on.

Example: the symmetric *TSP* Consider the *TSP* with neighbourhood N_{R_2} (the so called 2-optimal exchanges), where two arcs are removed, two different arcs are added and the intermediate sequence is reversed, so as to build again an Hamiltonian circuit (see Figure 8.15).

(a) Before the exchange. (b) After the exchange.

Figure 8.15: Example of an exchange for neighbourhood N_{R_2}.

Notice that this is not a k-swap neighbourhood, as the number of arcs that are removed and added is not a fixed number k: the number of arcs replaced by their opposite ones is variable, and in general it is $O(n)$. Therefore, apparently, the property above discussed does not hold. However, if the cost function is symmetric, the variation of $f(x)$ is

$$\delta f(x,A,D) = c_{s_i,s_j} + c_{s_{i+1},s_{j+1}} - c_{s_i,s_{i+1}} - c_{s_j,s_{j+1}}$$

and consists of exactly four terms: the costs of the two removed arcs, (s_i,s_{i+1}) and (s_j,s_{j+1}), and the costs of the two added arcs (s_i,s_j) and (s_{i+1},s_{j+1}). The costs of the reversed arcs, in fact, are perfectly compensated by those of their opposite ones. Of course, this does not work for the asymmetric *TSP*.

8.4.1.1 Updating a quadratic objective function

Now, let us consider a problem, such as the *MDP*, in which the objective function is quadratic, so that computing its value $f(x)$ costs $O(n^2)$. Let us focus on the single swap neighbourhood N_{S_1},

that contains the subsets obtained moving from x to $x' = x \setminus \{i\} \cup \{j\}$. Updating the objective function requires to compute:

$$\delta f(x,i,j) = f(x \setminus \{i\} \cup \{j\}) - f(x) = \sum_{h,k \in x \setminus \{i\} \cup \{j\}} d_{hk} - \sum_{h,k \in x} d_{hk}$$

The first sum includes the distances between pair of points that belong to x, excluding point i, or between those points and point j. The second sum includes the distances between pair of points in x. Both sums consist of $O\left(n^2\right)$ terms, but most of the terms in the first one appear also in the second one: when computing the difference they will cancel each other out. This allows a general "trick" for simmetric quadratic functions (with $d_{ii} = 0$):

$$\begin{aligned}\delta f(x,i,j) &= \sum_{h \in x \setminus \{i\} \cup \{j\}} \sum_{k \in x \setminus \{i\} \cup \{j\}} d_{hk} - \sum_{h \in x} \sum_{k \in x} d_{hk} \\ &= 2 \sum_{k \in x} d_{jk} - 2 \sum_{k \in x} d_{ik} - 2 d_{ij}\end{aligned}$$

Computing this expression only takes $O(n)$ time.

In fact, the terms that appear only in the first sum are those involving j, that is, all the distances from point j to every point in $x \setminus \{i\}$ and viceversa. since the objective function is simmetric, the opposite distances are the same and can be replaced by the double of one of them. The terms that appear only in the second sum are those involving i, that is, all the distances from point i to every point in x and viceversa. Once again, by symmetry, we can take only one of the two distances and double it. Finally, the first sum can be extended including the distance d_{ij}, provided that an identical term is subtracted. This guarantees that both sums take place for $k \in x$. The final expression involves $n+n+1$ terms, and is linear, instead of quadratic.

Adding and removing $2d_{ij}$ makes little sense, but allows a powerful trick. Let us denote by $D_\ell = \sum_{k \in x} d_{\ell k}$ the total distance of point ℓ from the solution x. The variation of the objective function can be written in the more compact form:

$$\delta f(x,i,j) = 2(D_j(x) - D_i(x) - d_{ij})$$

If D_ℓ is known for each $\ell \in B$, the computation takes $O(1)$ time. In other words, the quadratic time is reduced to a constant time, provided that the information D_ℓ is available.

Example: the *MDP* Now consider the instance of the *MDP* reported in Figure 8.16 (a): a feasible solution consists of the 3 points marked in red, on the left, out of the 6 overall points, while the complement of the complementary points, on the right, are marked in blue. Suppose that we need to evaluate the exchange

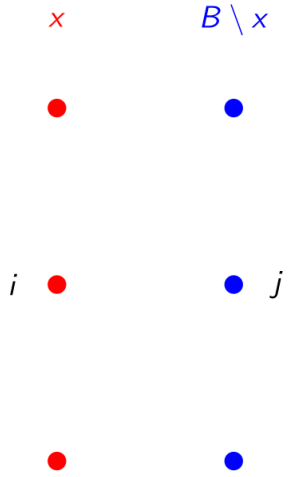
$$x \to x' = x \setminus \{i\} \cup \{j\}$$

with $i \in x$ and $j \in B \setminus x$. As discussed above, the variation of the objective function is obtained subtracting the distances from point i to x (see Figure 8.16 (b)), adding the distances from point j to x (see Figure 8.16 (c)), and subtracting the spurious term d_{ij}, that the second operation generated (see Figure 8.16 (d)).

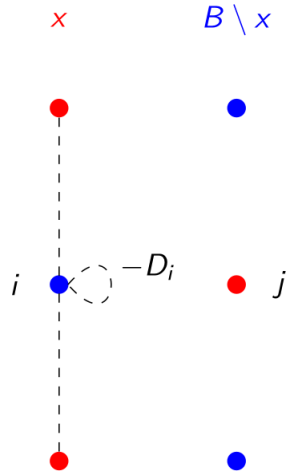
In the end, the updated value is

$$f(x \cup j \setminus i) = f(x) - D_i + D_j - d_{ij}$$

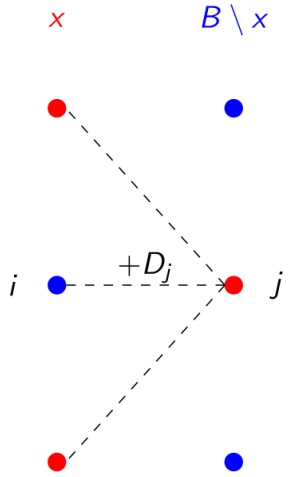
and the cost is computed in $\mathcal{O}(1)$ time for each possible swap, given that D_ℓ is known.



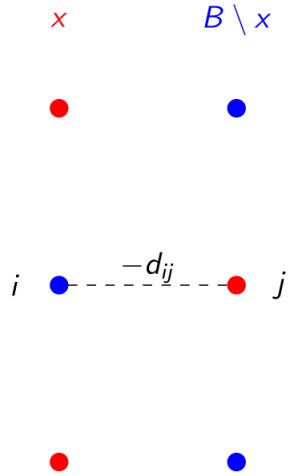
(a) The instance.



(b) Subtract all the distances between i and points in x.



(c) Add all the distances between j and points in x.



(d) Subtract the spurious distance d_{ij}.

Figure 8.16: A swap on the neighbourhood for the *MDP*.

At the end of the neighbourhood exploration, when all the swaps have been evaluated, the current solution is updated, performing one of the swaps considered. This means that the values D_ℓ are no longer valid. However, they do not need to be recomputed from scratch: they can be easily updated, since the solution has changed only slightly, by subtracting the distance of each point ℓ from the removed point i and adding its distance from the added point j, that is

$$D_\ell = D_\ell - d_{\ell i} + d_{\ell j}$$

The auxiliary data structure is updated in $O(n)$ time for each iteration. Since it allows to reduce the computational complexity for each neighbour solution from $O(k^2)$ or $O(k)$ to a constant time, and the number of neighbour solutions is $n(n-k)$, the additional time required at the end is negligible with respect to the huge time saving.

Updating a nonlinear objective

Even some nonlinear objective functions can be updated in constant time, using suitable auxiliary information. The *PMSP* offers an interesting example. The "scheme" is the same as above: save some aggregated information on the current solution $x^{(t)}$, use it to compute $f(x')$ efficiently for each $x' \in N(x^{(t)})$, update it in the end, when $x^{(t)}$ is replaced by $x^{(t+1)}$.

Example: the *PMSP* Consider the transfer or the swap neighbourhood for the *PMSP*. We have already observed that moving to the solutions in such neighbourhoods changes the objective function in a rather convoluted way with respect to the "current" solution, due to its nonlinearity.

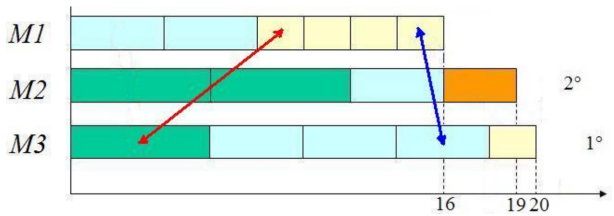


Figure 8.17: Instance of *PMSP*.

Figure 8.17 reports an instance in which moving any task from machine $M1$ to machine $M2$ keeps the objective function unchanged. In fact, the completion time is the total working time of machine $M3$, which remains the maximum. Moving a task from machine $M3$ to another may modify the objective function by amounts different from the length of the task moved: in general, the variation of the objective function is any value up to the time duration of the task moved, both in the positive and in the negative ($[-d_{task}, d_{task}]$). In the case of a swap, things are similar, considering the difference between the durations of the two tasks involved.

Recomputing the overall objective function is expensive: it requires to compute the sum of durations for each machine and to find the maximum one. Therefore, the cost is linear in the number of tasks and machines. The cost can be reduced if one notices that the machines not involved in the transfer or swap do not their working time. Therefore, saving the working time for each machine, one only needs to recompute (in constant time) the working times for the two modified machines and recompute (in linear time with respect to the machines) the maximum total working time.

The following procedure, however, works in constant time for each neighbour solution. The auxiliary information required are the total working time for all machines and the indices of the

two machines with the maximum and the second maximum working time. Now, any transfer or swap modifies the working times of two machines, which can be recomputed in constant time. More precisely, one of them increases and the other one decreases (except for swaps between tasks of the same duration, that leave both numbers unchanged). If none of them is the maximum, the new completion time (overall maximum) is obtained comparing the old one with the working time that has increased. If the maximum is the decreased value, the new completion time can remain the same, be replaced by the second maximum time or, finally, be replaced by the increased value. In any case, the new value of the objective function can be obtained in constant time.

Once the neighbourhood is explored and one new solution is chosen, the auxiliary information must be updated: updating the two modified working times is simple and fast; updating the maximum and second maximum can be done in constant time by keeping all working times in a max-heap. This takes $O(log(|M|))$ time. Notice that, in a max-heap the maximum value is in the first position, whereas the second maximum one is either in the second or in the third position.

Using local auxiliary information

So far, the auxiliary information exploited to update the objective function referred to the starting solution x. In the next example, we consider a case in which it refers to the previous neighbour solution. This can come in handy when the number of elements added or deleted is not constant with respect to x, but it progressively increases.

Consider the asymmetric *TSP* with neighbourhood N_{R_2}: the neighbour solutions differ from x for $O(n)$ arcs, because arcs (s_i, s_{i+1}) and (s_j, s_{j+1}) are removed, arcs (s_i, s_j) and (s_j, s_{i+1}) are added and a whole path is reversed. If the cost function were symmetric, the objective function could be updated in constant time, but since it is asymmetric, this is no longer true! The reversed arcs have a different cost, and part of the solution must be computed again, possibly in $O(n)$ time.

(a) TSP instance before swap. (b) TSP instance after swap.

Figure 8.18: Swap for an asymmetric *TSP* instance.

Can we figure out a way to do the update in constant time? We apply again the idea of keeping additional information, but referring to the previous neighbour solution, instead of the current solution. Suppose that the indices i and j follow the lexicographic order, as in Figure 8.18: the picture on the left represents solution x_2, while the one on the right represents the following neighbour solution, x_3. The first arc removed is the same for both solutions ((π_i, π_{i+1})). The second arc in the latter is the following one with respect to the former ((π_{j+1}, π_{j+2}), instead of (π_j, π_{j+1})). This means that the reversed path in x_3 has just one arc more than the reversed path in x_2, and its old and new cost can be updated, instead of computing it from scratch.

So, the variation of $f(x)$ is

$$\delta f(x,i,j) = c_{s_i,s_j} + c_{s_{i+1},s_{j+1}} - c_{s_i,s_{i+1}} - c_{s_j,s_{j+1}} + c_{s_j \cdots s_{i+1}} - c_{s_{i+1} \cdots s_j}$$

where the fifth term is the cost of the reversed path and the sixth is the cost of the original direct path. The first four terms are specific for each neighbour solution, but can be computed

in constant time. The last two term can be updated in constant time with respect to the previous neighbour solution by saving and updating $c_{s_j \cdots s_{i+1}}$ and $c_{s_{i+1} \cdots s_j}$ with the cost of the additional direct or reversed arc:

$$\begin{cases} c_{s_j,\cdots,s_{i+1}} & = c_{s_j,\cdots,s_{i+1}} + c_{s_{j+1},s_j} \\ c_{s_{i+1},\cdots,s_j} & = c_{s_{i+1},\cdots,s_j} + c_{s_j,s_{j+1}} \end{cases}$$

So, when x_j is visited, its objective function value is computed using information related to x_{j-1} and obtained from the visit of x_{j-1}. The chain of updates goes back to the first neighbour solution x_1, which draws information directly from x. In fact, at the first step the reversed path is just one arc and its cost can be easily computed from the data in constant time.

This implies that the neighbourhood of solution x is visited in a given order, designed so as to guarantee the possibility of the update in constant time. Is it acceptable to explore the neighbourhood in a predefined order? The answer depends on the specific algorithm considered. If the neighbour contains many equivalent solutions, in general the first one is accepted, and using a specific exploration order adds a bias on the exploration. Moreover, methods that do not explore the whole neighbourhood cannot usually adopt this trick, because skipping part of the neighbourhood could break the chain updates, that must be on the contrary be uninterrupted.

8.4.2 Updating the feasibility check

Not all definitions of neighbourhood allow to generate only feasible subsets: swaps are certainly feasible in the *MDP*, while other problems (for example, the *MSTP*) allow to restrict the exploration to feasible swaps (add an edge and remove only the edges on the unique paths between the extreme vertices of the added edge and the root of the tree). In general, however, the exploration process generates also unfeasible subsets:

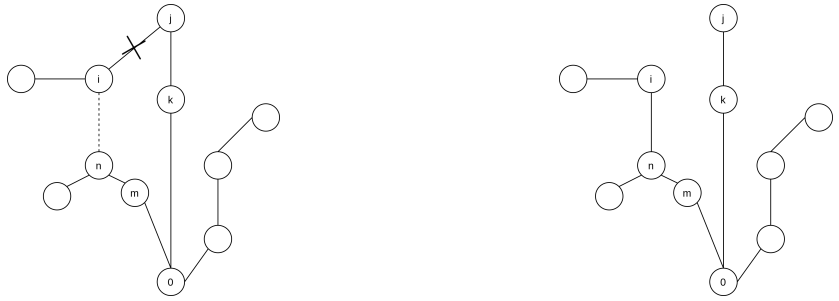
$$\tilde{N}_{\mathcal{O}}(x) = \{x' \subseteq B : \exists o \in \mathcal{O} : o(x) = x'\} \supseteq N_{\mathcal{O}}(x) = \tilde{N}_{\mathcal{O}}(x) \cap X$$

In this situation, before computing the objective function one must first check the feasibility of the generated subsets. This usually requires to estimation one or more auxiliary functions (for example, the total volume of solution x in the *KP*, the number columns covering each row in the *SCP*, and so on). Of course, it is more efficient to update these functions instead of recomputing them from scratch. The techniques that can be used are exactly the same used to update the objective function. For example, the total volume of a subset in the *KP* can be updated by adding the volume of the added elements and subtracting the volume of the removed ones, which takes constant time for all exchanges in which the number of added and removed elements is constant.

Example: the *CMSTP*

A more sophisticated example concerns the *CMSTP* with the single edge swap neighbourhood N_{S_1}. A single edge swap transfers a whole subtree from a branch to another one. This implies a variation in the total weight of the two branches, one of which increases, and could possibly exceed the capacity. While the single vertex transfer neighbourhood N_{T_1} allows a constant time update of the total volumes, because a single vertex is transferred, the edge swap can move $O(n)$ vertices, and recomputing the volume apparently takes more time. In fact, the total volume of a branch can be recomputed visiting the branch, in linear time with respect to the number of vertices in the branch.

Suppose, however, that the total volume of the subtree appended to each vertex of the graph in the current solution is known. The solution can be represented with a vector that provides for



(a) Before exchange (b) After exchange

Figure 8.19: Edge exchange in CMST

each vertex the "father" vertex, that is the first one on the unique path to the root. Cutting the edge between a vertex and its father disconnects a subtree, whose total weight can be saved in another vector, in the position corresponding to the index of the vertex considered. Assuming to have these three vectors, consider the instance in Figure 8.19 (a): adding edge (i,n) closes a cycle, whose edges can be determined moving from i to the root and from n to the root through the vertices indicated by the father vector. This also provides the two vertices immediately appended to the root on the two paths. Each edge thus obtained could be removed in order to obtain a new spanning tree. For example, adding edge (i,j) to replace edge (i,n) would modify the objective function by $\delta f = c_{in} - c_{ij}$, which can be computed in constant time. The swap also moves a subtree from a branch to another branch of the original solution. The total volume of the moved subtree is given by the auxiliary weight vector in position i, whereas the residual capacity of the receiving branch is given by the same vector in the position corresponding to the vertex appended to the root. Comparing the former with the latter determines whether the swap is feasible or not. This takes constant time for each edge. The branch from which the subtree is removed, in fact, is obviously feasible.

Of course, in the end, when the best swap is performed, all the auxiliary information must be updated, but only some vertices of the two involved branches need modifications, and this can be done in $O(n)$ time (exploiting the father vector, or with a visit).

A general scheme for efficient neighbourhood exploration

In summary, the efficient exploration of a neighbourhood (possibly in constant time for each solution) exploits additional data structures that provide suitable information in aggregated form. These structures must be initialised and updated. They can be

- global, when they save information about the current solution, and are therefore updated at the end of the exploration, when the incumbent solution replaces it;
- local, when they save information about the previous neighbour solution, and are therefore updated when a new neighbourh solution is considered.

The former is the case of the vector of total distances from the solution in the *MDP*, the total processing time in each machine for the *PMSP*, the total volume appended to each vertex in the *CMSTP*. The latter is the case of the total cost of the reversed subpath in the asymmetric *TSP*. The update of the latter is much more frequent, and therefore must be much faster (typically, in constant time). The overall scheme is summarised in Algorithm 12.

Algorithm 12 Sophisticated Exchange Heuristic Pseudocode

```
1:  procedure STEEPESTDESCENT(I)
2:      x := x^(0)
3:      GD := InitializeGD()
4:      Stop := false
5:      while Stop = false do
6:          x̃ := 0
7:          δ̃ := 0
8:          LD := InitializeLD()
9:          for ∀x' ∈ N(x) do
10:             if f(x') < f(x̃) then
11:                 x := x̃
12:             end if
13:             LD := UpdateLD(LD, x')
14:         end for
15:         if f(x̃) ≥ f(x) then
16:             Stop := true
17:         else
18:             x := x̃
19:             GD := UpdateGD(GD, x̃)
20:         end if
21:     end while
22:     return (x, f(x))
23: end procedure
```

8.4.3 Partial saving of the neighbourhood

Instead of accelerating the evaluation of the single neighbour solutions, several problems allow to avoid evaluating part of the neighbourhood. This nice sophisticated technique does not work for all problems. It is based on the idea that some problems have solutions composed of independent parts that have an implication on the effects of the moves. For example, the solutions of the *BPP* partition a set of objects into containers, those of the *VRP* divided customers into vehicles, the *CMSTP* divides vertices into branches, etc...

In formal terms, when performing an operation $o \in \mathcal{O}$ (that is a pair of subsets A and D, we remind) on a solution $x \in X$, the variation of the objective

$$\delta f(x,o) = f(o(x)) - f(x)$$

and the feasibility of $o(x)$ depends only on a part of x. In this case, there might exists a large subset $\tilde{\mathcal{O}}(o^*) \subset \mathcal{O}$ such that

$$\forall o \in \tilde{\mathcal{O}} : \delta f(o^*(x), o) = \delta f(x,o)$$

The variation in the objective function due to the application of an operation $o(x)$, does not always depend on the whole solution x, so it remains the same for $x' = o^{ast}(x)$, making it advantageous to

1. compute $\delta f(x,o)$ for each $o \in \mathcal{O}$ and save the values
2. perform the best operation o^*, generating the new solution x'

3. retrieve $\delta f(x', o)$ for each $o \in \tilde{\mathcal{O}}$, as they are still valid values, recompute $\delta f(x', o) o \in \mathcal{O} \setminus \tilde{\mathcal{O}}$ and save them
4. go back to (2)

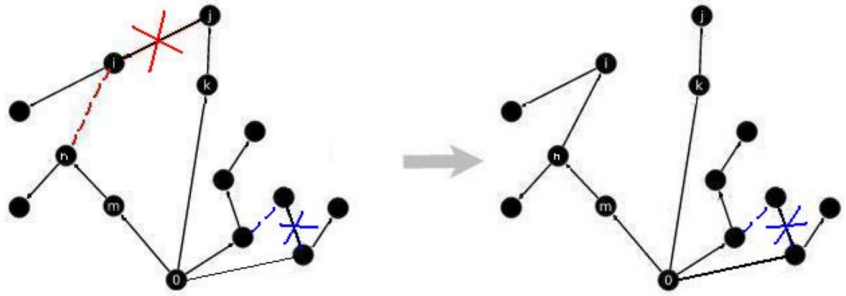


Figure 8.20: A move in the N_{S_1} neighbourhood for the *CMSTP*.

Example: the *CMSTP* Consider the *CMSTP* instance in Figure 8.20. Suppose that both the red exchange, that consists in replacing edge (i, j) with edge (i, n), and the blue exchange are improving: the red exchange decreases the objective function from 100 to 95 and the blue exchange decreases it to 99. The effect of the first exchange is -5 and the effect of the second is -1. The steepest descent algorithm selects the first exchange, obtaining the solution on the right side of the picture. One can easily see that the blue exchange is still applicable in the new solution, as it involves different subtrees. Its effect is still a decrease of -1 in the objective function, making it 94. This exchange is also feasible as it was before, because nothing changed in the distribution of vertices in the two subtrees concerned with the exchange. Therefore, one can build the set of feasible exchanges with the associated variations δf, perform the best exchange, remove from the set of feasible exchanges those that operate on the same subtrees and recompute the exchanges on those subtrees, instead of recomputing all possible exchanges.

Such a technique cannot be used for the *MDP*, as every exchange interacts with every other exchange: the objective function is given by the sum of the total distance of every point with every other point in the solution, so every point in every solution interferes with the value of the objective function.

8.4.4 Tradeoff between efficiency and effectiveness

The choice of the neighbourhood implies a tradeoff between efficiency and effectiveness. In fact, the complexity of the heuristic depends on the number of iterations performed (t_{max}) which is hard to predict a priori, on the number of subsets generated to visit the neighbourhood (strictly related to its cardinality) and to the complexity of computing the feasibility and cost of every subset, as discussed in the previous sections.

It is rather clear that the first two elements, the number of iterations and the cardinality of the neighbourhood, conflict with each other: a very small neighbourhood decreases the latter, but probably increases the former, because small neighbourhoods do not have many good solutions and, therefore, the objective is decreased less at each step. A large neighbourhood, on the contrary, might have very good solutions and decrease the objective in fewer steps. So, the overall speed depends on both factors, and one has to find a neighbourhood large enough to contain many good solutions, but small enough to be explored quickly. The tradeoff is particularly complex as different neighbourhoods also induce different local optima. Enlarging the

size of the neighbourhood improves the effectiveness of the algorithm by an unknown amount, while the efficiency quickly worsens.

8.4.5 Fine tuning of the neighbourhood

Once a neighbourhood is defined, one can still tune apply small modifications to it, in order to find better tradeoffs between effectiveness and efficiency. For example, some elements of the ground set can be more promising than other ones (the simplest case is when the objective function is additive and some elements have lower costs or higher values). Instead of evaluating all the possible exchanges defined by a neighbourhood N, therefore, one could restrict to the exchanges between element of low cost out of the solution with elements of high cost in it. This corresponds to defining a subneighbourhood of N, with a smaller size: the efficiency improves, the effectiveness probably worsens, but the effect might be stronger for the first than for the second. For example, one could find (nearly) the same local optimum in (much) shorter time.

Instead of considering the data, one can identify promising elements of the ground set based on the results obtained in the previous steps. This sounds like a memory mechanism, but it is not based on several runs of the whole algorithm. The idea is simply to notice that, when visiting a large neighbourhood, possibly good improvements over the current one are found immediately: does it make sense to go on and find the best solution overall in the neighbourhood?

The steepest descent algorithm applies the **global-best** strategy, completing the exploration of the neighbourhood and taking the best choice overall. Other exchange algorithms applies the **first-best** strategy, that stops as soon as the first improvement is found

$$\text{if } f(\tilde{x}) < f(x) \text{ then } x := \tilde{x}, \text{break};$$

Such an algorithm probably does not improve much the objective in each single iteration, but it saves a lot of time and performs many more iterations. Whether this is a good idea or not depends on the problem. In a problem with a smooth landscape, typically neighbour solutions are rather similar to each other, so the best solution is not much better than the first improving one, and the first-best strategy may make sense. If the landscape is rugged the best neighbour solution could be much better than the first improving one, so that it may be better to use a global-best strategy. Of course, many intermediate strategies can also be designed between these two extremes.

8.5 Very large scale neighbourhood search

We have already discussed the remark that large neighbourhoods probably include better solutions and lead to better local optima. A neighbourhood whose size is exponential, or a polynomial of high-order with respect to the size of the ground set, could provide very good local optima. However, exploring such a neighbourhood exhaustively would require a high computational time. A solution to this drawback is to find another way to evaluate the solutions, so as to combine a very large neighbourhood with a very small exploration cost. This can be done in two ways:

1. define the neighbourhood in such a way that one can avoid visiting it exhaustively, but apply a suitable algorithm to visit efficiently. After all, the exploration of a neighbourhood is a combinatorial optimization problem: we're trying to find the best solution in a finite set, and this problem is exactly the problem that's being solved with the additional constraint on the contents of the solution.

2. explore the neighbourhood heuristically and return a promising neighbouring solution instead of the best one.

8.5.1 Efficient visit of exponential neighbourhoods

Visiting an exponential neighbourhood in short time requires specific features from the neighbourhood. This section describes, therefore, four specific examples, which are interesting because they can be generalised to families of problems. The first, called *Dynasearch*, is based on the idea of building an auxiliary binary matrix and solving a *Set Packing Problem* on it. The second one involves making cyclic exchanges between components of the solution, and modelling them as the search for negative cost circuits in an auxiliary graph. The last two examples involve shortest path problems in auxiliary graphs: in the case of *order-first split-second* methods, they are easy shortest path problems; in the case of *ejection chains*, they are $\mathcal{NP}$-hard variants.

All examples have in common the idea of transforming the problem of finding the optimal solution in the neighbourhood into an auxiliary subproblem on a combinatorial structure such as a matrix or a graph.

Dynasearch

Section 8.4.3 presented a technique to accelerate the exploration of a neighbourhood by saving the effect of moves evaluated in previous iterations. The idea was that when solutions are divided into components, sometimes operations o and $o' \in \mathcal{O}$ that work on different components of the solution are completely independent.

The idea of Dynasearch is that in the same situation the order in which different **independent** moves are performed becomes irrelevant and, for additive objective functions, their effects are simply summed:

$$f(o'(o(x))) = f(o(o'(x))) = \delta f(o) + \delta f(o')$$

with no dependence on x. The problem is now to find the *best combination* of reciprocally independent moves that can be applied to the current solution.

The example of Section 8.4.3 concerned edge swaps for the *CMSTP*, where swaps involving edges in different branches are independent from each other. A less obvious example of this behaviour is portrayed by 2-opt exchanges for the *TSP*. Figure 8.21 represents a solution that starts from vertex u_1 and after following a path to u_n goes back to the first ($u_{n+1} \equiv u_1$).

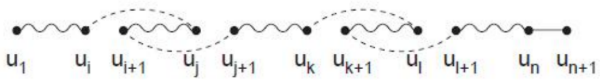


Figure 8.21: TSP instance with two optional exchanges.

Consider the exchange that replaces arcs (u_i, u_{i+1}) and (u_j, u_{j+1}) with arcs (u_i, u_j) and (u_{i+1}, u_{j+1}) and reverts the intermediate subpath, and the exchange that replaces arcs (u_k, u_{k+1}) and (u_l, u_{l+1}) with arcs (u_k, u_l) and (u_{k+1}, u_{l+1}) and reverts the intermediate subpath. Performing the first exchange leaves the second exchange feasible; the reverse is true, as well. The cumulative effect of the two exchanges is the sum of the two simple effects: each effect is independent from the other, and both can be applied together. Consider, instead, the exchange that replaces (u_i, u_{i+1}) and (u_k, u_{k+1}) with (u_i, u_k) and (u_{i+1}, u_{k+1}) and the exchange that replaces (u_j, u_{j+1}) and (u_l, u_{l+1}) with (u_j, u_l) and (u_{j+1}, u_{l+1}), still reversing the intermediate path. These two exchanges are not independent and are actually incompatible, because each

one reverts an arc that should be removed by the other. The same holds for pairs of exchanges where the second acts on the subpath reversed by the first one.

Improvement matrix How to compute the subsets of moves that are reciprocally compatible and have the best total effect? One can build a matrix, called *improvement matrix*, whose rows represent the components of the solution, while the columns represent the elementary moves. The definition of *component* depends on the problem: for a *CMSTP* they will be branches, for the *VRP* circuits, for the *TSP* paths. In general, they are the "parts" are effected by the moves. The improvement matrix is binary: if a move j impacts on component i, then $a_{ij} = 1$, otherwise $a_{ij} = 0$. Each column also has a value, that is equal to the improvement of the objective, $-\delta f$.

The problem is to determine the optimum packing of the columns, that is a maximum value subset of nonconflicting columns that is a *Set Packing Problem*. This is in general $\mathcal{NP}$-hard, but on special matrices it is polynomial. For example, if each move modifies at most two components, the rows can be seen as vertices of an auxiliary graph, the columns as edges, and each packing of columns becomes a matching: finding a maximum matching is a polynomial problem. The improvement matrix generated by the *TSP* example described above does not satisfy this property, but the corresponding *Set Packing Problem* is still polynomial. Even if the problem is not polynomial, one may solve exactly in a reasonable time if its size is sufficiently small, or heuristically, given that the solution is only used to improve the current solution of the original problem in a heuristic method: it is not necessary to obtain exactly the optimum (this can be seen as an application of the first-best strategy).

Complementary to the idea of performing one move and saving the other ones for the following iterations, Dynasearch uses the whole neighbourhood, making a lot of moves, finding a much better solution, but of course has to reexplore the whole neighbourhood at every iteration, to build the new improvement matrix from scratch. Notice that performing a single move is not simply a heuristic way to compute many moves and postpone the execution of the other ones to following iterations, since the following iterations will also introduce additional moves that now are not feasible. So, it is not easy to determine which of the two complementary strategies is better.

Cyclic exchanges

A second way to explore in polynomial time an exponential size neighbourhood is to define it through **cyclic exchanges**. Once again, the ground set should be the Cartesian product of elementary objects and components and the solutions should be structured into components (vertices or edges divided into subtrees, nodes into circuits, objects into containers, and so on), that is, made of pairs (i, S_i). Quite often, the feasibility depends on each single component: a given solution is feasible if each component satisfies suitable conditions. For example, in the *BPP* the volume of each bin must respect the capacity, and the same holds for the *CMSTP*, the *VRP* and so on. Often, the objective function is additive with respect to the components:

$$f(x) = \sum_{S^{(i)} \in x} f(S^{(i)})$$

which is more general than requiring additivity with respect to the single elements. For example, the value of the objective for the *BPP* is a sum of terms related to the single containers: the term is 1 if the container is used, 0 otherwise.

It is natural to define for these problems the set of operations $\mathcal{T}_k$ that includes the transfer of k elements from the current component to another one. The corresponding neighbourhood

is denoted as $N_{\mathcal{T}_k}$. Quite often, a single transfer can be unfeasible: if the additional constraints are tight, moving a single object without violating any constraint is frequently difficult. On the other hand, moving k objects can be done in a really large number of ways, and computing the objective function for each one is expensive, so that the overall complexity of the exploration becomes too large. A good situation would be having a subset of $N_{\mathcal{T}_k}$ that is large, but efficient to explore.

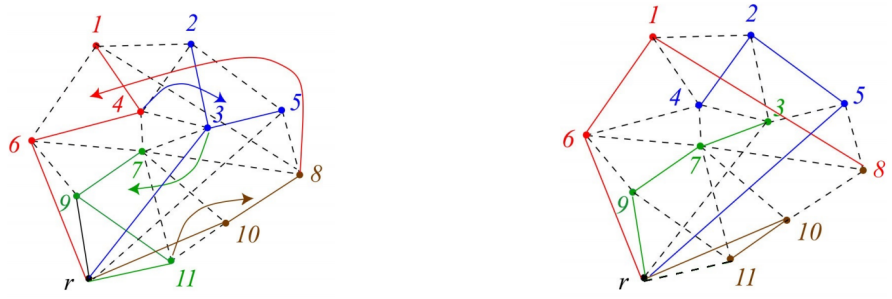
Improvement graph A possible solution to this problem is to find a sequence of k transfers such that each element moved to a container takes the place of another one. In this way, it is more likely that the container is able to host the new element. In particular, in case of capacity constraints, the volume occupied by the incoming element can be compensated by the volume left free by the outgoing one. If the sequence is cyclic, all constraints could be satisfied. The problem to find the best cyclic sequence can be trasnformed into an optimisation problem on an auxiliary graph, known as the **improvement graph**. This graph has the following features:

- each node i corresponds to an element (for example, in the *CMSTP* each node of the improvement graph corresponds to a vertex of the original one);
- each arc (i, j) connects two nodes; this has a complex meaning, as it represents:
  - the entrance of element i into the current component S_j of element j
  - the exit of element j from component its current component S_j
- the cost of an arc, c_{ij}, corresponds to the variation of the contribution that component S_j gives to the objective:
$$c_{ij} = f(S_j \cup \{i\} \setminus \{j\}) - f(S_j)$$
with $c_{ij} = +\infty$ if it is unfeasible to transfer i deleting j.

Notice that an arc does not represent the transfer of an element: it represents two complementary half-transfers, referring to two elements. None of them is self-sufficient, because element i must come from somewhere and element j must go somewhere. However, appending another arc (j,k) to (i, j) provides the missing information for element j, creating a further problem for element k. In the end, it is necessary to create a circuit, so that the overall sequence of arcs represents a correct sequence of transfers. The cost of the circuit corresponds to the cost of the sequence and the feasibility of all arcs corresponds to the feasibility of the sequence.

To be more precise, this is not exactly correct, because circuits that visit several nodes associated to the same component imply a basic problem: it is no longer possible to simply add the costs of the arcs and to combine conjunctively their feasibility. This is because two exchanges involving the same component (for example, two swaps on the same branch of a *CMSTP* solution) might both be feasible while their combination is not, or might imply a variation of the objective different from the sum of the single variations. Therefore, the auxiliary problem must be restricted to circuits that visit at most one node for each component.

Example: the *CMSTP* Figure 8.22 reports an example for the *CMSTP*. The solution on the left is composed by four branches, represented in different colours. The picture on the righe represents the corresponding improvement graph. The circuit $(4,3),(3,11),\ (11,8),(8,4)$ in this graph represents a compound cyclic move: vertex 4 moves into the blue subtree to replace vertex 3, vertex 3 moves into the green subtree to replace vertex 11, vertex 11 moves into the brown subtree to replace vertex 8 and, finally, vertex 8 moves into the red subtree to replace vertex 4.



(a) A solution with possible exchanges.

(b) After the exchanges.

Figure 8.22: Moves on the solution for the *CMSTP*.

Is this solution feasible? From the structural point of view it is: all vertices remain assigned to exactly one branch. From the point of view of the capacity constraint, the feasibility depends on the single arcs, that is on the difference between the weights of the two extreme vertices: if they are the same, the move is feasible; if they are different, one must compare with the capacity the original weight of the subtree S_j plus the weight w_i of the new vertex minus the weight w_j of the displaced vertex. If all arcs are feasible, then the compound move is feasible. If an arc corresponds to an unfeasible move, the arc must be forbidden, typically setting its cost to $+\infty$. The total cost of the circuit, that is the sum of the costs of the single arcs ($c_{4,3} + c_{3,11} + c_{11,8} + c_{8,4}$), is also the sum of the variations of cost incurred by the four subtrees involved ($c_{S_{\text{blue}}} + c_{S_{\text{green}}} + c_{S_{\text{brown}}} + c_{S_{\text{red}}}$). Each of these costs is computed comparing the minimum tree spanning the new vertices with the one spanning the original vertices. This is not a trivial task, but can be done in polynomial time, and updating the cost with some smart procedure is certainly better than recomputing it from scratch.

Search for the minimum cost circuit The problem of finding the minimum cost circuit is actually $\mathcal{NP}$-hard, even neglecting the condition on visiting at most one node per component. In fact, it generalises the *TSP* (if all costs are negative, the circuit necessarily visits all vertices). However, several nice properties can be exploited to reduce the computational cost to a manageable amount.

First of all, the constraint of visiting each component only once allows a rather efficient dynamic programming algorithm, since the circuit has at most r arcs for solutions made of r components.

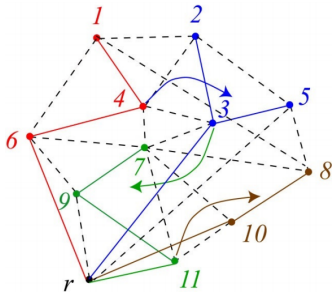
A second property that strongly reduces the complexity is that we are looking for a circular sequence of numbers with a negative sum. Now, such a sequence always admits a cyclic permutation such that all its partial sums are negative. For example, given $+1$, $+2$, $+3$ and -10, whose sum is negative, at least one of the four cyclic permutations ($2+3-10+1$, $3-10+1+2$ or $-10+1+2+3$) has all partial sums smaller than zero: in this case, it is $-10+1+2+3$. Since a cyclic permutation of a circuit is the same circuit with a different starting node, instead of generating all circuits starting from a given node, we can generate all circuits starting from each node, but immediately drop any path with a nonnegative partial sum. Such a path either is not part of any negative cost circuit or it is part of a negative cost circuit which can be found in another way.

Alternatively to the exact computation of the best circuit, one can adopt heuristic algorithms. The Floyd-Warshall algorithm, for example, that finds all the point to point shortest paths in a graph, but fails when the graph includes a negative circuit, returns anyway such a circuit: it is a heuristic solution, because in general it is not the minimum one. The minimum average

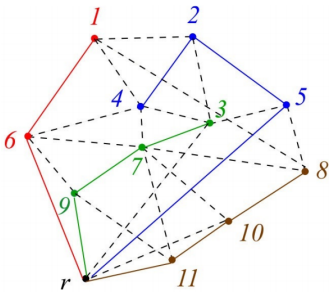
cost circuit (computed as the total cost divided by the number of arcs) can be computed in polynomial time: it is not the optimal solution of the auxiliary subproblem, but a heuristic one. To be precise, in both cases we are ignoring the constraint on the components, which is a relaxation of the auxiliary subproblem: if the result is anyway feasible, it can be used; if it is not feasible, but has a nonnegative cost, it is a proof that negative circuits do not exist; if it is infeasible, but negative, it could possibly be modified to obtain a feasible one.

Noncyclic exchange chains

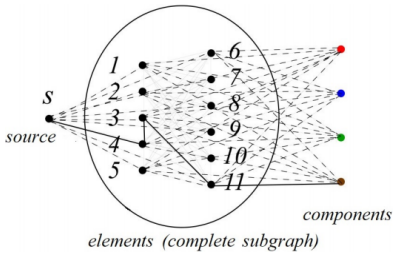
Cyclic exchanges have the basic property that the number of elements in each component remains the same. We have already seen this property in the case of swap neighbourhoods, that keep the same number of elements in the solution, and we have seen that for some problems this disconnects the search space. Keeping the same number of elements in each component makes the problem more critical and concerning many more problems. If the optimal solutions have different cardinalities in some components, they are impossible to obtain.



(a) CMSTP solution and possible exchanges.



(b) After the exchanges.



(c) The extended noncylic exchange improvement graph.

Figure 8.23: CMSTP instance and its noncylic improvement graph.

In order to change the cardinality of each component, one can build a chain of transfer moves such that is noncyclic: it starts by removing an element, putting it into another component, possibly making a chain of transfers and, finally, putting the last element of the chain in a component different from the first one. The improvement graph can be augmented to represent also this kind of transfer chain simply adding:

- a source node, which is fictitious and doesn't represent any node of the problem;
- a sink node associated with each component;
- arcs from the source node to the original nodes associated with the elements;
- arcs from the original nodes associated with the elements to the nodes associated with the components.

The arcs from the source node represent the removal of an element, so they are always feasible (for capacity constraints). The arcs from the original nodes to the component one represent the

insertion of an element in a component: they are feasible only if the residual capacity of the component allows the insertion of the element.

Figure 8.23 (c) shows the resulting graph: the subgraph in the middle is the original improvement graph, in which every node is connected to every other node, apart from the arcs made infeasible by the capacity constraints; the node on the left is the new source node; the nodes on the right are the new components nodes, one for each subtree. Consider now the path

$$(s,4),(4,3),(3,11),(11,S_4)$$

This represents the transfer chain that takes node 4 out of the red branch (first arc), moves it to replace node 3 in the blue branch (second arc), moves the latter to replace 11 in the green branch (third arc), and finally moves 11 into the brown subtree. Every transfer chain can be represented by a path going from nodes s to one of the component nodes S_k in the improvement graph. Again, the feasibility of the transfer chain is guaranteed by the feasibility of the single arcs and the total cost is the sum of the single arc costs provided that each component is visited at most once (of course, using the component node counts as a visit of the component).

Order-first split-second

We consider once again problems in which the solutions are partitions of elements into components. There is another way to build an auxiliary graph and use the shortest path problem as a tool to find an optimal solution in an exponential neighbourhood. These methods are known as **order-first split-second** methods.

The idea is the following: first, put the elements of the ground set in a suitable permutation. Then, define the set of all partitions in which the elements of each component appear consecutively in the permutation. Each partition is a subset of the ground set (feasible or not, according to the other constraints of the problem). The overall set is an exponential collection of subsets. The feasible ones can be seen as a neighbourhood, if the permutation is obtained from a starting solution. It is not the whole solution set, because some solutions keep in the same component elements that are not consecutive in the permutation.

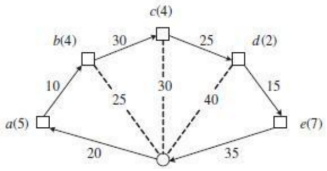
An interesting problem is how to find the best solution in this exponential set. There is a nice polynomial way to solve this problem, that is to explore this specific neighbourhood. This can be integrated into a two-level method in which the upper level (i.e., an outer loop) generates permutations and the lower level (i.e., an inner loop) finds the optimal partition for the current permutation. The limitation of this method is that no smart way is known to select an effective permutation. Moreover, different permutations can produce the same solution, since there are many more permutations than solutions. The method risks, therefore, to be inefficient.

The auxiliary graph To find the optimal partition corresponding to a given permutation $(s_1,\cdots,s_n)$, one builds an auxiliary graph such that each node s_i corresponds to an element s_i, with an additional fictitious node s_0. Then, each pair (s_i,s_j) with $i<j$ corresponds to a potential component $S_\ell=(s_{i+1},\cdots,s_j)$, formed by the elements of the permutation from s_i excluded to s_j included; each pair (s_0,s_j) corresponds to a potential component $S_\ell=(s_1,\cdots,s_j)$, formed by the elements of the permutation from s_1 to s_j, both included. If such a component is unfeasible, the pair does not correspond to an arc; if it is feasible, the arc (s_i,s_j) is added to the graph, with a cost c_{s_i,s_j} equal to the cost of the component, $f(S_\ell)$.

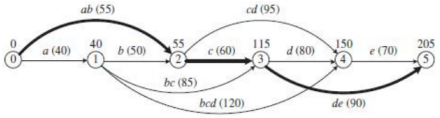
Given the graph, any path from the source node s_0 to the last node s_n corresponds to a solution, that is a partition of the elements: the arcs, in fact, identify disjoint components that overall contain all elements. The cost of the path coincides with the cost of the solution. Therefore, the

optimal path from s_0 to s_n corresponds to the best solution that can be represented by the graph. Since the graph is acyclic, finding the optimum path costs $\mathcal{O}(m)$, where $m \leq n(n-1)/2$ is the number of arcs.

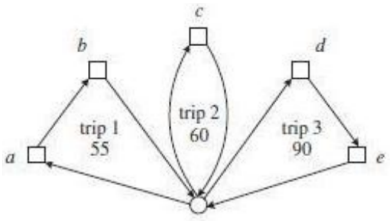
Example: the *VRP* Consider the instance of the *VRP* represented in Figure 8.24, with a depot node and five customer nodes. Each customer has an associated weight, each arc an associated cost; the capacity of the vehicles is set to $W = 10$. The problem aims to find a set of vehicle to service all customers at minimum total cost, without exceeding the capacity of each vehicle.



(a) VRP instance.



(b) The order first split second graph.



(c) The optimal path.

Figure 8.24: VRP instance and its OFSS solution.

Figure 8.24 (b) reports the auxiliary graph associated with the permutation (a,b,c,d,e). There are five nodes representing the customers and a fictitious node 0. The arcs go only *forward* from nodes of smaller index to nodes of larger index. Arc $(1,2)$ represents a component (a circuit visited by a vehicle) that includes only customer 2; arc $(1,3)$ represents a component that includes elements 2 and 3 and so on. The costs of the arcs are the costs of such circuits. Theoretically, in order to compute them, one sholud solve a *TSP* problem for each potential component. This can be done exactly as long as the number of customers is small, as in this case. When the arcs represent large components, one must use a heuristic. For example, arc $(0,2)$ has a cost equal to $20+10+25=55$, because it represents a circuit visiting customers 1 and 2 (or 2 and 1), and this is the minimum cost to do it. Arc $(0,3)$ does not exist, because it would represent a circuit visiting customers 1, 2 and 3, but their total weight is $5+4+4$, which exceeds the capacity of the vehicles, making such a component unfeasible.

8.5.2 Heuristic visit of large neighbourhoods

So far, we have considered large neighbourhoods whose optimal solution is computed exactly, at least theoretically and at the cost of having to solve an NP-hard problem. In other cases, the neighbourhood is visited with a heuristic without even trying to find the best solution exactly. This is in particular the case of neighbourhoods whose size is not even exactly defined *a priori*.

8.5.2.1 Variable depth search

The **variable depth search** method is based, once again, on the idea of parameterising an elementary neighbourhood: given a set of operations $\mathcal{O}$, a sequence of k operations from $\mathcal{O}$

generate a subset that, if feasible, belongs to a more general neighbourhood

$$N_{\mathcal{O}_k} = \{x' \in X : \forall o_1, \cdots, o_k \in \mathcal{O}\ x' = o_k(o_{k-1}(\cdots(o_1(x)))),\}$$

If k is small, this neighbourhood contains only few solutions, probably not of high quality; if k is large, the neighbourhood includes very good solutions, but exploring them takes a very high computational cost. The idea is to build a composite move as a sequence of elementary moves. Contrary to the case of *Dynasearch* the elementary moves are not required to be independent; on the contrary, they will be bound to move further and further from the original solution at each move.

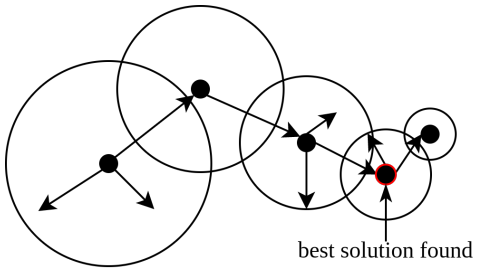


Figure 8.25: Schematic of VDS search.

The basic idea is to visit a simple neighbourhood of the given solution. Instead of using the value of the objective function in each neighbour solution to select the next current one, the algorithm could run a whole local search procedure from each neighbour solution to determine a locally optimal solution. This could be seen as a sort of rollout or lookahead algorithm, as done in Section 6.4.4 to extend the basic constructive scheme. The rollout algorithms add each potential element to a current subset and run the constructive heuristic to completion, using the final result as a selection criterion to choose the best element to add.

In an exchange algorithm, one could try to run a local search algorithm to completion starting from each neighbour of the current solution. This is not promising, because the local search procedure could easily go back to the original solution, or stop immediately. However, the scheme can be refined as follows:

- consider each neighbour solution obtained with an elementary move;
- make a sequence of moves optimizing each elementary step, *forbidding backward moves* and also *allowing worsening moves*
- terminate when the sequence when the current solution becomes worse than the starting one: this makes the length k not fixed *a priori*
- return the best solution found along the sequence.

So, referring to Figure 8.25, in each step the objective function can both improve and worsen (after being improved), provided that it never gets above the original cost.

The rollout algorithm multiplied the complexity of the basic constructive heuristic by a factor n^2, because it made $O(n)$ attempts for each of the $O(n)$ steps, and this strongly affected its efficiency. Using this technique for an exchange heuristic could be much worse, because the number of neighbour solution can be much larger than $O(n)$ and the number of steps of the single local search procedures is usually not bounded by $O(n)$ (first, they go on at least to a local optimum, with an unknown number $t_{\max}$ of steps, then they proceed, because worsening moves are allowed). So the idea in its basic form is inefficient.

In order to improve it, one applies a number of refinements. First, the auxiliary local search procedures apply the first-best strategy, instead of the global-best strategy. Second, also the visit of the neighbourhood of the initial solution stops at the first improvement. Third, the original neighbourhood N is used to generate the starting solutions, but the auxiliary local search procedures use a neighbourhood $\hat{N}$ that is strictly smaller, so that they are comparatively faster. Moreover, in order to guarantee that the initial solution is not regenerated, the neighbourhood $\hat{N}$ forbids to reverse the moves performed in the previous steps of the sequence, by progressively fixing part of the solution, so that it can no longer be modified. This increases the distance from the starting and gradually restricts the neighbourhood at each step. It also guarantees a maximum value of k of the order of $O(n)$ for the length of the sequence of moves. Finally, the local search procedures stop as soon as they find solutions not better than the starting one. If this happens at the first step, the exploration stops immediately.

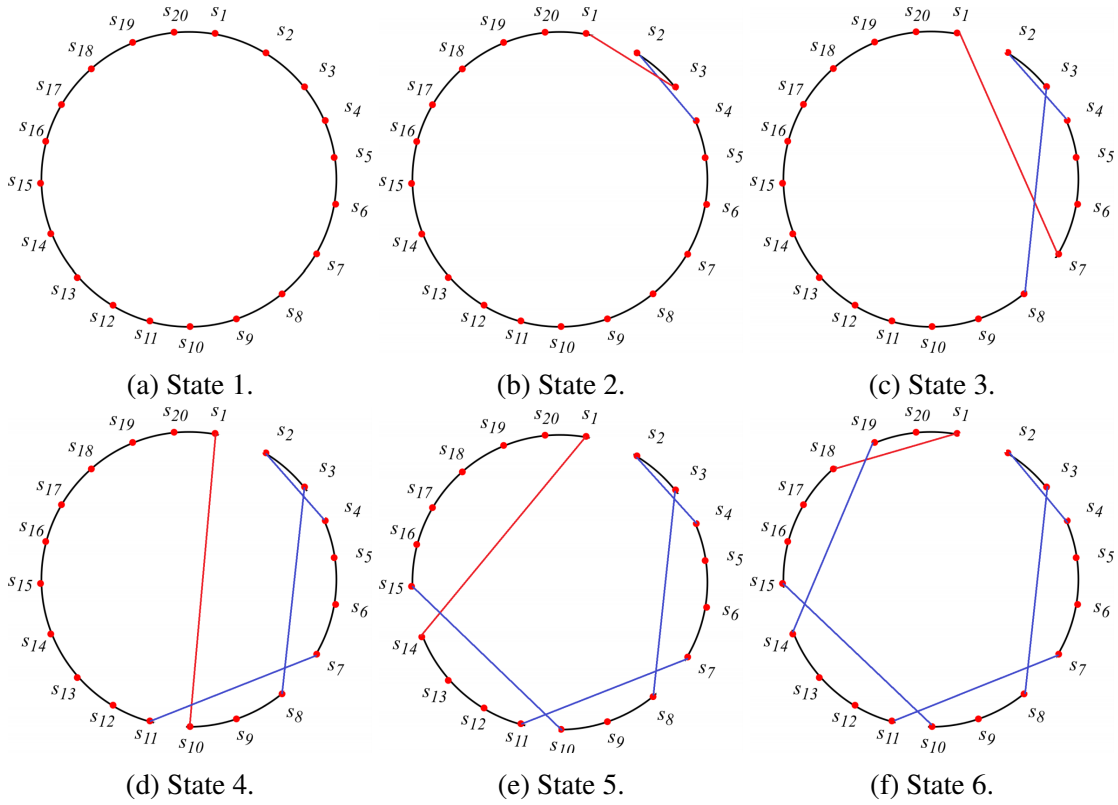
Algorithm 13 Variable Depth Search Pseudocode

```
1:  procedure VDS(x')
2:      y := x'
3:      y* := x'
4:      Stop := false
5:      while Stop = false do
6:          ỹ := arg min_{y∈N̂(x)} f(y')
7:          if f(ỹ) ≥ f(x^(t)) then
8:              Stop := true
9:          else
10:             y := ỹ
11:         end if
12:         if f(ỹ) < f(y*) then
13:             y* := ỹ
14:         end if
15:     end while
16:     Compute f(y*)
17: end procedure
```

Why stop as soon as a solution worse than the starting one is found? The reason is, once again, the nice property concerning the cyclic permutations of numbers with negative sum discussed when presenting the cyclic exchanges: a sequence of moves that is improving can be performed starting from a point such that all subsequences are also improving. Therefore, we can require that all subsequences are improving with respect to the starting solution.

Lin-Kernighan's algorithm for the symmetric *TSP* A classical application of *VDS* is the **Lin-Kernighan algorithm** for the symmetric *TSP*, which is currently the best performing heuristic algorithm for that problem (actually, in a variation that includes refinements of the neighbourhood and exploits additional information).

The basic neighbourhood N_{R_k} consists in deleting k arcs from a solution x and adding other k arcs so that the new solution is still a Hamiltonian circuit. The Lin-Kernighan algorithm applies a composite moves that consist of sequences of 2-opt exchanges. Notice that every k-opt exchange is equivalent to a sequence of 2-opt exchanges of length $(k-1)$: each exchange deletes one of the two arcs added by the previous exchange.



(a) State 1. (b) State 2. (c) State 3.

(d) State 4. (e) State 5. (f) State 6.

Figure 8.26: Chain of moves in Lin-Kernighan's algorithm for the symmetric *TSP*.

Starting from solution x, the algorithm visits (with a first-best strategy) the whole neighbourhood $\mathcal{N}_{R_2}(x)$. For each solution x', it runs a sequence of 2-opt exchanges as follows. Suppose that x' has been obtained working on indices (i,j) (delete (s_i,s_{i+1}) and (s_j,s_{j+1}), add (s_i,s_j) and (s_{i+1},s_{j+1})). Evaluate the 2-opt exchanges that delete the added arc (s_i,s_j) and each arc of $x \cap x'$ to find the best exchange (i',j'); if this improves upon x (not necessarily upon x'!), perform the exchange and obtain x''. Then, evaluate all the exchanges that delete $(s_{i'},s_{j'})$ and each arc of $x \cap x''$ and evaluate the best, and so on. Terminate when the current solution is not better than x' and return the best solution among $x',x'',\cdots$. If this is better than x, the first-best strategy accepts it; otherwise, consider the next neighbour of x in $\mathcal{N}_{R_2}(x)$.

Starting from the Hamiltonian circuit of Figure 8.26 (a), the algorithm the N_{R_2} neighbourhood consists of $n(n-2) = 20*18/2 = 180$ solutions, associated with the pairs of nonadjacent arcs. The first exchange is $(1,3)$, that removes arcs (s_1,s_2) and (s_3,s_4), reverses the path $(s_2,\cdots,s_3)$ and adds the arcs (s_1,s_3), (s_2,s_4), producing the solution reported in Figure 8.26 (b). The sequence of exchanges will start from this solution. It considers not all R_2 exchanges, but only the $O(n)$ exchanges that remove (s_1,s_3) and one of the other original edges. The algorithm explores, one by one, each possible solution: let us say that the best one is $(1,7)$, that is removing (s_1,s_3) and (s_7,s_8), adding (s_1,s_7) and (s_3,s_8) and reversing the path $(s_3,\cdots,s_7)$ (Figure 8.26 (c)). One can interpret this as the result of a 3-opt exchange applied on x, as well as a sequence of two 2-opt exchanges with a constraint on the choice of the arcs.

The algorithm goes on with a third exchange that removes arc (s_1,s_7) and tries to remove one of the remaining original arcs, keeping all those that have been added by the procedure (the blue arcs in the pictures). Let us say that the best exchange is $(1,10)$, so that (s_1,s_7), (s_{10},s_{11}) are removed, (s_1,s_{10}) and (s_7,s_{11}) are added and path $(s_7,\cdots,s_{10})$ is reversed (Figure 8.26 (d)). Let us suppose, then, that the algorithm chooses exchange $(1,14)$ (Figure 8.26 (e)), resulting in the

situation reported in Figure 8.26 (f), and suppose that the best exchange that removes (s_1, s_{18}) yields an objective function value that is worse than that of x'. At this point, the algorithm stops and returns the best solution found in the sequence. This is compared to x and replaces x if it is better; otherwise, the following starting point in $\mathcal{N}_{R_2}(x)$ is considered.

Implementation details As mentioned above, the second arc deleted each time must belong to the original solution x, in order to avoid destroying the moves already performed. This implies an upper bound on the length of the sequence.

Stopping the sequence as soon as the exchanges no longer improve x' does not impair the result. In fact, the overall variation of the objective function is the sum of the variations due to the single exchanges (of course, if the objective function is additive):

$$\delta f(x, o_1, \cdots, o_k) = \sum_{\ell=1}^{k} \delta f(x, o_\ell)$$

and every sequence of numbers with a negative sum admits a cyclic permutation whose partial sums are all negative. Therefore, a cyclic permutation of the same moves $o_1, \cdots, o_k$ would provide the same result and also have negative sum at each step.

8.5.2.2 Iterated greedy method (destroy and repair)

The **iterated greedy method**, also called **destroy and repair** or or **ruin and recreate**, is based on the remark that all exchange heuristics consist in adding a subset A of elements and removing a subset D of elements from the current solution. Performing an exhaustive visit of the neighbourhood implies that A and D should have a small size, otherwise the number of subsets tested would become huge. However, in this way, the result could be frequently unfeasible (as in the transfer neighbourhood for the *BPP*) or nonimproving (as in the swap neighbourhood for the *MCP*). On the other hand, enlarging the neighbourhood to exchanges of more elements quickly becomes inefficient. The complexity of an exhaustive exploration, in fact, is $\mathcal{O}(n^{|A|}n^{|D|}\gamma(n))$, where

- the number of possible subsets to add is $\mathcal{O}(n^{|A|})$;
- the number of possible subsets to remove is $\mathcal{O}(n^{|D|})$;
- $\gamma(n)$ is the complexity of evaluating feasibility and the objective function for a given subset.

The problems in which solutions have variable cardinalities (such as the *KP* and the *SCP*) also make it difficult to determine the correct sizes of A and D.

An alternative approach can be the use of heuristics to determine a subset $D \subset x$ of elements to delete and a subset $A \subset B \setminus x$ of elements to add. In particular, the deletion heuristic could be (at least partly) random (and dependent on the costs of the elements, or other functions related to the constraints of the problem), whereas the addition heuristic could be a simple constructive heuristic. Complementarily, one could first add redundant elements with a random choice of elements (depending on costs and constraints) and reduce it to a minimal one by applying a simple destructive heuristic.

Even if it is possible to design perfectly deterministic schemes for this idea, randomisation and/or memory are often used. These methods, therefore, generally fall under the class of metaheuristics.

8.6 Exercises

8.6.1 Exercise 1

Consider the following instance of the *Knapsack Problem* with capacity $V = 10$:

Objects	a	b	c	d	e	f	g
Prize ϕ	10	3	5	12	7	6	8
Volume v	5	1	2	4	3	1	6

and the starting solution $x^{(0)} = \{b, e, g\}$.

How many solutions are contained in neighbourhood $N_{\mathcal{H}_1}(x^{(0)})$ (that is, Hamming distance not larger than 1)?[2]

How many solutions are contained in neighbourhood $N_{\mathcal{S}_1}x^{(0)}$ (that is, swap an element in the solution with one out of it)?

How many solutions are contained in neighbourhood $N_{\mathcal{H}_2}x^{(0)}$ (that is, Hamming distance not larger than 2)?

Perform a single iteration of *steepest descent* with the *global best strategy* adopting neighbourhood $N_{\mathcal{S}_1}$.

Perform a single iteration of *steepest descent* with the *first-best strategy* visiting neighbourhood $N_{\mathcal{S}_1}$ in lexicographic order with respect first to the exiting object than to the entering one.

Solution Neighbourhood $N_{\mathcal{H}_1}$ is obtained flipping a single bit of the incidence vector of the starting solution $x^{(0)} = \{b, e, g\}$, that is $\xi^{(0)} = [\,0\,1\,0\,0\,1\,0\,1]$. The most approximate estimate of the required number, therefore, is

$$\left|N_{\mathcal{H}_1}(x^{(0)})\right| \in O(n)$$

Since $n = 7$ subsets can be generated in this way, a better estimate is

$$\left|N_{\mathcal{H}_1}(x^{(0)})\right| \leq n = 7$$

The neighbourhood contains in general only the feasible solutions, unless the algorithm relaxes this constraint (which we have never done in the course). The $|x^{(0)}| = 3$ removals certainly yield feasible solutions. The $n - |x^{(0)}| = 4$ additions must be checked comparing the additional volume v_i with the residual capacity $V = \sum_{i \in x} v_i = 10 - (1+3+6) = 0$: they are all unfeasible. The most precise answer is, therefore:

$$\left|N_{\mathcal{H}_1}(x^{(0)})\right| = 3$$

Neighbourhood $N_{\mathcal{S}_1}$ is obtained exchanging an object inside the starting solution $x^{(0)} = \{b, e, g\}$ with one outside. The asymptotic estimate is

$$\left|N_{\mathcal{S}_1}(x^{(0)})\right| \in O\left(n^2\right)$$

[2] In all answers, the starting solution $x^{(0)}$ is never counted in the definition and in the enumeration of the neighbourhoods for the sake of simplicity and because it is totally useless for the algorithm. It is of course correct to count it, increasing by 1 all answers.

The precise estimate of the number of subsets generated by these moves is

$$\left|N_{\mathcal{S}_1}(x^{(0)})\right| \leq |x|\,(n-|x|) = 12$$

In order to check the feasibility, one needs to compute the additional volume occupied, that is the difference between the volume of the added object and that of the removed one, $v_j - v_i$ for all $i \in x$, $j \in E \setminus x$ and keep only the values that do not exceed the residual capacity (that is 0):

$-v_i + v_j$	$v_a = 5$	$v_c = 2$	$v_d = 4$	$v_f = 1$
$v_b = 1$	4	1	3	**0**
$v_e = 3$	2	**-1**	1	**-2**
$v_g = 6$	**-1**	**-4**	**-2**	**-5**

The most precise answer is, therefore:

$$\left|N_{\mathcal{S}_1}(x^{(0)})\right| = 7$$

Neighbourhood $N_{\mathcal{H}_2}$ is obtained flipping at most two bits of the incidence vector of the starting solution $x^{(0)}$. The most approximate estimate of the required number, therefore, is

$$\left|N_{\mathcal{H}_2}(x^{(0)})\right| \in O\left(n^2\right)$$

and the precise estimate of the number of subsets generated by these flips is

$$\left|N_{\mathcal{H}_2}(x^{(0)})\right| \leq n + n(n-1) = 49$$

The precise neighbourhood is obtained, as before, comparing the variation of the total volume of the solution, δv and keeping only the nonpositive values:

Flip	j						
i (v_i)	a (5)	b (1)	c (2)	d (4)	e (3)	f (1)	g (6)
a (5)	-	$+5-1=+4$	$+5+2=+7$	$+5+4=+9$	$+5-3=+2$	$+5+1=+6$	$+5-6=-1$
b (1)	-	-	$-1+2=+1$	$-1+4=+3$	$-1-3=-4$	$-1+1=0$	$-1-6=-7$
c (2)	-	-	-	$+2+4=+6$	$+2-3=-1$	$+2+1=+3$	$+2-6=-4$
d (4)	-	-	-	-	$+4-3=+1$	$+4+1=+5$	$+4-6=-2$
e (3)	-	-	-	-	-	$-3+1=-2$	$-3-6=-9$
f (1)	-	-	-	-	-	-	$+1-6=-5$
g (6)	-	-	-	-	-	-	-

to which one must sum the 3 feasible single flips, already evaluated above. The most precise answer is, therefore:

$$\left|N_{\mathcal{H}_2}(x^{(0)})\right| = 12$$

The neighbourhood exploration requires to compute the variations of the objective function for all swaps, which is done considering the differences in values, as done above with the differences in volume:

$-\phi_i + \phi_j)$	$\phi_a = 10$	$\phi_c = 5$	$\phi_d = 12$	$\phi_f = 6$
$\phi_b = 3$	NF	NF	NF	3
$\phi_e = 7$	NF	-2	NF	-1
$\phi_g = 8$	2	-3	4	-2

where NF stands for "not feasible" (the values exceeding the residual capacity in the matrix above, that reported the variation of occupied volume for the swap moves).

The global-best strategy selects the best neighbour solution, that corresponds to the largest increase in objective function, that is swap (g,d), with $\delta f = 4$. The resulting solution is $x' = \{b,d,e\}$ with value $f(x') = 3+12+7 = 22$, that is obviously equal to $f(x)+\delta f = 18+4$.

The first-best strategy scans the neighbourhood in lexicographic order, that is row by row. It stops at the first improving swap, that is (b,f), with $\delta f = 3$. The resulting solution is $x'' = \{e,f,g\}$ with value $f(x'') = 7+6+8 = 21$, that is obviously equal to $f(x)+\delta f = 18+3$. It is worse than x', but it has been obtained in one third of the time.

8.6.2 Exercise 2

Given the following instance of the symmetric *Travelling Salesman Problem* (*TSP*)

Cost	a	b	c	d	e	f
a	0	8	8	10	9	6
b	8	0	10	8	4	8
c	8	10	0	9	9	8
d	10	8	9	0	8	7
e	9	4	9	8	0	7
f	6	8	8	7	7	0

and the solution $x = (a,b,c,d,e,f)$, how many solutions are contained in neighbourhood $N_{\mathcal{R}_2}$ (that is, the 2-opt exchange neighbourhood)?

Perform a single iteration of *steepest descent* with the *global best strategy* adopting neighbourhood $N_{\mathcal{R}_2}$.

Perform a single iteration of *steepest descent* with the *first-best strategy* visiting neighbourhood $N_{\mathcal{R}_2}$ in the order determined by the sequence of nodes (that is also the lexicographic order in this case).

Solution Neighbourhood $N_{\mathcal{R}_2}$ is obtained removing two nonconsecutive arcs, adding (in a single possible way) two other arcs and reversing the sequence of nodes between the removed arcs. In a complete graph, all the subsets thus obtained are feasible solutions. Therefore:

$$\left|N_{\mathcal{R}_2}(x)\right| = \frac{n(n-3)}{2} = 9$$

The starting solution $x = (a,b,c,d,e,f)$ has a cost equal to $f(x) = 8+10+9+8+7+6 = 48$ The variations of the cost corresponding to all such exchanges are reported in the following table, where rows and columns identify the removed arcs, while the added ones are implicitly determined by the former:

δf	(a,b)	(b,c)	(c,d)	(d,e)	(e,f)	(f,a)
(a,b)	-	-	$-8-9+8+8 = -1$	$-8-8+10+4 = -2$	$-8-7+9+8 = 2$	-
(b,c)		-	-	$-10-8+8+9 = -1$	$-10-7+4+8 = -5$	$-10-6+8+8 = 0$
(c,d)			-	-	$-9-7+9+7 = 0$	$-9-6+8+10 = 3$
(d,e)				-	-	$-8-6+7+9 = 2$
(e,f)					-	-
(f,a)						-

where label "-" marks the exchanges that are impossible because they concern adjacent or identical arcs, and the lower part of the matrix is empty because the exchanges are symmetric.

The global-best strategy selects the best neighbour solution, that corresponds to the largest decrease in objective function, that is remove arcs (b,c) and (e,f) and add arcs (b,e) and (c,f), with $\delta f = -5$. The resulting solution is $x' = (a,b,e,d,c,f)$ with value $f(x') = 8+4+8+9+8+6 = 43$, that is obviously equal to $f(x)+\delta f = 48-5$.

The first-best strategy scans the neighbourhood in the indicated order, that is row by row. It stops at the first improving swap, that is remove arcs (a,b) and (c,d) and add arcs (a,c) and (b,d), with $\delta f = -1$. The resulting solution is $x' = (a,c,b,d,e,f)$ with value $f(x') = 8+10+8+8+7+6 = 47$, that is obviously equal to $f(x)+\delta f = 48-1$. It is worse than x', but it has been obtained in one ninth of the time.

8.6.3 Exercise 3

Given the following instance of the *Parallel Machine Scheduling Problem* (*PMSP*) with 3 machines:

Task	a	b	c	d	e
d	9	3	4	5	8

assume that the current solution x assigns tasks a and b to the first machine, tasks c and d to the second machine and task e to the third one.

How many solutions are contained in neighbourhood $N_{\mathcal{T}_1}(x)$ (that is, the transfer of a task to a different machine)? Does this number depend on solution x?

How many solutions are contained in neighbourhood $N_{\mathcal{S}_1}(x)$ (that is, the swap of two tasks)? Does this number depend on solution x?

Perform a single iteration of *steepest descent* with the *global best strategy* starting from solution x and adopting neighbourhood $N_{\mathcal{T}_1}$.

Perform a single iteration of *steepest descent* with the *first-best strategy* starting from solution x and visiting neighbourhood $N_{\mathcal{T}_1}$ in lexicographic order first with respect to the tasks, then to the machines.

Do the same adopting neighbourhood $N_{\mathcal{S}_1}$; in the case of the first-best strategy, sort the neighbour solutions in lexicographic order with respect to the first and then the second task swapped.

Solution How many solutions are contained in neighbourhood $N_{\mathcal{T}_1}x$ (that is, the transfer of a task to a different machine)? Does this number depend on solution x?

Neighbourhood $N_{\mathcal{T}_1}$ is obtained trasferring a single task from the current machine to another one. Since all transfers are feasible, the required number is

$$\left|N_{\mathcal{T}_1}(x)\right| = |T| \cdot (|M|-1) = 10$$

and it is independent from the solution.

Neighbourhood $N_{\mathcal{S}_1}$ is obtained swapping two tasks, that is assigning each one to the machine currently assigned to the other. All swaps are feasible, but those that involve tasks currently assigned to the same machine yield the starting solution. Therefore, they are indistinguishable (and we are not counting them, even if counting them as a single solution is correct).

Let $T_m(x)$ denote the subset of tasks assigned to machine $m \in M$ in solution x. The required number is

$$\left|N_{S_1}(x)\right| = \frac{|T|\cdot(|T|-1)}{2} - \sum_{m\in M}\frac{|T_m(x)|\cdot(|T_m(x)|-1)}{2} = \frac{5\cdot 4}{2} - \left(\frac{2\cdot 1}{2}+\frac{2\cdot 1}{2}+\frac{1\cdot 0}{2}\right) = 8$$

It depends on the solution: if all tasks were assigned to different machines, it would be $|T|(|T|-1)/2 = 10$; it they were all assigned to the same one, it would be zero.

Considering neighbourhood $N_{\mathcal{T}_1}$, the effect of the possible moves is reported in the following table, where label "-" marks the transfers of a task to the same machine.

Transfer	$M1$	$M2$	$M3$
a	-	$\max(3,18,8) = 18$	$\max(3,9,17) = 17$
b	-	$\max(9,12,8) = 12$	$\max(9,9,11) = 11$
c	$\max(16,5,8) = 16$	-	$\max(12,5,12) = 12$
d	$\max(17,4,8) = 17$	-	$\max(12,4,13) = 13$
e	$\max(20,9,0) = 20$	$\max(12,17,0) = 17$	-

Notice that some transfers are obviously not improving and one could avoid to compute them when applying the steepest descent algorithm: all transfers to the machine with the largest working time are strictly worsenings; all the transfers from a machine with a working time smaller than the largest one are nonimproving. In the present case, only the 4 transfers from $M1$ (tasks a and b) to $M2$ or $M3$ are promising.

The global best strategy selects the best neighbour solution, that corresponds to transferring task b to machine $M3$, with $x' = \{(a,M1),(b,M3),(c,M2),(d,M2),(e,M3)\}$ and $f(x') = 11$.

The first best strategy scans the table by rows (lexicographic order with respect first to the tasks, then to the machines) and finds the same solution, that is the only improving one with respect to x.

Considering neighbourhood N_{S_1}, the effect of the possible moves is reported in the following table, which contains only swaps between tasks assigned to different machines and the first index preceding the second (for symmetry).

Swap	c	d	e
a	$\max(7,14,8) = 14$	$\max(8,13,8) = 13$	$\max(11,9,9) = 11$
b	$\max(13,8,8) = 13$	$\max(14,7,8) = 14$	$\max(17,9,3) = 17$
c	-	-	$\max(12,13,4) = 13$
d	-	-	$\max(12,12,5) = 12$

As above, some swaps are obviously not improving and one could avoid to compute them. In particular, swaps not involving the machine with the largest working time are non improving. Only the 6 swaps involving a and b are promising (and b is the task of minimum duration, so swapping it only worsens the objective function).

Since none of the neighbour solutions is strictly better than the current one, both the global-best and first-best strategy do not return any solution (or, equivalently, return the current solution, thus implying the termination of the algorithm).

8.6.4 Exercise 4

Given the instance of *Maximum Weighted Satisfiability Problem* (*Max-WSAT*) with

$$\text{CNF} = (x_1 \vee x_4 \vee \bar{x}_5) \wedge (x_2 \vee x_3 \vee x_5) \wedge (x_1 \vee x_2 \vee x_5) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_5)$$

$$w = [\,5\ 8\ 2\ 11\,]$$

how many solutions are contained in neighbourhoods $N_{\mathcal{F}_1}$ and $N_{\mathcal{F}_2}$ (i.e., flip the value of one or two variables) of solution $x = (0,0,0,0,0)$?

Perform a single iteration of steepest descent exploring neighbourhood $N_{\mathcal{F}_1}$ with the global-best strategy from solution x.

Perform a single iteration of steepest descent from solution x with the first-best strategy, visiting neighbourhood $N_{\mathcal{F}_1}$ in increasing index order with respect to the variables.

Do the same adopting neighbourhood $N_{\mathcal{F}_2}$; in the case of the first-best strategy, sort the neighbour solutions in lexicographic order with respect to the indices of the first and second variable flipped.

Solution Neighbourhood $N_{\mathcal{F}_1}$ includes all solutions obtained "flipping" one of the logical variables (replacing its value by the complementary one). All such moves are feasible, so that the required number is

$$\left|N_{\mathcal{F}_1}(x)\right| = |V| = 5$$

and it is independent from the solution x.

Neighbourhood $N_{\mathcal{F}_2}$ includes all solutions obtained flipping two of the logical variables[3]. All such moves are feasible, so that the required number is

$$\left|N_{\mathcal{F}_1}(x)\right| = \frac{|V|(|V|-1)}{2} = 10$$

and it is independent from the solution x.

Considering neighbourhood $N_{\mathcal{F}_1}$, the effect of the possible moves is reported in the following table: the clauses satisfied when flipping a variable are marked by a cross.

Single flip	Clauses (w_j)				
Variables	C_1 (5)	C_2 (8)	C_3 (2)	C_4 (11)	$f(x'$
x_1	×		×	×	18
x_2	×	×	×	×	26
x_3	×	×		×	24
x_4	×			×	16
x_5		×	×	×	21

Since the current solution x has a value equal to $f(x) = 5+11 = 16$, the global-best strategy selects the best neighbour solution, that corresponds to flipping variable x_2, obtaining $x' = (0,1,0,0,0)$ with $f(x') = 5+8+2+11 = 26$. This is also optimal, as all clauses are satisfied.

The first best strategy stops at the first improving solution, that is obtained flipping variable x_1, and is $x'' = (1,0,0,0,0)$ with $f(x'') = 5+2+11 = 18$. It is obtained much quicker, visiting a single neighbour solution instead of five.

Considering neighbourhood $N_{\mathcal{F}_2}$, the effect of the possible moves is reported in the following table: the clauses satisfied when flipping a pair of variables are marked by a cross.

[3]It is correct to replace "two" with "at most two", thus including all elements of $N_{\mathcal{F}_1}$. In practice, algorithm designers appreciate the possibility to tune the distance of the neighbour solutions from the current one, and often adopt the restricted definition we follow here. This simply generalises the remark on whether to include x or not in the neighbourhood.

Double flip Pairs	Clauses (w_j) C_1 (5)	C_2 (8)	C_3 (2)	C_4 (11)	$f(x'$
(x_1,x_2)	×	×	×	×	26
(x_1,x_3)	×	×	×		15
(x_1,x_4)	×		×	×	18
(x_1,x_5)	×	×	×	×	26
(x_2,x_3)	×	×	×	×	26
(x_2,x_4)	×	×	×	×	26
(x_2,x_5)		×	×	×	21
(x_3,x_4)	×	×		×	24
(x_3,x_5)		×	×	×	21
(x_4,x_5)	×	×	×	×	26

The global-best strategy selects the best neighbour solution. There are many equivalent ones. Assuming that they have been visited in lexicographic order (even if this is not specified), we can choose to flip variables x_1 and x_2, obtaining $x''' = (1,1,0,0,0)$ with $f(x''') = 5+8+2+11 = 26$, that is optimal.

The first best strategy stops at the first improving solution, that is the same obtained with the global-best strategy. Once again, the computational time required in this case is much smaller (though this is not guaranteed to hold for any solution x and for any instance of the problem).

8.6.5 Exercise 5

Given the instance of the *Capacitated Minimum Spanning Tree Problem*, with root vertex a, weight function $w_v = 1$ for all $v \in V \setminus \{a\}$ and $w_a = 0$, capacity $W = 2$ and cost function:

Cost	a	b	c	d	e	f
a	0	10	5	4	9	8
b	10	0	9	8	11	7
c	5	9	0	12	15	3
d	4	8	12	0	2	13
e	9	11	15	2	0	7
f	8	7	3	13	7	0

consider the starting solution $x = \{(a,b),(a,d),(a,f),(b,c),(d,e)\}$.

How many solutions are contained in neighbourhood $N_{\mathcal{H}_2}(x)$, that includes all solutions with a Hamming distance not larger than 2 from x? Does this number depend on x?

And what about neighbourhood $N_{\mathcal{S}_1}(x)$, that includes all solutions obtained swapping one edge inside the solution with one outside?

Perform one step of the steepest descent algorithm from solution x, exploring neighbourhood $N_{\mathcal{S}_1}$ with the global-best strategy.

Perform one step of the steepest descent algorithm from solution x, exploring neighbourhood $N_{\mathcal{S}_1}$ with the first-best strategy and the following visit order: consider the removed edges in lexicographic order with respect to the indices of the first and second extreme vertex; for each removed edge, consider the added edges once again in lexicographic order[4].

[4]From the computational point of view, it would be better to consider for each added edge the possible removals, but that is more confusing if done by hand, as $m-n+1 = 10$ edges can be added, only 5 removed, and it is easier to spot the best edge to reconnect the spanning tree than the best edge to remove to break a cycle. Of course, the result is the same, apart from the effect of visiting the neighbour solutions in a different order.

Solution Neighbourhood $N_{\mathcal{H}_2}(x)$ includes all solutions obtained adding or removing at most two edges with respect to x. Since the feasible solutions are all spanning trees, with exactly $n-1$ edges (where $n = |V|$ is the number of vertices), the only feasible solutions are obtained swapping an edge inside and with an edge outside the solution:

$$N_{\mathcal{H}_2}(x) = N_{\mathcal{S}_1}(x) \text{ for all } x \in X$$

Not all swaps generate feasible solutions, first because the removed edge must belong to the single cycle closed by the added edge (otherwise, the resulting subset is not a tree), second because the total weight of the branches appended to the root vertex must not exceed the capacity. Therefore, it is possible to give an *a priori* asymptotic estimate of the number of neighbour solutions:

$$N_{\mathcal{H}_2}(x) = N_{\mathcal{S}_1}(x) \in O(nm)$$

where m is the number of edges of the graph, and a more precise *a priori* overestimate:

$$N_{\mathcal{H}_2}(x) = N_{\mathcal{S}_1}(x) \leq (n-1)\,(m-(n-1)) = 50$$

which is correct when the capacity is very large and x has a single branch appended to the root vertex (so that every swap yields a feasible solution).

A picture is certainly useful to keep track of the exchanges (see Figure 8.27).

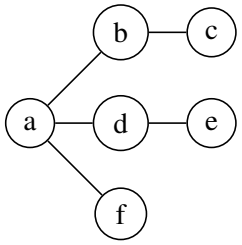


Figure 8.27: A *CMSTP* instance with a starting solution for the steepest descent method

The capacity constraint strongly limits the feasible swaps. In fact, the first and second branch cannot receive any additional subtree and the third one can receive at most one additional vertex, while new branches can be built *ad libitum*. Therefore, no edges between the first two branches can be added to solution x: only edges with at least one extreme in the third branch or coinciding with the root. In detail:

- removing (a,b) allows to add only (a,c) (with $\delta f = -10+5 = -5$), because the disconnected subtree is too weighty to be moved into another branch;
- removing (a,d) allows to add only (a,e) (with $\delta f = -4+9 = 5$), because the disconnected subtree is too weighty to be moved into another branch;
- removing (a,f) allows no feasible swap, because the disconnected subtree is too weighty to be moved into another branch;
- removing (b,c) allows to add (a,c) (with $\delta f = -9+5 = -4$) or (c,f) (with $\delta f = -9+3 = -6$), because the disconnected subtree can be moved only to a new branch or to the third one;

- removing (d,e) allows to add (a,e) (with $\delta f = -2+9 = 7$) or (f,e) (with $\delta f = -2+7 = 5$), because the disconnected subtree can be moved only to a new branch or to the third one.

Therefore, the global-best strategy suggests to swap (b,c) with (c,f), obtaining $x' = \{(a,b),(a,d),(a,f),(c,f),(d,e)\}$ with $f(x') = 10+4+8+3+2 = 27$, that is obviously equal to $f(x)+\delta f = 33-6$.

The first-best strategy stops at the first improving, that is also the first explored one: replace (a,b) with (a,c), obtaining $x'' = \{(a,c),(a,d),(a,f),(b,c),(d,e)\}$ with $f(x'') = 5+4+8+9+2 = 28$, that is obviously equal to $f(x)+\delta f = 33-5$.

8.6.6 Exercise 6

Given the instances of the *Weighted Vertex Cover Problem* represented in Figure 8.28, consider

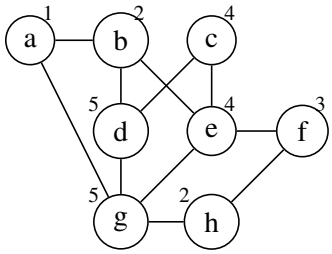


Figure 8.28: A *WVCP* instance with a starting solution for the steepest descent method

the starting solution $x = \{a,c,d,e,h\}$.

How many solutions are contained in neighbourhood $N_{\mathcal{A}_1}(x)$, that includes all solutions obtained adding a single vertex to x? And in neighbourhood $N_{\mathcal{D}_1}(x)$, that includes all solutions obtained deleting a single vertex from x? Do these numbers depend on x?

How many solutions are contained in neighbourhood $N_{\mathcal{H}_2}(x)$, that includes all solutions with a Hamming distance not larger than 2 from x? Does this number depend on x?

Perform one step of the steepest descent algorithm from solution x, exploring neighbourhood $N_{\mathcal{H}_2}(x)$ with the global-best strategy.

Perform one step of the steepest descent algorithm from solution x, exploring neighbourhood $N_{\mathcal{H}_2}(x)$ with the first-best strategy and flipping the bits of the incidence vector of x in lexicographic order (consider the single flips as a double flip with two identical indices, that is represent a as (a,a)).

Solution Neighbourhood $N_{\mathcal{A}_1}(x)$ includes all solutions obtained adding a single vertex to x. If x is feasible, any such solution is necessarily feasible. Therefore

$$|N_{\mathcal{A}_1}(x)| = |V \setminus x| = 3$$

that necessarily depends on x.

Neighbourhood $N_{\mathcal{D}_1}(x)$ includes all solutions obtained deleting a single vertex to x. Such solutions could be feasible or not, depending on whether x is redundant or not. Their number can be overstimated as:

$$|N_{\mathcal{D}_1}(x)| \leq |x| = 5$$

In this particular case, all vertices of x are strictly required to cover at least one of the edges, with the exception of vertex c, that is redundant and can be safely removed:

$$|N_{\mathcal{D}_1}(x)| = 1$$

This number is clearly dependent on x.

Neighbourhood $N_{\mathcal{H}_2}(x)$ includes all solutions with a Hamming distance not larger than 2 from x. Its size can be asymptotically estimated as

$$|N_{\mathcal{H}_2}(x)| \in O\left(n^2\right)$$

where $n = |V|$ is the number of vertices of the graph. A more precise overestimate is given by the number of vertices plus the number of pairs of vertices, whose corresponding values in the incidence vector can be reversed:

$$|N_{\mathcal{H}_2}(x)| \leq n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2} = 36$$

The precise number needs to consider the incidence relation between vertices and edges.

Now let us list the possible exchanges for solution $x = \{a,c,d,e,h\}$. First the ones that remove vertices from x:

- vertex a covers edges (a,b) and (a,g) and is essential for both: removing it requires to add both b and g, which is impossible in $N_{\mathcal{H}_2}(x)$;
- vertex c is the only redundant vertex: no other vertex can be removed at the same time, but any vertex can be safely added; hence, deleting c or swapping c with b, f and g is feasible;
- vertex d covers edges (b,d), (c,d) and (d,g) and is essential for the first and the last one: removing it requires to add both b and g, which is impossible in $N_{\mathcal{H}_2}(x)$;
- vertex e covers edges (b,e), (c,e), (e,f) and (e,g) and is essential for the first, the third and the last one: removing it requires to add b, f and g, which is impossible in $N_{\mathcal{H}_2}(x)$;
- vertex h covers edges (f,h) and (g,h) and is essential for both: removing it requires to add f and g, which is impossible in $N_{\mathcal{H}_2}(x)$.

Then, let us consider the exchanges that only add vertices (those adding one and removing one have been considered above): they are $(n-|x|)(n-|x|+1)/2 = 6$ (three add a single vertex and three add a pair of vertices); they are all feasible and all worsening. In summary, we have a single deletion (c), three swaps (c with b, f and g), three single additions (b, f and g) and three double additions (b and f, b and g, f and g):

$$|N_{\mathcal{H}_2}(x)|1+3+3+3 = 10$$

Out of these exchanges, only the single deletion and the three swaps could improve the value of the objective. Since the three swaps involve the same vertex affected by the deletion, they are dominated: deleting vertex c is the best possible move. The global-best strategy, therefore, returns $x' = x \setminus \{c\} = \{a,d,e,h\}$, with $f(x') = 1+5+4+2 = 12$, that is obviously $f(x) - w_c$.

The first-best strategy takes into account the 10 moves in lexicographic order:

$$(b,b)\ (b,c)\ (b,f)\ (b,g)\ (c,c)\ (c,f)\ (c,g)\ (f,f)\ (f,g)\ (g,g)$$

As already observed, the only possible improvements can be obtained from the single deletion and the three swaps, so the list can be reduced to the following table:

	(b,c)	(c,c)	(c,f)	(c,g)
δf	$+2-4=-2$	-4	$+3-4=-1$	$+5-4=1$

that suggests to swap c with b, obtaining $x'' = x \setminus \{c\} \cup \{b\} = \{a,b,d,e,h\}$, with $f(x') = 1+2+5+4+2 = 14$, that is obviously $f(x) - w_c + w_b$.

8.6.7 Exercise 7

Given the following instance of the *Bin Packing problem* (*BPP*):

Items	a	b	c	d	e	f	g
Volumes	6	4	5	10	3	11	8

Bin capacity
20

consider the starting solution x that puts objects a, c and e in the first container, objects b and f in the second, objects d and g in the third one.

Build the auxiliary graph associated to permutation (a,b,c,d,e,f,g) in the *order-first split-second* method.

Solution The auxiliary graph used in the order-first split-second method has a node for every object, plus a fictitious node 0, therefore $n' = n+1 = 8$ nodes in this case. It is an acyclic graph, with arcs going only from a node to node that follow it in the given permutation. There are therefore at most $n'(n'-1)/2 = 28$ arcs. Each arc represents the subset of objects between the one following the tail of the arc to the head of the arc. The arcs corresponding to unfeasible subsets are removed. In this case, the feasibility of the subsets depends on the total volume of the objects. Therefore, the arcs can be generated lexicographically: for each tail node, we consider all possible head nodes increasing step by step the volume of the current subset, and stopping as soon as the capacity is violated.

This generated the following arcs:

- from node 0:
  - to node a, the total volume is feasible ($6 \leq 20$): arc $(0,a)$ exists;
  - to node b, the total volume is feasible ($6+4 \leq 20$): arc $(0,b)$ exists;
  - to node c, the total volume is feasible ($6+4+5 \leq 20$): arc $(0,c)$ exists;
  - to node d, the total volume is unfeasible ($6+4+5+10 > 20$): move to the next tail node;
- from node a:
  - to node b, the total volume is feasible ($4 \leq 20$): arc (a,b) exists;
  - to node c, the total volume is feasible ($4+5 \leq 20$): arc (a,c) exists;
  - to node d, the total volume is feasible ($4+5+10 \leq 20$): arc (a,d) exists;
  - to node e, the total volume is unfeasible ($4+5+10+3 > 20$): move to the next tail node;
- from node b:
  - to node c, the total volume is feasible ($5 \leq 20$): arc (b,c) exists;

  - to node d, the total volume is feasible ($5+10 \leq 20$): arc (b,d) exists;
  - to node e, the total volume is feasible ($5+10+3 \leq 20$): arc (b,e) exists;
  - to node f, the total volume is unfeasible ($5+10+3+11 > 20$): move to the next tail node;

- from node c:
  - to node d, the total volume is feasible ($10 \leq 20$): arc (c,d) exists;
  - to node e, the total volume is feasible ($10+3 \leq 20$): arc (c,e) exists;
  - to node f, the total volume is unfeasible ($10+3+11 > 20$): move to the next tail node;

- from node d:
  - to node e, the total volume is feasible ($3 \leq 20$): arc (d,e) exists;
  - to node f, the total volume is feasible ($3+11 \leq 20$): arc (d,f) exists;
  - to node g, the total volume is unfeasible ($3+11+8 > 20$): move to the next tail node;

- from node e:
  - to node f, the total volume is feasible ($11 \leq 20$): arc (e,f) exists;
  - to node g, the total volume is unfeasible ($11+8 > 20$): arc (e,g) exists;

- from node f:
  - to node g, the total volume is feasible ($8 \leq 20$): arc (f,g) exists;

Overall, there are 16 arcs. Notice that the 7 arcs connecting a node to the following one could be given for granted, given that they represent singletons and the volume of a single object respects the capacity, if the instance is feasible.

CHAPTER 9

Exchange metaheuristics

9.1 Introduction to exchange metaheuristics

The basic scheme of exchange heuristic, the *steepest descent* algorithm, is characterised by the starting solution, the definition of the neighbourhood and the termination condition. This scheme finds only local optima. There are two possible ways to extend this scheme without abandoning it completely. The first one is to repeat the search, starting from another point and applying again the scheme of steepest descent. The problem here is to avoid falling into the same local optimum. The second way is to prolong the search, accepting new solutions in the neighbourhood of the local optimum. This requires to allow worsening the objective function, managing the strong risk that the algorithm "falls" back into the original local optimum, immediately or after a small number of steps, and therefore starts cycling.

In Section 7.1, when introducing constructive metaheuristics, we proposed a single way to extend the basic scheme, that was repeating the search. Prolonging the search beyond its natural termination condition is impossible for constructive metaheuristics, because they are baesd on the idea of augmenting a current subset of the ground set, and this is intrinsically limited by reaching a size after which no feasible solution exists (in the worst case, by the size of the ground set, $|B|$. This is no longer true for exchange metaheuristics, which add, but also remove elements from the current subset.

In any case, the mechanisms used to design exchange metaheuristics are again randomisation and memory. They can be applied, possibly combined, to the elements of the basic exchange algorithm, that is, the starting solution $x^{(0)}$ (this will be the case of multi-start, Iterated Local Search and Variable Neighbourhood Search), the neighbourhood (Variable Neighbourhood Descent), the selection criterion $\varphi(x,A,D)$ (Dynamic Local Search) and the selection rule (Simulated Annealing and Tabu Search).

9.1.1 Termination condition

Before discussing the techniques, it is necessary to address the issue that repeating or prolonging the search removes the intrinsic termination condition of the algorithm. This has already been observed with respect to the constructive metaheuristics.

A new termination condition must be introduced. There are two general ways to define one. It is possible to use an "absolute" termination condition, fixed *a priori* at the beginning of the search:

- a given total number of exploration of the neighbourhood (i.e., loops in the basic scheme) or a given total number of repetitions of the local search, that is how many times the basic

overall scheme is applied from a starting solutions to a final local optimum;

- a given total execution time;
- a given target value of the objective;

or a "relative" termination condition, which is *triggered* by some event that happens during the search, so it is not perfectly controlled by the user of the algorithm:

- a given number of explorations of the neighbourhood or repetitions after the last improvement of f^*: for example, after 1 000 iterations with no improvements, the algorithm might decide that better solutions will not be found in a reasonable time, and therefore stop;
- a given execution time after the last improvement;
- a given minimum value of the ratio between the improvement of the objective and a given number of explorations or execution time.

Absolute conditions are commonly adopted in algorithmic comparisons, because they guarantee fairness between the competing algorithms. They are also used when a given resource is available and employed to get the best possible result. For example, the total execution time is used when the time is limited, but it makes little sense to save it. A problem of fixing the total execution time in algorithmic comparisons is that running several times the same algorithm with the same limit could yield different results, due to small variations in the speed of the machine: a machine could perform 1 002 iterations in the first run, 999 in the second, and so on, yielding different results if the best one is found in one of the last iterations. This is avoided when the number of iterations is fixed, even for randomised algorithms (if the random seed is also fixed). Fixing a target value implies the risk that the algorithm never finds that value, because it is better than the best value the algorithm is able to find (e.g., than the optimum). Relative conditions typically aim to find a compromise between efficiency and effectiveness, between computational effort and the quality of the result.

9.2 Repeating the search

How to generate different starting solutions in order to obtain different results when repeating the search? There are three possible main strategies: the first is to generate them at random, the second one is to obtain them with different constructive heuristics, the third one is to modify solutions generated by the exchange algorithm using some neighbourhood.

9.2.1 Random generation

The use of random generation has some advantages. First of all, it is conceptually simple. Second, it is usually quick. This mainly depends on how easy it is to guarantee feasibility, which strongly depends on the single problem: if it is hard, most of the subsets generated will be infeasible, leading to a waste of time. Third, the user could have control on the probability distribution. Some elements could be favoured with respect to other ones. For example, favouring low cost elements is somehow similar to constructive metaheuristics with a random component. Favouring elements that during the past repetitions were frequent intensifies the search, whereas favouring the rare ones diversifies the search. Finally, random generation guarantees asymptotic convergence to the optimum in infinite time, because every combination of elements has a nonzero probability of being generated.

Of course, there are some disadvantages, as well. The starting solution generated is typically of very scarse quality (this does not necessarily mean that the final solution will be bad). Reaching a good solution from a very bad one typically requires many iteations, and therefore a long time (depending also on the complexity of the neighbourhood exploration). The same holds for feasibility if the decision problem is $\mathcal{NP}$-complete, as the *TSP* on general graphs.

9.2.2 Multi-start methods

The multi-start methods have already been introduced in Section 7.1.1 as constructive metaheuristics, designing and applying different constructive heuristics and returning the best result found in all applications. Of course, each solution generated by a constructive heuristic can be improved with the use of an exchange heuristic. The advantage of these method is that the method used to build the starting solutions can be chosen *ad libitum*, but there are several disadvantages. The first is that choosing the method gives little control on the solution obtained: even applying similar ideas does not guarantee to generate similar solution. A second big problem is that a multistart algorithm cannot go on proceed indefinitely: its number of repetitions is given by the number of constructive heuristics designed. For times longer than the maximum total time of all heuristics, the algorithm does not improve further the solution. The termination condition is intrinsic. They also requires a high design effort, as several different algorithms must be designed and, furthermore, they usually give no guarantee of convergence, not even in infinite time. To be strict, multistart heuristics are not real metaheuristics, unless they have a component that is randomised or based on memory. Indeed, multistart methods tend to be replaced by combining a constructive metaheuristic with an exchange heuristic (the standard *GRASP* or *Ant System* approach always includes an exchange phase, which was not discussed in Sections 7.4 and 7.5 to focus on the relevant aspects of the two methods).

What is the difference between a constructive metaheuristic with an exchange procedure and an exchange metaheuristics with a randomised initialisation procedure? The difference is small and mainly conventional. As we only presented the constructive phase in the previous chapter, so we consider only the exchange phase in this chapter. All methods can be hybridised. If a taxonomy is required, one can consider an algorithm as a constructive metaheuristic, if its constructive phase is highly refined with respect to the exchange one, as an exchange metaheuristic if its exchange phase is highly refined with respect to the constructive one.

Influence of the starting solution

The choice between a random procedure or a constructive one to initialise the search is based on the length of the influence exerted by the starting solution on the current one. A very good exchange heuristic should be able to cancel very quickly this influence, so that while the value of $x^{(0)}$ can be strongly different, the value of $x^{(t)}$ tend to coincide as iterations elapse. A bad starting solution could take some more steps, but in the end it will reach the same values obtained from a good one.

Figure 9.1 shows a case in which the difference between a bad and a good initialisation is progressively reduced, but is still perceivable after one hour or computation. If the good constructive procedure takes a short time (even if longer than the random one), it anyway pays in the subsequent process. In other cases, the two diagrams could converge quickly, and even reverse (a good initialisation is not guarantee of a good final result).

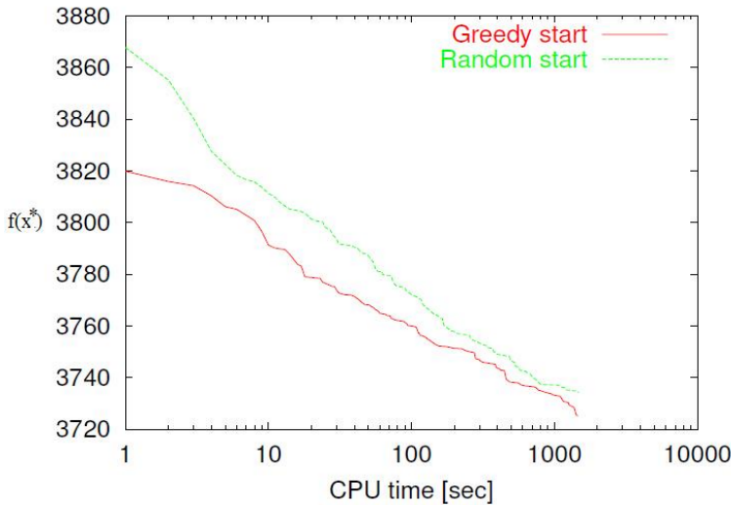


Figure 9.1: Greedy vs random start for a bad exchange heuristic.

9.2.3 Exploiting previous solutions

A third way to restart the search is to exploit the information gathered by previously visited solutions, saving *reference solutions* and generating new starting ones based on them. Typical reference solutions are the local optima found previously.

These methods give the user a good deal of control on the starting solution: the user can choose the reference solution, how to modify it and how much, so that the search can be intensified or diversified *ad libitum*. Typically, if the modification is not too large and the reference solution is good, the starting solution is also very good!

The method is simple conceptually, as the user does not have to design new algorithms, but only a modification of the starting solution: such modifications are often easy, as they can be based on operations like the ones used to define the neighbourhood of the exchange heuristic (not the same, but similar ones). Finally, under suitable conditions, these methods guarantee asymptotic convergence to the optimum in infinite time: the user should be able to control the distance of the new solution from a given one so as to potentially reach any feasible solution.

Two methods are here proposed: the *Iterated Local Search* and the *Variable Neighbourhood Search*. Both have a rich literature and there are discussions on the relative differences. In these notes, we adopt the opinion that they are the same method with minor differences.

9.2.4 Iterated Local Search

The **Iterated Local Search** (*ILS*) iteratively the classical steepest descent algorithm (from which "local search"), generating the new starting solution every time with a *perturbation procedure* that operates on a reference solution that is a locally optimal solution returned by steepest descent in the previous iterations. An *acceptance condition* decides whether the reference solution changes, becoming the last locally optimal solution found, or remains the same. Of course, a termination condition must be given, as already explained.

Algorithm 14 starts from a feasible solution $x^{(0)}$ and improves it with steepest descent, as in the basic exchange heuristics. Instead of terminating after steepest descent, the result is saved and, as long as the termination condition is not satisfied, a *perturbation* is operated on the current solution x, obtaining a new solution x', which is in turn improved again, finding a new local optimum. This is compared with the overall optimum x^*, in order to update it, but also

Algorithm 14 Iterated Local Search Pseudocode

```
1:  procedure ILS(ℓ, x^(0))
2:      x := SteepestDescent(x^(0))
3:      x* := x;
4:      for l := 1 to ℓ do
5:          x' := Perturbate(x)
6:          x' := SteepestDescent(x')
7:          if Accept(x', x*) then
8:              x := x'
9:          end if
10:         if f(x') < f(x*) then
11:             x* := x'
12:         end if
13:     end for
14:     return (x*, f(x*))
15: end procedure
```

to decide if it is accepted or not as the new reference solution. If it is accepted, the reference solution becomes x'.

The basic idea is that the exchange heuristic quickly explores an attraction basin, terminating into a local optimum. Then, the perturbation procedur moves the starting solution to another attraction basin. The acceptance condition evaluates if the most promising starting point for the following perturbation is the new local optimum or the previous one (see Figure 9.2).

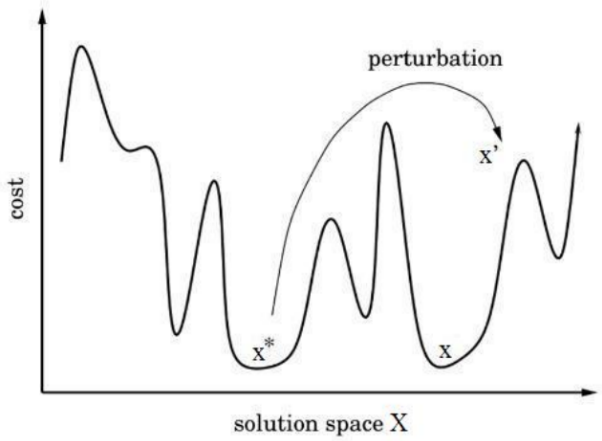


Figure 9.2: Schematic representation of the Iterated Local Search algorithm.

Example: *ILS* for the *TSP* A classical example is the application of *ILS* to the *TSP*. The perturbation procedure in this case uses as exchange heuristic the steepest descent with neighbourhood N_{R_2} or N_{R_3}. The perturbation procedure is based on a *double-bridge* move, represented in Figure 9.3. It consists in removing 4 arcs from the current solution and replacing each one with another arc so that the direction of the remaining subpaths is not reversed (which makes it simple to use also for the asymmetric *TSP*). This is a particular kind of 4-opt exchange. It generates, therefore, a solution in a neighbourhood of x, but different from the ones explored in N_{R_2} or N_{R_3}.

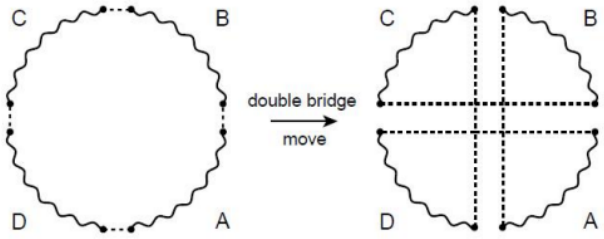


Figure 9.3: Representation of a double-bridge move.

The new solution can be better or worse than the current one (in general, it is worse). The relevant point is that it is feasible and out of the original neighbourhood. This makes it a reasonable starting point for a new exploration. The computation then goes on with the steepest descent algorithm on the original neighbourhood until it finds a local optimum. This could be the same as before (an unlucky case), a worse one or a better one. This particular application of *ILS* adopts an acceptance condition that requires the new solution to be better than the best known one:

$$f(x') < f(x^*)$$

If this is true, the new local optimum is used for the following perturbation; otherwise, the old local optimum is used once again to generate a different random starting point.

The perturbation procedure Let $\mathcal{O}$ be the operation set that defines the neighbourhood $N_{\mathcal{O}}$. The perturbation procedure performs a random operation o chosen in a different set ($o \in \mathcal{O}' \nsubseteq \mathcal{O}$), to avoid that the exchange heuristic drives the solution x' back to the starting local optimum x. Two typical definitions of $\mathcal{O}'$ are:

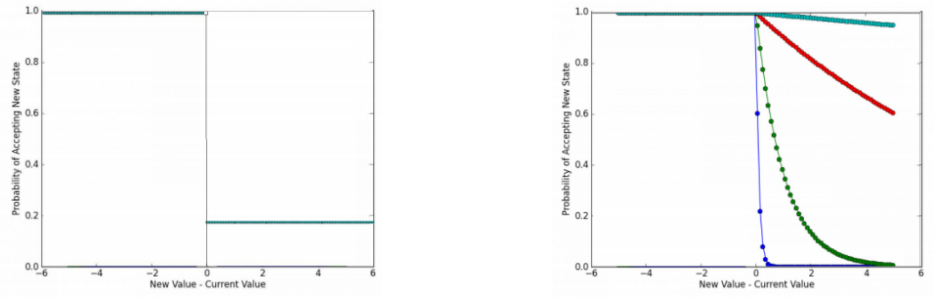
- sequences of $k > 1$ operations of $\mathcal{O}$ if generating a random sequence is cheap;
- conceptually different operations, for example vertex exchanges instead of arc exchanges.

The main difficulty of *ILS* is in tuning the perturbation: if it is too strong, it turns the search into a random restart, but if it is too weak, it guides the search back to the starting optimum, wasting time and possibly losing the asymptotic convergence. Ideally, one would like to tune the perturbation so that is is able to enter, but also to get out of any basin of attraction.

9.2.4.0.1 The acceptance condition The acceptance condition of the previous example is rather reasonable: when the algorithm finds a new best known solution, it is wise to move there and explore its surroudings. If the new local optimum is worse than the given one, should the algorithm accept it or not? This depends on how the balance between intensification and diversification should be tuned: accepting only improving solutions favours intensification, while accepting any new solution favours diversification. Between these two options, there are lot of situations, which can be defined on the basis of $\delta f = f(x') - f(x)$. The typical scheme adopted by *ILS* is randomised: if $\delta f < 0$, the algorithm always accepts x'; otherwise, it accepts x' with a certain probability $\pi(\delta f)$, that is a nonincreasing function of the worsening δf.

There are, of course, many distribution schemes. The simplest ones are using a fixed probability $\pi(\delta f) = \bar{\pi}(\delta f) \in (0,1)$ for each $\delta f \geq 0$ (see Figure 9.4 (a)), and a monotonically decreasing probability with $\pi(0) = 1$ and $\lim_{\delta f \to +\infty} \pi(\delta f) = 0$. Among the monotonically decreasing functions, the most common are the exponential and the linear one, with a single parameter that measures the slope, as represented in Figure 9.4 (b).

A different possibility is to introduce a memory mechanism, accepting x' if many iterations have passed since the last improvement of x^*, in order to diversify the search.



(a) Uniform probability. (b) Monotonically decreasing probability.

Figure 9.4: Different probabilistic schemes for the acceptance condition.

9.2.4.1 Variable Neighbourhood Search

Variable Neighbourhood Search (*VNS*) was proposed by Hansen and Mladenović in 1997. It is based on the same concept as *ILS*, but it is, in a sense, complementary. *ILS* perturbates each local optimum found with a simple move, and manages the acceptance of the new local optimum with a sophisticated probabilistic mechanism. *VNS* uses a refined perturbation mechanism, but has a very simple acceptance condition that accepts only improving new local optima.

In fact, *VNS* adopts an **adaptive perturbation mechanism**. It requires a hierarchy of neighbourhoods, that is a family of parametrically defined neighbourhoods with an increasing size s

$$N_1 \subset N_2 \subset \cdots \subset N_s \subset \cdots \subset N_{s_{\max}}$$

For the sake of simplicity, typically one uses the parameterised neighbourhoods N_{H_s}, based on the Hamming distance between subsets, or N_{O_s}, based on the sequences of operations from a basic set $\mathcal{O}$. This technique is called *variable neighbourhood* because the neighbourhood used to randomly extract $x^{(0)}$ to reinitialise the basic exchange heuristic varies along the hierarchy based on the results of the exchange heuristic. If a worse local optimum is found, the algorithm uses a slightly larger neighbourhood to generate a starting solution slightly farther from x^*, diversifying the search. If the local optimum is better, the algorithm uses the smallest neighbourhood in the hierarchy, to generate a starting solution very close to x^*, intensifying the search.

The method has three parameters:

- $s_{\min}$ identifies the smallest neighbourhood to generate new solutions,
- $s_{\max}$ identifies the largest neighbourhood to generate new solutions,
- δs identifies the increase of k between two subsequent attempts.

The basic exchange heuristic adopts the smallest neighbourhood in the hierarchy, for the sake of efficiency: N_e with $e \leq s_{\min}$.

Parameter tuning The value $s_{\min}$, the minimum neighbourhood in which new starting solutions are generated, should be not too small, as that would forbid or make it harder to exit the current attraction basin. It must not be too large, in order to avoid "jumping over" promising basins of attraction. In general one sets $s_{\min} = 1$ and increases it if the experiments show that it is profitable (avoiding useless iterations of the original basin of attraction).

The value $s_{\max}$ must be large enough to reach any useful attraction basin, but small enough to avoid reaching useless regions of the solution space. The diameter of the search graph is a value of $s_{\max}$ that allows to reach any solution from any other solution.

Algorithm 15 Variable Neighbourhood Search Pseudocode

```
1: procedure VNS(ℓ, x^(0))
2:     x := SteepestDescent(x^(0))
3:     x* := x;
4:     s := s_min;
5:     for l := 1 to ℓ do
6:         x' := Shaking(x*, k)
7:         x' := SteepestDescent(x')
8:         if f(x') < f(x*) then
9:             x* := x'
10:            s := s_min
11:        else
12:            s := s + δs
13:        end if
14:        if s > s_max then
15:            s = s_min
16:        end if
17:    end for
18:    return (x*, f(x*))
19: end procedure
```

The value of δs must be large enough to bridge the gap between $s_{\min}$ and $s_{\max}$ in a reasonable number of steps, but small enough to allow every reasonable value of s. Typically, $\delta s = 1$.

Skewed *VNS* We have seen that accepting also worsening local optima can improve diversification when discussing the *ILS* approach. This was noticed also by the proposers of *VNS*, who introduced a variant known as *skewed* VNS. The basic acceptance criterion is generalised as

$$f(x') < f(x^*) + \alpha \cdot d_H(x', x^*)$$

where $d_H(x', x^*)$ is the Hamming distance between x' and x^* and $\alpha \geq 0$ is a suitable parameter. With $\alpha = 0$, we have the classical *VNS*, wherease $\alpha > 0$ allows to accept worsening solutions, provided that they are at a certain distance from the original solution: the farther the better. With $\alpha \gg 0$, the algorithm tends to accept any solution. The value of α must be tuned taking into account also the fact that the two terms (Hamming distance and variation of the objective) are measured in different units.

Of course, the random strategies used by the *ILS* can also be adopted: the two approach can be trivially merged using both the sophisticated perturbation of *VNS* and the sophisticated acceptance condition of *ILS*.

9.3 Extending the local search

Instead of repeating the search, it is possible to prolong it after a local optimum has been found, modifying the choice mechanism. As long as, the basic selection rule (choose the neighbour solution which minimises the selection criterium in the neighbourhood) remains the same, this can be achieved in two ways:

- the *Variable Neighbourhood Descent* (*VND*) **changes the neighbourhood** N; this guarantees an evolution with no cycles, since the objective strictly improves at each step; the method terminates in a finite time, when the current solution is a local optimum for all neighbourhoods considered;

- the *Dynamic Local Search* (*DLS*) **changes the objective function** f: the new solution $\tilde{x}$ is better than x for the new objective function, but possibly worse for the old one; since the objective changes over time, the method can proceed indefinitely, but also be "trapped" in loops.

9.3.1 Variable neighbourhood descent

Variable Neighbourhood Descent is strictly related to Variable Neighbourhood Search. In fact, they were proposed in the same papers by Hansen and Mladenović. They are both based on the idea of exploiting several neighbourhoods. The difference is that, the neighbourhoods are used to apply the classical steepest descent algorithm, instead of to restart the search.

VND requires a sequence of neighbourhoods $N_1, \cdots, N_{s_{\max}}$ (not necessarily a hierarchy). It starts applying the classical steepest descent algorithm with the first neighbourhood, finding a local optimum with respect to it. Then, the algorithm moves to the next neighbourhood in the sequence and checks if the current solution can be improved by steepest descent. A flag $flag_s$ marks the indices s of the neighbourhoods with respect to which the current solution is locally optimal. The algorithm stops when the current solution is locally optimal for all neighbourhoods.

The fundamental difference between the *VND* and *VNS* is that the *VND* explores systematically the neighbourhoods, instead of using them just to extract random solutions. Therefore, at each step the current solution is the best known one. Moreover, all neighbourhoods used must be fairly small (otherwise, they could not be visited), and it is not required that they form a hierarchy of increasing size and distance. Finally, when a local optimum for every N_s has been reached, the algorithm terminates, so *VND* is deterministic. Randomisation could be used in the selection of the next neighbourhood. Memory mainly concerns the neighbourhoods already used, not the selection of solutions. As these are minor points, *VND* can also be seen as not strictly a metaheuristic.

Neighbourhood scan strategies in the *VND*

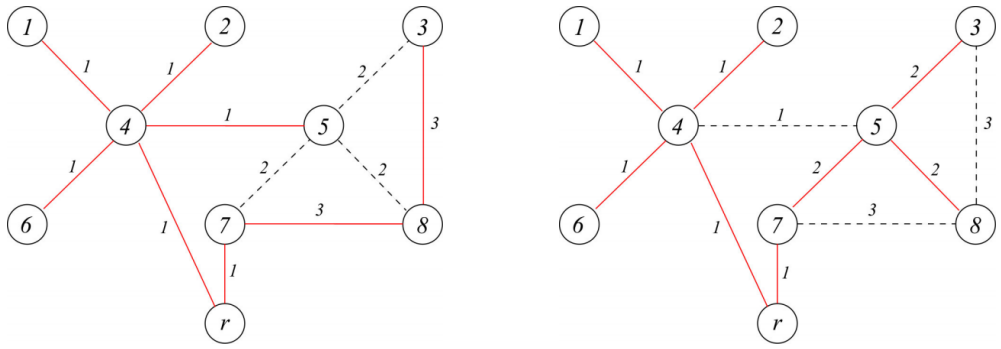
There are two main classes of *VND* "implementations": methods with heterogeneous neighbourhoods and methods with hierarchical neighbourhoods. The former are based on the definition of topologically different neighbourhood. For example, when solving the *CMSTP*, one can transfer vertices between subtrees or swap edges. These two neighbourhoods are intrinsically different and in general a local optimum for one is not a local optimum for the other. The current neighbourhood can alternate between them. If more neighbourhood of this kind are used, s can periodically scan them, or be extracted at random.

The methods with hierarchical neighbourhood have larger neighbourhoods that typically include the smaller ones. In this case, one generally exploits the small and fast neighbourhoods, resorting to the large and slow ones only to get out of the local optima. Differently from Algorithm 16, the steepest descent is applied for small values of s, but with large values the exchange heuristic can be terminated in advance, performing only some moves to guarantee that the algorithm gets far away from the original local optimum, and then the search goes back to the first and smallest neighbourhood. Therefore the neighbourhoods are not scanned periodically,

Algorithm 16 Variable Neighbourhood Descent Pseudocode

```
1:  procedure VND(x^(0))
2:      ∀s flag_s := false;
3:      x̄ := x^(0);
4:      x* := x^(0);
5:      s := 1;
6:      while ∃s : flag_k = false do
7:          x̄ := SteepestDescent(x̄, N_k)
8:          if f(x̄) < f(x*) then
9:              x* := x̄
10:             ∀s' ≠ s flag_s' := false
11:         else
12:             flag_s := true
13:         end if
14:         s := Update(s)
15:     end while
16:     return (x*, f(x*))
17: end procedure
```

but focusing on the first elements of the hierarchy. When the algorithm finds a local optimum for a neighbourhood N_s, s must increase k and the flags for the previous values are all marked. When the maximum value $s_{\max}$ is reached, it makes no sense to go back to $s_{\min}$: the algorithm can terminate directly.



(a) Solution 1. (b) Solution 2.

Figure 9.5: Two different solutions for the CMSTP

Example: the CMSTP Consider the instance of *CMSTP* reported in Figure 9.5: there are $n = 9$ vertices with uniform weights $w_v = 1$, the capacity is $W = 5$; the graph is complete and the edge not depicted for the sake of clarity have cost $c_e \gg 3$. Consider the neighbourhood N_{S_1} for the solution in Figure 9.5 (a): no edge in the right branch can be deleted because the disconnected subtree cannot be appended to the left branch (there is no residual capacity) and to the root (the required edge has a large cost). On the other hand, deleting any edge in the left branch increases the total cost. So, the solution is a local optimum for N_{S_1}.

Consider, now, the neighbourhood N_{T_1} (transfers of single vertices between branches): moving vertex 5 to the right branch yields the solution represented in Figure 9.5 (b), that improves the objective function.

9.3.2 Dynamic local search

Dynamic local search (*DLS*), also known as *Guided Local Search* (*GLS*) is, in a sense, complementary to *VND*, as the object to be modified is not the neighbourhood, but the objective function. This is typical of problems in which the objective function is not very informative, because it is "flat", that is, it has wide *plateaus*.

The idea is to introduce an additional penalty function $w : X \to \mathbb{N}$, defined on the solutions, usually with integer values, and build an auxiliary function $\tilde{f}(f(x), w(x))$ that combines the original objective value $f(x)$ and the penalty value $w(x)$. Then, the steepest descent algorithm is applied to the combined function $\tilde{f}$, instead of the objective. This means that the steepest descent algorithm may take some step in a direction where f worsens, as long as $\tilde{f}$ improves. Furthermore, at each iteration the penalty w is updated based on the results obtained, and consequently also $\tilde{f}$ changes. These updates allow the search to go on, possibly for an indefinite time, but they also could get stuck in a cycle, repeatedly visiting the same solutions: this situation is obviously to be avoided.

Algorithm 17 Dynamic Local Search Pseudocode

```
1:  procedure DLS(I, x^(0))
2:      x := SteepestDescent(x^(0))
3:      w := StartingPenalty(I)
4:      x* := x;
5:      while Stop() = false do
6:          (x̄, x_f) := SteepestDescent(x̄, f, w)
7:          if f(x_f) < f(x*) then
8:              x* := x_f
9:          end if
10:         w := UpdatePenalty(w, x̄, x*)
11:     end while
12:     return (x*, f(x*))
13: end procedure
```

The general idea, represented in Algorithm 17, is quite abstract, as it can be developed and adapted in many different ways to different problems. The steepest descent algorithm optimises $\tilde{f}$ and returns two solutions: the current solution $\bar{x}$ is locally optimal with respect to $\tilde{f}$ and is used to update w, while x_f is the best solution found with respect to f.

Penalty functions

Two classical families of penalty functions are the **additive penalty**:

$$\tilde{f}(x) = f(x) + \sum_{i \in x} w_i$$

which is defined on the ground set and simply summed on the elements of the solution, and the **multiplicative penalty**

$$\tilde{f}(x) = \sum_j w_j \phi_j(x)$$

which is associated to "components" of the solution, and requires $f(x) = \sum_j \phi(x)$, so that the penalties are multiplied by the components of the objective. We have seen many cases of problems in which the solution is formed by structures, and the objective is a sum of values associated with these structures. All problems with additive objective functions fall in this category,

but many other ones also do. The objective of the *BPP* (number of containers) is a sum of unitary terms associated with containers, the objective of *Max-SAT* is a sum of weights associated with satisfied clauses. Of course, there are also cases in which this does not hold: the objective of the *PMSP* is the maximum of several values associated to machines, instead of the sum (and yet, one could think of penalties associated to the machines and combined somehow with the objective).

The penalty can be modified with simple deterministic rules or "noisy" randomised perturbations of the costs, aiming to avoid cycles, or based on memory, to favour the most frequent elements for intensification or the less frequent ones for diversification.

The update can be performed at each iteration (neighbourhood exploration), or when a local optimum for $\tilde{f}$ is reached or when the best known solution x^* remains unchanged for a certain amount of time. The number of possible schemes is huge.

Additive example: *DLS* for the MCP An example of additive penalty is given by the *MCP*, that is the search for a subset of vertices of maximum cardinality that are reciprocally adjacent in an undirected graph.

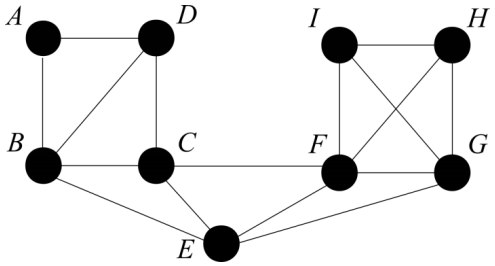


Figure 9.6: MCP instance.

Consider the instance reported in Figure 9.6. Vertices C and E form a clique, since they are reciprocally connected, but not a maximal or a maximum cardinality one. A natural definition of neighbourhood for this problem could be N_{A_1}, that is the neighbourhood built adding a single vertex to the current subset. Most of the time, the newly added vertex is not adjacent to all of the original ones, and therefore the resulting subset is not a feasible solution: they neighbourhood is often empty. Moreover, if there are feasible additions, the objective function is the same for all of them (e. g., add C or F to $x = \{E\}$: the value of the result is 2 in both cases). A flat objective is typical of this neighbourhood.

Wide *plateaus* are also typical of the single-swap neighbourhood N_{S_1}: given $x = \{B,C,D\}$, one can remove C and add A, or remove D and add E. Both moves give the same final value of the objective function. Moreover, all feasible solutions obtained have exactly the same value of the original solution, and therefore are non improving. How to guide the search in neighbourhoods such as these ones?

In the following example, we use as a basic exchange heuristic a *VND* using the N_{A_1} and N_{S_1} neighbourhoods, that is trying to add vertices as long as possible, switching to swaps when this is no longer possible, and going back to additions as soon as it becomes again possible. In order to design a *DLS* approach, we change the objective function, associating with each vertex i a penalty w_i, that is initially set equal to 0. Since in both neighbourhoods the objective function is completely flat, the steepest ascent heuristic will simply select the neighbour solution with minimum penalty. When the exploration of N_{S_1} terminates, the penalty is updated, increasing by 1 its value on the vertices of the current clique. To avoid an indefinite increase, after a given number of neighbourhood explorations all the nonzero penalties decrease by 1. The rationale

of the method is to try and expel the internal vertices, in particular the oldest internal vertices, in order to diversify the search, exploiting a form of memory.

Given the starting solution $x^{(0)} = \{B,C,D\}$, with $w = [0\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0]$, the algorithm performs the following steps:

1. N_{A_1} is empty, so we must explore N_{S_1}. There, we have $w(\{B,C,E\}) = w(\{A,B,D\}) = 2$ but $\{A,B,D\}$ wins lexicographically, so $x^{(1)} = \{A,B,D\}$ with $w = [1\ 2\ 1\ 2\ 0\ 0\ 0\ 0\ 0]$.
2. $x^{(2)} = \{B,C,D\}$ with $w = [1\ 3\ 1\ 3\ 0\ 0\ 0\ 0\ 0]$ is the only neighbour in N_{S_1} (N_{A_1} is empty.)
3. N_{A_1} is empty. $w(\{B,C,E\}) = 5 < 7 = w(\{A,B,D\})$, so $x^{(3)} = \{B,C,E\}$ with $w = [1\ 4\ 3\ 3\ 1\ 0\ 0\ 0\ 0]$
4. N_{A_1} is empty. $w(\{C,E,F\}) = 4 < 10 = w(\{B,C,D\})$, so $x^{(4)} = \{C,E,F\}$ with $w = [1\ 4\ 4\ 3\ 2\ 1\ 0\ 0\ 0]$
5. N_{A_1} is empty. $w(\{E,F,G\}) = 3 < 11 = w(\{B,C,E\})$, so $x^{(5)} = \{E,F,G\}$ with $w = [1\ 4\ 3\ 3\ 1\ 0\ 0\ 0\ 0]$
6. N_{A_1} is empty. $w(\{F,G,H\}) = w(\{F,G,I\}) = 3 < 9 = w(\{C,E,F\})$, so $x^{(6)} = \{F,G,H\}$ with $w = [1\ 4\ 4\ 3\ 3\ 3\ 2\ 1\ 0]$

Now, neighbourhood N_{A_1} is not empty, and the following solution is found exploring it: $x^{(7)} = \{F,G,H,I\}$.

Multiplicative example: *DLS* for the Max-SAT The *Max-SAT* problem requires to find a truth assignment for n logical variables to satisfy the maximum number of clauses in a set of m given ones. The classical single-flip neighbourhood, N_{F_1}, is generated by complementing the value of a single variable.

A possible way to apply *DLS* is to associate a penalty with each logical clause and modify the original objective function, that sums a unitary value for each satisfied clause

$$f(x) = \sum_{j \in C} \phi_j(x) \quad \text{with } \phi_j(x) = \begin{cases} 0 & \text{for } j \in U(x) \\ 1 & \text{for } j \in C \setminus U(x) \end{cases}$$

where $U(x) = \{j \in C : C_j(x) = \text{false}\}$ is the subset of clauses not satisfied by truth assignment x. In this case, the components of the solution are the logical clauses. They can be modified multiplying the binary contribution of each one by the penalty:

$$\tilde{f}(x) = \sum_j w_j \cdot \phi_j(x)$$

At the beginning, the multipliers are uniformly set to $w_j = 1$. The basic exchange heuristic maximizes the weight of the satisfied formulae, working on the combination of the penalty and objective function. The solution returned is not necessarily the one with the maximum number of satisfied clauses. Once a solution is found, *DLS* updates the penalty as follows:

- every time a local optimum is found (at the end of inner loops) increase the weight of unsatisfied formulae

  $$\forall j \in U(x)\ w_j := \alpha_{us} w_j$$

  with $\alpha_{us} > 1$, so that the algorithm will prefer to satisfy them in the following iterations;

- after a certain number of updates, or with a certain probability, reduce the penalty towards 1

$$\forall j \in C \; w_j := (1-\rho)w_j + \rho \cdot 1$$

with $\rho \in (0,1)$.

The rationale of the method consists in aiming to satisfy the currently unsatisfied clauses, focusing in particular on those that have remained unsatisfied for a longer time and more recently. This is a form of diversification based on memory.

The parameters tune intensification and diversification: small values of α_{us} and ρ preserve the current penalty (intensification), whereas large values of α_{us} and ρ cancel the current penalty (diversification).

9.4 Modifying the selection rule

The last way to modify the basic exchange scheme is to operate on the selection rule. While keeping the initial solution, the neighbourhood and the objective function, it is possible to modify the mechanism used to determine the incumbent solution. Instead of determining the best neighbour solution, other solutions may be considered, possibly even nonimproving ones. In fact, this is a basic point, since all methods that choose only improving solutions (such as the *first-best* strategy) still terminate the search in a local optimum.

The main problem that arises in these algorithms is the risk of cyclically visiting the same solutions. The two main strategies that allow to control this risk are:

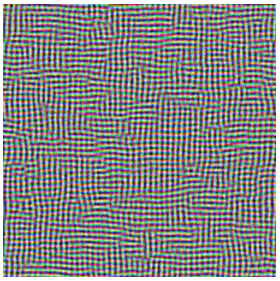
- *Simulated Annealing* (*SA*), which uses **randomisation** to make repetitions unlikely;
- *Tabu Search* (*TS*), which uses **memory** to forbid repetitions (or make them unlikely).

9.4.1 Simulated Annealing

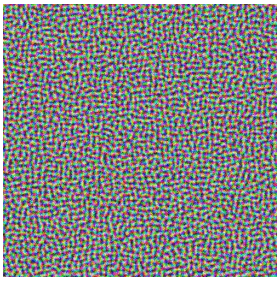
The **Simulated Annealing** algorithm has a long history: it derives from an algorithm proposed by Metropolis in 1953, not to solve an optimisation problem, but to simulate the behaviour of metals in the annealing process. This process consists in heating a block of metal up to a temperature close to its fusion point, allowing the particles of metal not only to oscillate, but to move in the volume of the metal, and possibly redistribute nearly at random. Then, the metal is cooled very slowly, so that the energy of the metal decreases, but converges to thermal equilibrium. In this way, all different parts of the metal have the same temperature. This means that the probability of every single particle of the metal to go in *any* direction is, more or less, the same.

The technological objective of the process is to guarantee that the thermal equilibrium distributes all particles in an even and regular way, obtaining a defectless *crystal lattice*, that corresponds to the minimum energy configuration, denoted as *base state*. This yields a material with useful physical properties.

Quite soon, the scholars realised that *continuous* optimisation problems and simulation problems have similar aspects. Then, this similarity was extended to combinatorial optimisation. In both cases, the idea is that, as physical systems have different states, so problems have different solutions. As each state of a physical system is characterised by a value of the energy function, each solution is characterised a value of the objective function. The base state, that has the minimum energy, corresponds to the globally optimal solution. A state transition in



(a) Slow annealing.



(b) Fast annealing.

Figure 9.7: Different result of annealing procedures.

physics describes the movements of particles; the state of a metal block is not fixed, but changes in time, with transitions from state to state. In combinatorial optimisation, this corresponds to moves from solution to solution, as in exchange algorithms. Finally, the overall process is ruled, in physics, by a numerical coefficient, that is the temperature. This corresponds to a numerical parameter in the resolution of combinatorial optimisation problems.

9.4.1.1 The details of Metropolis simulation algorithm

According to thermodynamics, at the thermal equilibrium the probability of observing each state i depends on its energy E_i

$$\pi'_T(i) = \frac{e^{\frac{E_i}{kT}}}{\sum_{j \in S} e^{\frac{-E_j}{kT}}}$$

where S is the state set, T is the temperature and k is Boltzmann's constant. The denominator of this expression just normalises the probability. The numerator shows that states of higher energy have smaller probabilities, whereas states of very low energy have larger and larger probabilities. Therefore, at thermal equilibrium states of low energy are more likely than states of high energy.

Boltzmann's constant k is only a value to relate the units of measure of energy and temperature. The temperature T, on the contrary, plays an interesting role: very large temperatures push the exponent of the numerator down to zero, the numerator towards 1 and the denominator towards the number of states. In other words, all states tend to be equiprobable. If the temperature is small, on the contrary, the role of energy becomes very important to distinguish the probabilities of different states.

Given a set of states with different energy levels, the Metropolis algorithm generates a random sequence of states, starting from a state chosen in some way. It iteratively perturbs the current state, generating another one at random, and decides whether to move from the current state to the perturbed one or not. The choice is done with probability:

$$\pi_T(i,j) = \begin{cases} 1 & E_j < E_i \\ \exp\left(\frac{E_i - E_j}{kT}\right) = \frac{\pi'(j)}{\pi'(i)} & E_j \geq E_i \end{cases}$$

The acceptance is deterministic if the perturbed state is improving, as decreasing the energy is the final purpose of the process; it is based on the conditional probability if the pertubed state is

worsening. In the probabilistic case, we have

$$\begin{aligned}\exp\left(\frac{E_i - E_j}{kT}\right) &= \frac{e^{\frac{-E_j}{kT}}}{\sum_{n\in S} e^{\frac{-E_n}{kT}}} \cdot \frac{\sum_{n\in S} e^{\frac{-E_n}{kT}}}{e^{\frac{-E_i}{kT}}} \\ &= \frac{e^{\frac{-E_j}{kT}}}{e^{\frac{-E_i}{kT}}} = e^{\frac{-E_j}{kT} - \frac{-E_i}{kT}} \\ &= e^{\frac{E_i - E_j}{kT}} = \frac{\pi'_T(j)}{\pi'_T(i)}\end{aligned}$$

which means that the probability that the current state is j is equal to the product of the probability that the previous state was i, multiplied by the probability of the transition from i to j, that is

$$\pi(j) = \pi(i,j) \cdot \pi(i)$$

This is the classical property of conditional probability. Since in this case $E_j - E_i$ is nonpositive, the move is accepted with low probability when the difference is very large (the transition increases the energy very much), with high probability when the difference is very small.

9.4.1.2 The algorithm

Simulated Annealing applies a similar process to the resolution of an optimisation problem. The pseudocode is presented in Algorithm 18.

Algorithm 18 Simulated Annealing Pseudocode

```
1:  procedure SA(I, x^(0))
2:      x := x^(0);
3:      x* := x^(0);
4:      T := T^[0];
5:      while Stop() = false do
6:          x' := RandomExtract(N, x)                ▷ random uniform extraction
7:          if f(x') < f(x*) or U[0,1] ≤ exp((f(x) − f(x'))/T) then
8:              x := x'
9:          end if
10:         if f(x') < f(x*) then
11:             x* := x'
12:         end if
13:         T := Update(T)
14:     end while
15:     return (x*, f(x*))
16: end procedure
```

The algorithm starts from a given solution $x^{(0)}$, that is also the best known one. As long as a termination condition is not reached, it randomly generates a single solution x' in a suitable neighbourhood N of the current solution x. The neighbourhood is not fully explored; it is only used to generate a solution, which could be worse than the current one even if improving neighbour solutions exist. The neighbour solution is then compared with the current one. If it is better, it becomes the incumbent, and replaces the current solution; otherwise, a random

number is extracted between $[0,1]$ from a uniform distribution of probability and compared to the already considered exponential expression to decide whether to accept or not the new solution. This implements the randomised choice. Since the generation of x' is typically fast, the computation of $\exp(\delta f/T)$ is not negligible. It has therefore been proposed to precompute a table of possible values in order to speed up this instruction (since T is a real number, there are infinitely many such values, but a sample of approximate values is acceptable, given that they are used to compute probabilities for randomised choices.

Since the algorithm does not explore completely the neighbourhood, the quality of the solution obtained is on average much worse, but the single iteration is also much faster (typically, it takes constant time, instead of a time proportional to the number of neighbour solutions). Therefore, the algorithm may perform many more steps to reach good solutions, but the steps are much quicker. There is a tradeoff, as usual, between the efficiency and the effectiveness of the approach. Notice also that, since the neighbourhood is not systematically explored, the methods that compute in constant time the feasibility and cost of a solution based on local information, that is on the previous neighbour solutions, cannot be applied. The computational advantage is then smaller than in other cases.

The acceptance criterion The profile that relates the probability of accepting a solution with the variation of the objective function is reported in Figure 9.8. If the new solution is improving, the exchange is always accepted ($\pi_T(x,x') = 1$); otherwise, the choice depends on the temperature T: large temperatures favour diversification, because nearly all solutions are accepted (in the extreme case, it becomes a random walk); small temperatures ($T \approx 0$) reject nearly all solutions (in the extreme case, it becomes a steepest descent algorithm). This profile is very similar to the one used by *ILS*, which actually derived it from *SA*.

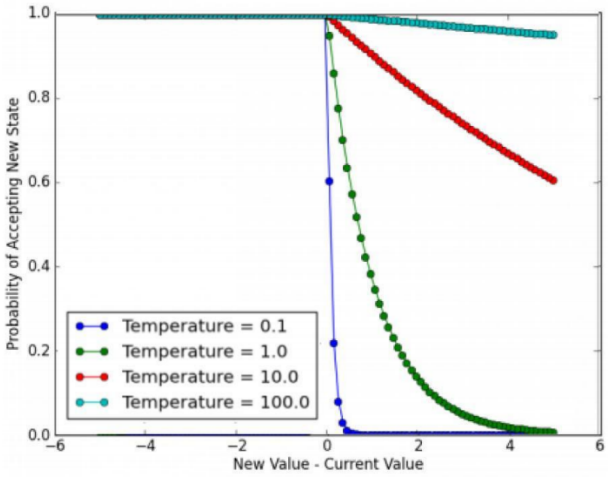


Figure 9.8: Probability distribution for different temperature values.

Asymptotic convergence to the optimum

SA enjoys a fundamental property of asymptotic convergence to the optimum. As all such results, it requires infinite time (while combinatorial optimisation problems can be solved exactly in finite time by the exhaustive algorithm). It is anyway important because it provides insights on the tuning of the temperature parameter.

The basic idea is that, since the algorithm performs random steps, the current solution x is a random variable. The *state probability* $\pi'(x)$ for a given step t combines, on all possible

predecessors $x^{(t-1)}$, the state probability $\pi'(x^{(t-1)})$ of the predecessor with the probability to choose the move from $x^{(t-1)}$ to x (that is uniform) and the probability to accept the move, that is $\pi_T(x^{(t-1)},x)$.

Since the state probability depends only on its value at the previous step (and not on the previous ones), the sequence of solutions is a **Markov chain**. For a fixed temperature T, the transition probabilities are stationary, and the process is a homogeneous Markov chain. Finally, if the search space for neighbourhood N is strongly connected, the probability to reach every state from every other state is strictly positive, and the process is an irreducible Markov chain.

Under all these assumptions, the state probability asymptotically converges to a stationary distribution that is independent from the starting state. This has the immediate practical consequence that the starting solution is irrelevant: for any solution $x^{(0)}$, after a sufficiently long time the probability that the current solution is x tends to be the same. It also tends to be higher for "good" solutions with respect to bad ones: the law is the same imposed by thermodynamics on physical systems at thermal equilibruim:

$$\forall x \in X \ \pi_T(x) = \frac{\exp(\frac{-f(x)}{T})}{\sum_{x \in X} \exp(\frac{-f(x)}{T})}$$

where X is the feasible region and T the temperature parameter.

The previous property holds for any fixed temperature T. The equilibrium distribution also converges to a limit distribution as $T \to 0$:

$$\pi(x) = \lim_{T \to 0} \pi_T(x) = \begin{cases} \frac{1}{|X|} & x \in X^* \\ 0 & x \in X \setminus X^* \end{cases}$$

which corresponds to a convergence to a certain globally optimal solution. This suggests to use very low values for T (setting $T = 0$ is not possible because it would give an indeterminate form $0/0$).

This results, however, holds at the equilibrium, after an infinite time. In practice, low values of T imply a high probability to visit a global optimum, but a slow convergence, as well. In any given finite time, the result can be far from optimal. The convergence becomes slower and slower as the temperature decreases.

The way out used by *SA* is to adopt a dynamic profile of temperature, simulating the annealing process: the algorithm starts from a high temperature value, which favours diversification, and decreases the temperature iteratively, waiting for the equilibrium at each update. The starting value $T^{[0]}$ should be high enough to allow to reach any solution quickly, but small enough to discourage visiting very bad solutions. A classical tuning for $T^{[0]}$ is obtained sampling the neighbourhood of the starting solution, $N(x^{(0)})$, and setting $T^{[0]}$ so that the expected value of the sampled solutions accepted reaches a given fraction α of the overall sample.

Updating the temperature The temperature update scheme must keep the temperature fixed for a number of iterations sufficient to reach a good approximation of the equilibrium. This is generally obtained going through subsequent phases $r = 0, \cdots, m$: each phase applies a constant value $T^{[r]}$ for $\ell^{[r]}$ iterations. The value of $T^{[r]}$ decreases exponentially from phase to phase:

$$T^{[r]} := \alpha T^{[r-1]} = \alpha^r T^{[0]}, \ \ \alpha \in (0,1)$$

while the length of the phases $\ell^{[r]}$ increases linearly from phase to phase, with values related to the diameter of the search graph (since it must be possible to move from every solution to every other solution with a reasonable probability).

If T is variable, the Markov chain x is no longer homogeneous. However, the aymptotic convergence property can be extended if the temperature decreases are slow enough. Theoretical results suggest that good parameters to tune the decrease depend on the instance. In particular, they depend on $f(\tilde{x}) - f(x^*)$, where $f(\tilde{x})$ is the second-best value of f. These values, however, are not known *a priori*.

Adaptive simulated annealing variants combine the randomisation mechanism with memory. In fact, they tune the temperature T based on the results: the value of T is decreased only if the solution has improved recently; otherwise, it is increased to favour diversification.

9.4.2 Tabu search

The **Tabu search** (*TS*), proposed by Glover in 1986, is one of the most powerful heuristics in use. Its basic idea is to keep the starting solution, neighbourhood and objective function of steepest descent, and also its selection rule (that is, choosing the best solution in the neighbourhood), but imposing a tabu (or prohibition) on solutions that have already been visited. This is done to avoid cycling behaviours, due to the "attraction" of known locally optimal solution. It is a sort of neighbourhood tuning, based on a memory mechanism, but allowing the visit of worsening solutions. Instead of setting

$$x^{(t)} = \arg \min_{x' \in N\left(x^{(t-1)}\right)} f\left(x'\right)$$

each neighbourhood exploration returns

$$x^{(t)} = \arg \min_{x' \in N\left(x^{(t-1)}\right) \setminus V} f\left(x'\right)$$

where $V = \{x^{(0)}, \ldots, x^{(t-1)}\} \subseteq X$ is the set of solutions visited in the previous iterations.

It is a simple idea, but in its basic form it poses several problems concerning the efficient and effective management of the tabu mechanism.

Efficiency

The main efficiency concern is how to keep track of tabu solutions. A basic exchange heuristic evaluates the feasibility of every subset generated by the operations that define the neighbourhood and (in the positive case) its cost. We have seen that both evaluations must be efficient, possibly taking constant time. Adding the tabu mechanism means that also the *tabu status* of the neighbour solutions (at least of those that improve the best one found so far) must be evaluated. The algorithm, in fact, selects as the incumbent the best feasible nottabu solution. Three conditions must be checked, instead of two.

The straightforward way to implement the evaluation of the tabu is to save each visited solution (i. e., solutions chosen as incumbent in some iteration) in a suitable structure, called **tabu list**, and checking each explored solution (i. e., neighbour solution in some iteration) to verify whether it matches a tabu one by making a query on the tabu list. The query must be as quick as possible, as it is performed very frequently; the update of the list occurs only once per iteration.

Using a real list, however, makes the query very inefficient: at iteration t, it requires time $\mathcal{O}(t\gamma(n))$, if $\gamma(n)$ is the time required to compare the explored solution to each tabu one. Notice that the iteration index t can easily be of the order of millions. The factor t can be reduced using

hash tables or search trees. However, as the number of iterations and visited solutions grows, the memory occupation grows correspondingly. These methods are rarely used in practice[1].

Tabu mechanisms using variations The *cancellation sequence method* and the *reverse elimination method* tackle these problems, exploiting the fact that, in general, the solutions visited form a chain with small variations between subsequent elements and that only few visited solutions are neighbours of the current one. As these techniques are quite sophisticated, we shall just summarise the gist of the methods. Since subsequent solutions in the tabu list are neighbours, they are very similar to each other, and instead of saving them completely it is possible to only save the *modifications* performed on them (in short, the move). This makes it easier to understand when a chain of moves cancel each other.

Consider, for example, a *MDP* instance, and a solution containing point 1, while points 2 and 3 are outside. Swapping 1 and 2 yields a solution including 2, while 1 is outside. Then, swap 2 and 3: the resulting solution includes 3, while 1 and 2 are outside. Now, the swap of 3 and 1 would go back to the original solution. This can be checked building the whole solution (possibly huge) or simply verifying that the chain of swaps $(1,2)$, $(2,3)$, $(3,1)$ is self-destroying. In other words, we need not save each solution, but only lists of moves, verifying whether there are chain of moves that eliminate each other, or that reduce to the opposite of one of the currently available moves (as $(1,2)$, $(2,3)$ reduces to $(1,3)$, that is the opposite of $(3,1)$).

Effectiveness

Even assuming that the tabu status can be checked efficiently, there are reasons against an explicit definition of the tabu list. In fact, forbidding solutions sparsifies the search graph: nodes are removed, together with the incident arcs, and consequently the graph becomes less connected. In extreme cases, the search graph can become disconnected, when the visited solutions create unavoidable "iron curtains" that block the search. At that point, the search is prisoner of one of the remaining connected components, possibly barred away from all optimal solutions.

Forbidding the visited solutions has another negative effects: it slows down the exit from attraction basins. In fact the selection rule always favours good solutions, that tend to be closer to the already visited local optimum. This implies that the search moves "spiralling", through the second best solution of the basin, the third and so on, with a "gradual filling" effect that takes a long time before a solution on the border of the basin is reached and allows to definitely leave. It would be much better to keep into account that the local optimum was reached from a certain direction, and that the search should move on in the same direction to leave it as quickly as possible, without ever coming back. This implies worser solutions, but a smaller number of iterations before the current solution reaches and trespasses the frontier of the basin of attraction.

The two issues described above suggest two modifications to the basic tabu search scheme. The need to avoid disconnecting the search graph suggests to avoid permanent prohibitions. The need to leave quickly the basin of attraction whose local optimum has already been found suggests to forbid not only the already visited solutions, but also solutions that are similar. These two suggestions seem to point in opposite direction: the former weakens the prohibition, the latter extends it. How to combine them?

[1] It must be mentioned that the current best performing algorithm for the *MDP*, proposed by Glover himself, is a *TS* using a hash table with three different hash functions to manage the tabu list limiting (but not avoiding) the probability of false positives ("collisions"), that is, of solutions falsely recognised as tabu.

Limits of the solution-based tabu mechanism

The following toy problem gives a very degenerate example of the two phenomena described above. Let the ground set be $B = \{1, \cdots, n\}$ and the feasible region include all subsets: $X = 2^B$. The objective function sums a nearly uniform additive term

$$\phi_i = 1 + \varepsilon i \;\; 0 < \varepsilon \ll 1$$

(each element has a slightly different cost), except for a single subset x^*, where a strong negative term is also introduced:

$$f(x) = \begin{cases} \sum_{i \in x} \phi_i & x \neq x^* \\ \sum_{i \in x} \phi_i - n - 1 & x = x^* \end{cases}$$

This term allows to choose arbitrarily the globally optimal solution x^*.

Considering the neighbourhood of all solutions at Hamming distance ≤ 1

$$N_{H_1}(x) = \{x' \in 2^B : d_H(x, x') \leq 1\}$$

the problem has a global optimum in x^* with $f(x^*) = n(n+1)\varepsilon/2 - 1 < 0$ and a local optimum in $\bar{x} = \emptyset$ with $f(\bar{x}) = 0$, provided that the Hamming distance between the two solutions is at least 2. The attraction basin of x^* includes only the n solutions with a single different bit, the attraction basin of $\bar{x}$ includes all other $2^n - n - 1$ solutions.

Starting from $x^0 = \emptyset = \bar{x}$ and forbidding all the solutions visited, the algorithm visits methodically most of 2^B, with the value of the objective ($f(x)$) and the Hamming distance from the starting local optimum ($d(x, \bar{x})$) going up and down. For $4 \leq n \leq 14$, the search graph is disconnected and the search gets stuck before visiting all solutions, but all solutions are at least explored, visiting a neighbour one. Therefore, the global optimum is actually found. For $n \geq 15$, the search gets stuck and some solutions are neither visited nor explored, possibly missing the global optimum.

t	f	x	$d(x, \bar{x})$
1	0	0000	0
2	$1+\varepsilon$	1000	1
3	$2+3\varepsilon$	1100	2
4	$1+2\varepsilon$	0100	1
5	$2+5\varepsilon$	0110	2
6	$1+3\varepsilon$	0010	1
7	$2+4\varepsilon$	1010	2
8	$3+6\varepsilon$	1110	3
9	$4+10\varepsilon$	1111	4
10	$3+9\varepsilon$	0111	3
11	$2+7\varepsilon$	0011	2
12	$1+4\varepsilon$	0001	1
13	$2+5\varepsilon$	1001	2
14	$3+7\varepsilon$	1101	3
15	$2+6\varepsilon$	0101	2

Table 9.1: Execution of tabu for $n = 4$.

Table 9.1 reports the execution for $n = 4$ with the simple additive objective function (no corrective term applied to introduce a globally optimal solution different from the starting solution $\bar{x} = 0000$. At iteration $t = 1$, the algorithm chooses the incumbent solution as the one with minimum cost in the neighbourhood, that is 1000. Its best neighbour is solution $x = \{0000\}$, that is forbidden, so the algorithm chooses $x = \{1100\}$. Then, items are progressively removed and added, with the objective moving down and up and the solution closer and farther from the starting solution until, at iteration $t = 15$, no move is feasible. Yet, solution 1011 has not been visited, so the graph is disconnected. It must be remarked that at iteration $t = 7$ solution 1011 is in the neighbourhood, so if it were the optimal solution it would be be selected and the global optimum would be found.

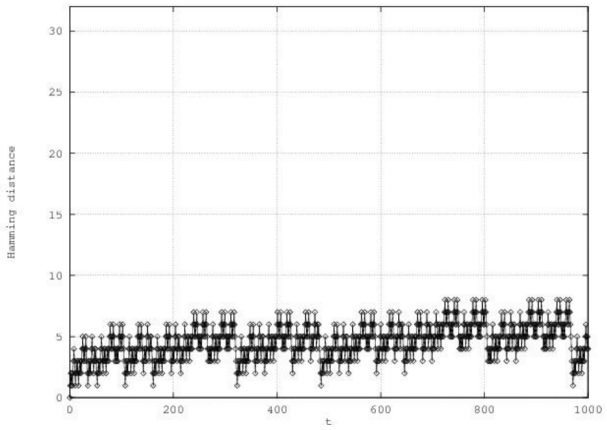


Figure 9.9: Objective function value for the previous example.

The objective function profile, shown in Figure 9.9, shows the limitations of the method. The solution x repeatedly gets far from $x^{(0)} = \bar{x}$ and close to it: it visits nearly the whole attraction basin of $\bar{x}$. For larger values of n, it does not get out of it, but gets stuck in a solution whose neighbourhood is fully tabu.

One could think of standing still, waiting for the oldest tabu to expire. In this case, the exploration proceeds, but the risk of looping is again to be taken into account.

Attribute-based tabu

The idea of **attribute-based tabu** is to forbid not only the visited solutions, but also those that share something with them. An *attribute* is an abstract concept, that we shall first introduce as such, giving examples later.

The method defines a finite set of attributes A. A subset $\bar{A} \subseteq A$, that is empty at the beginning, identifies tabu attributes. Every feasible solution $y \in X$ is associated with a subsets of attributes A_y. A solution with a tabu attribute is a *tabu solution*, and it is forbidden

$$y \text{ is tabu} \iff A_y \cap \bar{A} \neq \emptyset$$

A move from solution x to solution x' is acceptable only if x' is not tabu. After the move, the algorithm adds to $\bar{A}$ the attributes possessed by x and not possessed by x':

$$\bar{A} := \bar{A} \cup (A_x \setminus A_{x'})$$

so that x becomes tabu, while x' is not. This forbids to go back to x and to the previous solutions, but also avoids solutions similar to them. In this way the search get more quickly far away from

the visited local optima. However, it also strengthens the prohibition and, consequently, the problem of disconnecting the search graph. To deal with ths problem, the prohibition becomes temporary.

Temporary tabu and aspiration criterium The second basic idea is to give a limited persistence to the tabu, that is a limited number of iterations L, called the **tabu tenure**, so that "old" tabu attributes will no longer be tabu. Each attribute in $\bar{A}$ is removed from $\bar{A}$ when the tabu has expired, that is when L iterations have elapsed after its entrance in $\bar{A}$. This makes some previously tabu solutions eligible again. Visiting old solutions in a steepest descent algorithm would deterministically imply an infinite loop in the execution. Visiting old solutions with a different tabu list $\bar{A}$ could escape loops, even though it does not guarantee to avoid them. The tabu list $\bar{A}$ enriches the algorithm with memory. Tuning the tabu tenure is therefore fundamental for the effectiveness of the method.

Before discussing the tuning of L, it is worth pointing out two additional mechanisms. First, the tabu mechanism could forbid a globally optimal solution, provided that it is similar to an already visited one. To avoid this, one may introduce an **aspiration criterion**, such as declaring nontabu a solution that is better than the best known one, even if it has tabu attributes. If a solution is better than the best known one, of course, it does not induce any loop. Looser aspiration criteria have been proposed in the literature, but are not commonly used.

Second, if all neighbouring solutions are tabu, instead of terminating the algorithm, or keeping it still until some tabus expire, the algorithm may directly select the move with the oldest tabu (that would be performed anyway waiting for the expiration, but without wasting precious time). This can also be interpreted as another aspiration criterium, which overcomes a tabu status.

Algorithm 19 Tabu Search Pseudocode

```
1:  procedure TABUSEARCH(I, x^(0), L)
2:      x := x^(0);
3:      x* := x^(0);
4:      Ā := ∅;
5:      while Stop() = false do
6:          f' := +∞
7:          for y ∈ N(x) do
8:              if f(y) < f' then
9:                  if Tabu(y, Ā) = false or f(y) < f(x*) then x' := y f' := y
10:                 end if
11:             end if
12:         end for
13:         Ā := Update(Ā, x', L)
14:         if f(x') < f(x*) then
15:             x* := x'
16:         end if
17:     end while
18:     return (x*, f(x*))
19: end procedure
```

Algorithm 19 reports the abstract scheme of attribute-based *TS* algorithms. The practical implementation differs in that the management of the tabu list is not performed adding and

deleting elements from a real list of attributes. Before discussing this point, let us provide concrete examples of tabu attributes.

Tabu attributes The simplest (and yet very common) implementation of the concept of "attribute" is the inclusion of an element in the solution. The practical idea is that, when a move from x to x' expels an element i from the solution, the reinsertion of i in the solution is forbidden. This can be easily represented by defining $A_x = x$. Solution x has the attribute "presence of i" and x' hasn't got it. Therefore, the update rule includes i in the tabu attribute set $\bar{A}$, and automatically every solution including i becomes tabu.

A complementary definition of attribute is to focus on the added elements: when the move from x to x' inserts an element i into the solution, the tabu forbids the removal of i. This can be represented defining $A_x = B \setminus x$. Solution x has the attribute "absence of i", while x' has not got it. The update rule includes the i in $\bar{A}$ and every solution devoid of i becomes tabu.

Different attributes sets can be combined, each with its tenure and list: a common case is, when swapping i with j, to forbid the removal of j for L^{out} iterations and the insertion of i for L^{in} steps, with $L^{out} \neq L^{in}$.

Other attributes, less frequently adopted, are the value of the objective function, or of other auxiliary functions of the solution. Such tabus forbid solutions in which the considered function assumes a certain value, assumed by other previous solutions.

Complex attributes can be obtained combining simple attributes: the tabu can forbid the coexistence of two elements in the solution, or their separation (being one in and one out of it). Another example is to, after replacing element i with element j, to forbid the removal of j to include i, but allow the simple removal of j and the simple inclusion of i.

Efficient evaluation of the tabu status

The evaluation of the tabu status is performed for each neighbour solution, and therefore must be as efficient as possible, possibly taking constant time. Attribute-based tabu search does it with a simple data structure that associates with each attribute the information about when it became tabu (if ever). The data structure is a simple vector, with elements associated to the attributes[2]. In this way, when testing a move, one does not consider the whole resulting solution, but only the attributes removed from the old one.

If the tabu is on insertions, at iteration t the algorithm forbids the moves that add i when the tabu on i started at most L^{in} iterations ago. This corresponds to testing whether $t \leq T_i^{in} + L^{in}$, where t is the current iteration index. After choosing and performing the best move, the vector is updated setting $T_i^{in} := t$ for each attribute $i \in A_x \setminus A_{x'}$, that is for each element i removed (the set difference is not actually computed).

If the tabu is on deletions, at iteration t the algorithm forbids the moves that delete $i \in x$ when $t \leq T_i^{out} + L^{out}$ and, after choosing and performing the best move, it updates the vector, setting $T_i^{out} := t$ for each attribute $i \in A_x \setminus A_{x'}$, that is for each element i added.

Notice that, the two attributes are complementary, since either $i \in x$ or $i \in B \setminus x$. This means that, if one wants to combine them, a single vector T is enough for both. In fact, when using T_i, we know from other data structures whether i is in or out of the solution.

Example: the *TSP* Consider the neighbourhood N_{R_2} generated by 2-opt exchanges, and suppose that we adopt as attributes both the presence and the absence of arcs in the solution. In

[2]Of course, if the attributes are multidimensional, such as pairs, the vector will become a matrix, and so on.

other words, we forbid both the insertions of arcs that are currently out of the solution for L_{in} iterations and the removal of arcs that are currently in the solution for L_{out} iterations.

Since arcs are naturally defined by an adjacency matrix, a matrix T naturally replaces the auxiliary vector used to save the tabu status information. Each elements T_{ij} of the matrix is initialised to $-\infty$, so that every arc $(i,j) \in A$ is nontabu. In practical terms, the initial value must guarantee that t is strictly larger than both $T_{ij} + L_{in}$ and $T_{ij} + L_{out}$: such a value is $T_{ij} = -\max(L_{in}, L_{out}) - 1$.

At each step t, the algorithm explores $n(n-3)/2$ pairs of arcs from the current solution and the corresponding pairs of arcs that would replace them. Each move (i,j), which replaces (s_i, s_{i+1}) and (s_j, s_{j+1}) with (s_i, s_j) and (s_{i+1}, s_{j+1}) is tabu if any of the following conditions holds:

$$\begin{cases} t \leq T_{s_i,s_{i+1}} + L^{out} & \text{the removal of } (s_i, s_{i+1}) \text{ is tabu} \\ t \leq T_{s_j,s_{j+1}} + L^{out} & \text{the removal of } (s_j, s_{j+1}) \text{ is tabu} \\ t \leq T_{s_i,s_j} + L^{in} & \text{the removal of } (s_i, s_j) \text{ is tabu} \\ t \leq T_{s_{j+1},s_{i+1}} + L^{in} & \text{the removal of } (s_{j+1}, s_{i+1}) \text{ is tabu} \end{cases}$$

At the beginning, therefore, all moves are legal.

Once the move (i^*, j^*) is selected, the algorithm updates the auxiliary matrix, setting

$$T_{s_{j^*},s_{i^*+1}} := t$$

$$T_{s_{j^*},s_{j^*+1}} := t$$

$$T_{s_{i^*},s_{j^*}} := t$$

$$T_{s_{j^*+1},s_{i^*+1}} := t$$

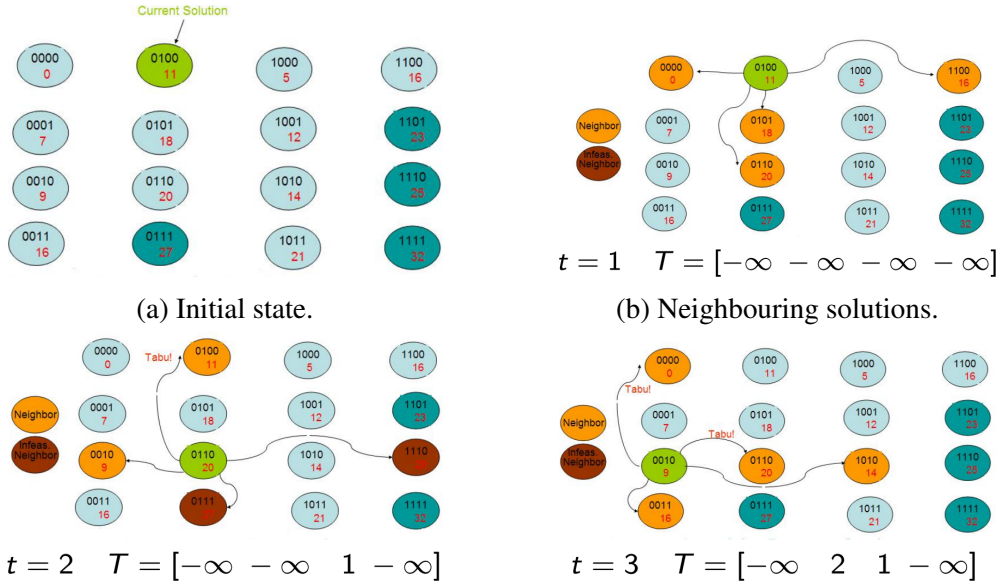
An interesting remark is that, when tuning the values of L^{in} and L^{out}, it is reasonable to set $L^{out} \ll L^{in}$ for all problems in which the solution is small with respect to the ground set (that is a frequent case). This is because, the length of the prohibition to remove an alement from a set (be it x or $B \setminus x$) should be proportionate to the cardinality of that set.

Example: the Max-SAT Consider the neighbourhood N_{F_1}, which includes the solutions obtained complementing the value of a variable: all n subsets thus obtained are feasible. This is equivalent to using both the existence and the absence of a pair (variable,value) in the solution. Since $|x| = |B \setminus x|$ for every $x \in X$, as each variable has exactly one of the two possible values in each feasible solution, the tabu mechanism can adopt the same tenure value L for both. This corresponds to forbidding the change of value of a variable. The attribute can be defined as the variable itself.

The algorithm proceeds as follows:

- at first set $T_i = -\infty$ for each variable $i = 1, \ldots, n$;
- at each step t explore the n solutions obtained complementing each variable;
- the move i, which assigns $x_i := \bar{x}_i$, is tabu at step t if $t \leq T_i + L$, so that at first all moves are non-tabu;
- perform the best move i^* and set $T_{i^*} := t$.

Example: the KP Consider the neighbourhood N_{H_1}, which includes all solutions at Hamming distance ≤ 1 from the current solution. This corresponds to adding or deleting a single element. We adopt as attribute the index of an element, and (for the sake of simplicity) forbid both the insertion and the removal of an element for the same time L. The tabu list is managed through a vector T which saves for each element $i \in B$ the iteration at which the last move was performed on it.



(a) Initial state.

(b) Neighbouring solutions.

(c) Second iteration.

(d) Third iteration.

Figure 9.10: Example of tabu on KP.

The example reported in Figure 9.10 considers an instance with $n = 4$ objects and capacity $V = 20$. We set the tabu tenure to $L = 3$. The starting solution $x^{(0)} = 0100$ (see Figure 9.10 (a)) has 4 neighbour solutions, represented in Figure 9.10 (b). They are all feasible and nontabu, and the best one is $x^{(1)} = 0110$. After moving to it, the algorithm sets $T = [-\infty, -\infty, 1, -\infty]$, assigning to the modified element 2 the index of the iteration in which it was modified. At iteration $t = 2$, half of the the neighbour solutions are feasible and half unfeasible; the feasible solution 0100 is tabu, because it is obtained modifying the third bit and $t + T_3 = 3 \leq L = 3$. In fact, it is once again the starting solution $x^{(0)}$. The algorithm is forced to chose $x^{(2)} = 0010$, even though it is worsening. At iteration $t = 3$, all neighbour solutions are feasible, but two are tabu. One of them is last solution visited, whereas the other is tabu as $t = 3 \leq T_3 + L = 1 + 3 = 4$, even if it was never visited.

Tuning the tabu tenure

The value of the tabu tenure L is a crucial parameter. If it is too large, it could conceal the global optimum by forbidding moves and disconnecting the graph. If the tabu tenure is too small, it can hold the exploration back in useless regions and, in the worst case, even produce cyclic behaviours. These can detected in practice by plotting the values of the solutions and observing a periodic behaviour of the plot.

Some general rules help to find the correct range of values: in general, the value of the tenure should grow as the instance size grows, but not proportionally. Glover suggested that $L \in \mathcal{O}(\sqrt{|A|})$ where A is the number of attributes. Quite often, if the size of the instances considered is not too different, a single value can be used with reasonable results.

Sometimes, no value can be found that is good overall, because in different regions of the solution space different values should be adopted. In this case, a common trick to break loops due to insufficient tenures is to extract the value of the tenure at random in a given range. The idea is that, instead of using always values too large or too small, one breaks the loop induced by a small tenure by randomly using, now and then, a large tenures, thus forbidding to repeat deterministically the same choices. On the other hand, instead of crippling the search with a large tenures, one randomly use a small one, now and then, exploiting it to intensify the search.

Another way to modify the tenure during the execution is to do it adaptively, that is taking into account the results obtained in the previous iterations. If a move improves the current solution, this is interpreted as moving towards a local optimum, and L is decreased (typically by 1) to favour intensification. If a move worsens the current solution, this is interpreted as moving away from an already visite local optimum, and L is increased (typically by 1) to favour diversification. Both variations occur remaining within a user-defined range $[L_{min}, L_{max}]$, to avoid unreasonably large or small tenure values.

Tabu search variants

Other adaptive strategies work in the long term, modifying the range in which the previous adaptive strategy is applied. The **reactive tabu search**, for example, saves the solutions visited in suitable data structure, as in the solution-based *TS*, with the aim to detect cyclic behaviours. Once a cycle is detected, the range $[L_{min}, L_{max}]$ is moved upwards. After a while, it is moved back downwards.

A very similar idea is the so called **frequency-based tabu search**, which saves in an auxiliary vector the frequency of each attribute (the number of iterations in which each attribute appeared in the visited solutions. If the frequency is too large, in order to diversify, the move could be declared tabu or the objective function could be modified as in the *DLS*. Vice versa, the search could also intensify, favouring the moves that introduce more frequent attributes.

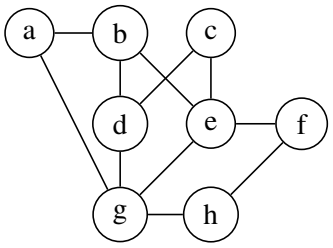
The **exploring Tabu Search** applies a restart strategy different from that of *VNS*. The idea is that, after a suitable number of iterations in which the best known result has not improved the search can be reinitialised. Instead of doing it with a random modification of a reference solution, this can be done retrieving one of the solutions previously explored, but not chosen. In particular, the "second-best solutions" of each neighbourood can form a pool and the search can be restarted from one of these solutions.

A final idea extends the fine neighbourhood tuning techniques. The idea is to heuristically limit the size of the neighbourood (more or less, depending on suitable parameters), start the exploration working in a reduced neighbourhood and periodically enlarge the neighbourhood, so that new solutions become reachable.

9.5 Exercises

9.5.1 Exercise 1

Given the following instance of the *Vertex Cover Problem* (*VCP*):



start from solution $x = \{b,c,e,g,h\}$ and perform two moves of a *Dynamic Local Search* (*DLS*) approach with the neighbourhood N_O based on the following moves: for every vertex $v \in V$ (external, but possibly also internal to the current solution x), move o_v adds v to the solution and removes all vertices that are redundant (nonnecessary to cover edges), keeping the added vertex last and considering the other ones in increasing degree order, and increasing index in case of ties. Use an additive penalty function $w(x) = \sum_{v \in x} w_v$ defined on the vertices and initialise it to 0 for the external vertices and 1 for the internal ones. After each move, increase by 1 the penalty of the vertices in the solution. The combined objective function is, therefore, $f(x) = |x| + \sum_{v \in x} w_v$.

Solution Neighbourhood N_O consists of at most $n = |V|$ solutions for all solutions x, as all moves are feasible, but different moves can generate the same solution. The current solution $x = \{c,h,b,e,g\}$ (with vertices sorted in increasing degree and increasing index order) has cost $f(x) = |x| = 5$ and $w = [\,0\,1\,1\,0\,1\,0\,1\,1\,]$, so that $w(x) = 5$, and $\tilde{f}(x) = f(x) + w(x) = 5 + 5 = 10$. Since x is nonredundant, all moves concerning the vertices in x simply return x. The other moves are:

- add a and remove a: $\tilde{f} = f + w = 5 + 5 = 10$;
- add d and remove c: $\tilde{f} = f + w = 5 + 4 = 9$;
- add f and remove h: $\tilde{f} = f + w = 5 + 4 = 9$.

The best ones are o_d and o_f; we select o_d lexicographically, obtaining $x' = \{b,d,e,g,h\}$. Now the penalty function is $w = [\,0\,2\,1\,1\,2\,0\,2\,2\,]$, so that $w(x') = 9$, and $\tilde{f}(x') = f(x') + w(x') = 5 + 9 = 14$.

The current solution is still nonredundant, so all moves associated to its vertices regenerate x'. The other moves are:

- add a and remove b and g: $\tilde{f} = f + w = 4 + 5 = 9$;
- add c and remove d: $\tilde{f} = f + w = 5 + 9 = 14$;
- add f and remove h: $\tilde{f} = f + w = 5 + 7 = 12$.

The best one is o_a, that generates $x'' = \{a,d,e,h\}$. Now the penalty function is $w = [\,1\,2\,1\,2\,3\,0\,2\,3\,]$, so that $w(x') = 8$, and $\tilde{f}(x') = f(x') + w(x') = 4 + 8 = 12$.

9.5.2 Exercise 2

Consider the following instance of the symmetric *Vehicle Routing Problem* (*VRP*)[3]

Cost	0	a	b	c	d	e
0	0	4	4	2	3	3
a	5	0	3	4	6	7
b	3	3	0	1	3	6
c	2	4	1	0	2	6
d	5	6	3	2	0	3
e	3	7	6	5	4	0

where 0 is the depot node, all other nodes have weights $v_i = 1$ and two vehicles of capacity $V = 3$ are available, so that no other vehicle can be added. Apply a *Variable Neighbourhood Descent* algorithm that starts from solution $x^{(0)} = \{(0,a),(a,b),(b,0),(0,d),\ (d,c),(c,e),(e,0)\}$ and applies first the 2-opt exchange neighbourhood $N_{\mathcal{R}_2}$ inside each circuit (not between different circuits![4]), then the single-node transfer neighbourhood $N_{\mathcal{T}_1}$.

Solution The 2-opt exchange neighbourhood $N_{\mathcal{R}_2}$ removes two nonconsecutive arcs from the current solution and add other two arcs so as to obtain a feasible solution. In general, the two removed arcs could belong to different circuits, but the text of the exercise limits the neighbourhood to exchanges within each single circuit. The cost of the starting solution is $f(x^{(0)}) = 4+3+3+3+2+6+3 = 24$. Only two exchanges are possible in solution x:

- remove $(0,d)$ and (c,e), add $(0,c)$ and (d,e), reverse arc (d,c): the cost of the new solution is $f = 4+3+3+2+2+3+3 = 20$;
- remove (d,c) and $(e,0)$, add (d,e) and $(c,0)$, reverse arc (c,e): the cost of the new solution is $f = 4+3+3+3+3+5+2 = 23$.

The best neighbour solution is $x^{(1)} = \{(0,a),(a,b),(b,0),(0,c),\ (c,d),(d,e),(e,0)\}$ with cost $f(x^{(1)}) = 20$.

Now, once again, $N_{\mathcal{R}_2}$ includes only two solutions:

- remove $(0,c)$ and (d,e), add $(0,d)$ and (c,e), reverse arc (c,d): the solution is once again $x^{(0)}$ and costs $f\left(x^{(0)}\right) = 27$;
- remove (c,d) and $(e,0)$, add (c,e) and $(d,0)$, reverse arc (d,e): the cost of the new solution is $f = 4+3+3+2+6+4+5 = 27$.

Therefore, $x^{(1)}$ is locally optimal for $N_{\mathcal{R}_2}$. The *VND* algorithm switches to $N_{\mathcal{T}_1}$.

Since no additional vehicle is available, the nodes assigned to each vehicle can only be transferred to the other one. The two nodes of the first circuit, however, cannot be moved due to the capacity constraint. Only the three nodes of the second can be moved to the first. After each transfer, the optimal route for both circuits should be recomputed (exactly or heuristically). Their small size allows to make it exactly:

[3] This exercise is rather an experiment, to see what an exercise on *VND* would look like. I am still trying to find a balance between significance and complexity.

[4] This restriction, as well as the previous one on the number of vehicles are meant to simplify the exercise: in a practical application they could be relaxed.

- transferring c to the first circuit requires to compute the best circuits to service nodes $\{a,b,c\}$ and $\{d,e\}$ from the depot; the former is $(0,a,b,c,0)$, the latter is $(0,d,e,0)$; their total cost is 19;
- transferring d to the first circuit requires to compute the best circuits to service nodes $\{a,b,d\}$ and $\{c,e\}$ from the depot; the former is $(0,d,b,a,0)$, the latter is $(0,e,c,0)$; their total cost is 24;
- transferring e to the first circuit requires to compute the best circuits to service nodes $\{a,b,e\}$ and $\{d,e\}$ from the depot; the former is $(0,a,b,e,0)$ (or $(0,e,a,b,0)$), the latter is $(0,d,e,0)$; their total cost is 26.

The best move is to transfer c, and the best neighbour solution is $x^{(2)} = \{(0,a),(a,b),(b,c),(c,0),(0,d),(d,e),(e,0)\}$ with cost $f(x^{(2)}) = 19$.

Now we should either go on with neighbourhood $N_{\mathcal{T}_1}$ testing the transfer of nodes a, b and c from the first to the second circuit or (if we consider the exploration of this neighbourhood too computationally expensive) switch back to neighbourhood $N_{\mathcal{R}_2}$. The exercise, however, is already too long.

Note: if one computes exactly the best circuit to visit a subset of nodes from the depot, obviously the 2-opt exchanges cannot improve further the solution. It would make more sense to transfer nodes heuristically (possibly testing the insertion without modifying the circuit, as in the *TSP* heuristics that augment circuits (the *Cheapest Insertion*, *Nearest Insertion* and *Farthest Insertion* algorithms). In this way, applying 2-opt exchanges could make sense, to reoptimise the single circuits (of course, only the two modified ones) after a node transfer.

9.5.3 Exercise 3

Consider the following instance of the *Parallel Machine Scheduling Problem* (*PMSP*) with 3 machines:

Task	a	b	c	d	e
d	9	3	4	5	10

Solution x, which assign tasks a and b to the first machine, tasks c and d to the second machine and task e to the third one is locally optimal for the single-transfer neighbourhood $N_{\mathcal{T}_1}$. Generate a new starting solution for the *VNS* metaheuristic with a *shaking* procedure based on neighbourhood $N_{\mathcal{T}_s}$, setting $s = 3$. For the sake of simplicity, select the s moves sequentially. Sort the moves first by increasing task index, then by increasing machine index. Assume that the pseudorandom number generator provides the following sequence: 0.15, 0.32, 0.17, 0.94, 0.52, ... The shaking procedure must avoid generating moves that overlap with previous ones (i. e., concern the same task), either filtering such moves and rescaling the probabilities at each step, or ignoring the overlapping moves and extracting additional random numbers.

Solution Solution $x = \{(a,1),(b,1),(c,2),(d,2),(e,3)\}$ is by assumption locally optimal for the neighbourhood $N_{\mathcal{T}_1}$ obtained transferring a single task from its current machine to a different one. Each task has $m-1$ possible moves (where $m = |M|$ is the number of machines), and they are all feasible, so that $|N_{\mathcal{T}_1}(x)| = n(m-1)$ for every solution x.

There are several ways to iterate these moves $s = 3$ times. One way is to enumerate the different triplets of transfers, generate a single pseudorandom number and select the corresponding

triplet. This is far too complex for a hand-made exercise, and remains quite complex even for a computer implementation, since the problem must be solved for several different values of s.

The more sophisticated way described in the text of the exercise enumerates the single transfers filtering away those that overlap with previous ones, then generates a pseudorandom number, selects and performs the corresponding move, repeating the process until s transfers are performed. This requires takes to mark the transferred tasks and rescale the probabilities in the subsequent steps:

1. at the first step, the set of single transfers, ordered first by task and then by machine, is:

$$(a,2),(a,3),(b,2),(b,3),(c,1)(c,3),(d,1),(d,3),(e,1),(e,2)$$

   each one with a probability equal to $1/10$. Pseudorandom number 0.15 corresponds to the second transfer, $(a,3)$.

2. now all transfers involving task a are removed from the set, that reduces to $(n-1)(m-1)=8$ moves, with probability $1/8$. Pseudorandom number 0.32 corresponds to the third remaining transfer $(c,1)$.

3. also the transfers involving task c are removed from the set, that reduces to $(n-2)(m-1)=6$ moves, with probability $1/6$. Pseudorandom number 0.17 corresponds to the first remaining transfer $(b,2)$.

The shaking procedure, therefore, modifies:

$$x=\{(a,1),(b,1),(c,2),(d,2),(e,3)\}$$

into

$$x'=\{(a,3),(b,2),(c,1),(d,2),(e,3)\}$$

that assigns task c to the first machine, tasks b and d to the second one, tasks a and e to the third.

The simpler shaking procedure described in the text enumerates the single transfers, generates a pseudorandom number, selects the corresponding move and performs it only if it does not overlap with the previous ones, repeating the process until s transfers are performed. This also requires takes to mark the transferred tasks, but not to rescale the probabilities:

1. at the first step, the set of single transfers, ordered first by task and then by machine, is:

$$(a,2),(a,3),(b,2),(b,3),(c,1)(c,3),(d,1),(d,3),(e,1),(e,2)$$

   each one with a probability equal to $1/10$. Pseudorandom number 0.15 corresponds to the second transfer, $(a,3)$.

2. now pseudorandom number 0.32 corresponds to the fourth transfer $(b,3)$.

3. pseudorandom number 0.17 corresponds to the second transfer, $(a,3)$, that has already been selected, so we ignore it;

4. pseudorandom number 0.94 corresponds to the tenth transfer: $(e,2)$.

The shaking procedure, therefore, modifies:

$$x = \{(a,1),(b,1),(c,2),(d,2),(e,3)\}$$

into

$$x' = \{(a,3),(b,3),(c,2),(d,2),(e,2)\}$$

that assigns tasks c, d and e to the second machine, tasks a and b to the third.

The second procedure is less efficient, as the number of additional extractions due to "collisions" with overlapping moves is in principle unbounded (its expected values depend on the values of s and n).

9.5.4 Exercise 4

Consider the instance of *Maximum Weighted Satisfiability Problem* (*Max-WSAT*) with

$$\text{CNF} = (x_1 \vee x_4 \vee \bar{x}_5) \wedge (x_2 \vee x_3 \vee x_5) \wedge (x_1 \vee x_2 \vee x_5) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_5)$$

$$w = [\,5\ 8\ 2\ 11\,]$$

Solution $x^* = (0,1,0,1,0)$ is locally optimal for the single-flip neighbourhood $N_{\mathcal{F}_1}$ (i.e., replace the value of one of the variables with the complementary one). Generate a new starting solution for the *VNS* metaheuristic with a *shaking* procedure based on neighbourhood $N_{\mathcal{F}_s}$, setting $s = 2$. For the sake of simplicity, select the s moves sequentially. Sort the moves first by increasing variable index. Assume the following sequence of pseudorandom numbers: $r = 0.90, 0.28, 0.52, 0.06, \ldots$ The shaking procedure must avoid generating moves that overlap with previous ones (i. e., flip the same variable), either filtering such moves and rescaling the probabilities at each step, or ignoring the overlapping moves and extracting additional random numbers.

Solution The neighbourhood $N_{\mathcal{F}_1}$ obtained flipping a single variable contains n solutions, that are all feasible for every given solution x.

Both shaking procedures enumerate $n = 5$ possible moves, with probability $1/5$ each. The pseudorandom number 0.90 corresponds to the fifth one (flip x_5). Now, the first procedure updates the probabilities forbidding any further flip of x_5 and assigning probability $1/4$ to all other variables. Pseudorandom numbe 0.28 corresponds to the second remaining move, that is to flip x_2. Therefore, solution $x = (0,1,0,1,0)$ becomes $x' = (0,0,0,1,1)$.

The second procedure also flips x_5, but does not rescale the probabilities. Pseudorandom number 0.28, however, still corresponds to the second move, that is to flip x_2. Therefore, solution $x = (0,1,0,1,0)$ becomes $x' = (0,0,0,1,1)$ also in this case.

9.5.5 Exercise 5

Consider the following instance of the *Knapsack Problem* (*KP*) with capacity $V = 10$:

Task	a	b	c	d	e	f	g
ϕ	6	4	5	10	3	1	8
v	6	1	3	4	2	1	5

A *Tabu Search* (*TS*) metaheuristic exploits neighbourhood $N = N_{H_1} \cup N_{S_1}$, where N_{H_1} includes the feasible solutions at Hamming distance ≤ 1 and N_{S_1} those obtained with a single swap of elements. The tabu tenures are $L_{\text{in}} = L_{\text{out}} = 1$ for the insertion and the removal of elements. At iteration $t = 5$, the current solution is $x^{(5)} = \{a, c, f\}$ and the tabu list is:

	a	b	c	d	e	f	g
T	4	4	3	2	$-\infty$	1	2

Assume that the moves in N_{H_1} precede those in N_{S_1}, that the former are sorted by increasing indices and those of the latter are sorted in lexicographic order with respect to the exiting element first and the entering element then.

Perform a single iteration, listing the nontabu solutions, selecting the best one, performing the corresponding move and updating the tabu list.

Solution Neighbourhood $N_{H_1}(x^{(5)})$ includes the feasible solutions at Hamming distance ≤ 1 from the current solution $x^{(5)}$. Neglecting $x^{(5)}$ itself, that is useless, its cardinality is $|N_{H_1}|(x^{(5)}) \leq n = 7$. To be more precise, the $|x^{(5)}| = 3$ deletions are feasible, whereas the feasibility of the $n - |(x^{(5)})| = 4$ additions must be checked comparing the volume of the added element with the residual capacity. Since $v(x^{(5)}) = 6 + 3 + 1 = 10 = V$, the residual capacity is zero, and no addition is feasible. The following table reports the feasible moves and the corresponding variations of the objective:

$N_{H_1}(x^{(5)})$	a	b	c	d	e	f	g
δf	-6	NF	-5	NF	NF	-1	NF

The tabu deletion moves have $t \leq T_i + L_{\text{out}}$, that is $T_i \geq t - L_{\text{out}} = 4$: hence, deleting a is tabu, deleting c and f is legal.

Neighbourhood $N_{S_1}(x^{(5)})$ includes the feasible solutions obtained swapping an internal element with an external one. The cardinality $|N_{H_1}|(x^{(5)}) \leq |x^{(5)}|(n - |x^{(5)}|) = 12$ must be evaluated comparing the volume variation δv with the residual capacity. The following table reports the feasible moves and the corresponding variations of the objective:

$N_{S_1}(x^{(5)})$	b	d	e	g
a	-6+4 = -2	-6+4 = -2	-6+3 = -3	-6+5 = -1
c	-5+4 = -1	NF	-5+3 = -2	NF
f	-1+4 = +3	NF	NF	NF

The tabu swap moves (i, j), with i exiting and j entering, have $t \leq T_i + L_{\text{out}}$ or $t \leq T_j + L_{\text{in}}$, that is $T_i \geq 4$: hence, all moves involving a or b are tabu, whereas the other ones are legal. The only remaining moves are:

- delete c (with $\delta f = -5$) or delete f (with $\delta f = -1$),
- swap c and e (with $\delta f = -2$).

The best one is to delete f, obtaining $x^{(6)} = \{a, c\}$.

After performing the move, the tabu list is updated marking the deleted element with the iteration index:

	a	b	c	d	e	f	g
T	4	4	3	2	$-\infty$	5	2

9.5.6 Exercise 6

Given the following instance of the symmetric *Travelling Salesman Problem* (*TSP*)

Cost	a	b	c	d	e	f
a	0	10	8	8	9	6
b	10	0	9	8	8	7
c	8	9	0	10	9	8
d	8	8	10	0	4	8
e	9	8	9	4	0	7
f	6	7	8	8	7	0

consider a *Tabu Search* metaheuristic exploiting neighbourhood $N_{\mathcal{R}_2}$ (that is, the 2-opt exchanges). The solution obtained at iteration $t = 5$ is $x^{(5)} = (a,b,f,c,d,e)$.

The following table reports the iteration at which each arc entered its current status (in or out of the current solution). Let the tenures be $\ell_{\text{in}} = 2$ and $\ell_{\text{out}} = 0$ (the inserted arcs can be immediately removed).

T	a	b	c	d	e	f
a	-3	1	-3	4	-3	-3
b	-3	-3	5	-3	4	-3
c	-3	2	-3	-3	-3	-3
d	1	-3	3	-3	-3	-3
e	-3	-3	-3	-3	-3	2
f	-3	3	-3	5	-3	-3

List the nontabu moves at iteration $t = 6$, perform the best one and update the table.

Solution The neighbourhood $N_{\mathcal{R}_2}(x)$ contains $n(n-3)/2 = 9$ solutions for every given solution x, as the two removed arcs must be different and nonadjacent. Exchange (i,j) removes arcs $(i,s(i))$ and $(j,s(j))$, and adds arcs (i,j) and $(s(i),s(j))$, where $s(i)$ is the node following i in the current solution. The solution obtained with this exchange at iteration $t = 6$ is tabu when $t \le T_{i,s(i)} + L_{\text{in}}$ or $t \le T_{j,s(j)} + L_{\text{in}}$, that is when $\max\left(T_{i,s(i)}, T_{j,s(j)}\right) \ge t - L_{\text{in}} = 4$. Consequently, all moves reinserting arcs (a,d), (b,c), (b,e), (f,d) are tabu.

The following table lists the exchanges, marking the tabu ones and reporting the variation of the objective function for the legal ones.

				$(j,s(j))$		
$(i,s(i))$	(a,b)	(b,f)	(f,c)	(c,d)	(d,e)	(e,a)
(a,b)	-	-	Tabu	-10-10+8+8 = -4	Tabu	-
(b,f)	-	-	-	Tabu	-7-4+8+7 = +4	Tabu
(f,c)	-	-	-	-	Tabu	-8-9+7+8 = -2
(c,d)	-	-	-	-	-	-10-9+9+8 = -2
(d,e)	-	-	-	-	-	-
(e,a)	-	-	-	-	-	-

Only 4 of the 9 exchanges are legal. The best one removes (a,b) and (c,d) and adds (a,c) and (b,d), turning $x = (a,b,f,c,d,e)$ with $f(x) = 10+7+8+10+4+9 = 48$ into $x' = (a,c,f,b,d,e)$ with $f(x') = 8+8+7+8+4+9 = 44$, confirming that $\delta f = -4$. The new table is the following.

T	a	b	c	d	e	f
a	-3	**6**	**6**	4	-3	-3
b	-3	-3	5	**6**	4	-3
c	-3	2	-3	**6**	-3	-3
d	1	-3	3	-3	-3	-3
e	-3	-3	-3	-3	-3	2
f	-3	3	-3	5	-3	-3

9.5.7 Exercise 7

Consider the following instance of the *Parallel Machine Scheduling Problem* (*PMSP*) with 3 machines:

Task	a	b	c	d	e
d	8	13	7	12	6

Let the starting solution $x^{(0)}$ assign tasks a and b to the first machine, tasks c and d to the second machine and task e to the third one. Apply a *Tabu Search* metaheuristic exploiting the single-transfer neighbourhood $N_{\mathcal{T}_1}$, visited with the global-best strategy. In case of ties, select the task with the smallest index; in case of further ties, select the machine with the smallest index. Set the tabu attribute as the index of the moved task and fix the tenure to $L = 2$.

Apply two iterations of the method, showing the moves performed and the update of the data structure to manage the tabu.

Solution The single-transfer neighbourhood $N_{\mathcal{T}_1}$ includes $n(m-1) = 10$ solutions, where $n = |T|$ is the number of tasks and $m = |M|$ the number of machines. Since the tabu attribute is the index of the task transferred, the tabu status is determined by a vector reporting for each task the last iteration in which the task was moved. At the beginning, this vector is initialised with $-\infty$, that is, $-L-1 = -3$.

$$T = [\,-3\ -3\ -3\ -3\ -3\,]$$

and all transfers are legal.

The following table represents the values of all possible exchanges:

	Machine		
Task	1	2	3
a	-	$\max(13, 8+7+12, 6) = 27$	$\max(13, 7+12, 8+6) = 19$
b	-	$\max(8, 13+7+12, 6) = 32$	$\max(8, 7+12, 13+6) = 19$
c	$\max(8+13+7, 12, 6) = 28$	-	$\max(8+13, 12, 7+6) = 21$
d	$\max(8+13+12, 7, 6) = 33$	-	$\max(8+13, 7, 12+6) = 21$
e	$\max(8+13+6, 7+12, 0) = 27$	$\max(8+13, 7+12+6, 0) = 25$	-

The best ones are $(a,3)$ and $(b,3)$: we select the first, lexicographically. Now the current solution is $x^{(1)} = \{(a,3),(b,1),(c,2),(d,2),(e,3)\}$ and the tabu attributes are:

$$T = [\,1\ -3\ -3\ -3\ -3\,]$$

The following table represents the values of all possible exchanges, marking the ones that are tabu at iteration $t = 1$. These exchanges have $t \leq T_i + L$, that is $T_i \geq t - L = 0$. In short, they are the transfers of task a:

Task	Machine 1	2	3
a	Tabu	Tabu	-
b	-	$\max(0,13+7+12,8+6)=32$	$\max(0,7+12,8+13+6)=27$
c	$\max(13+7,12,8+6)=20$	-	$\max(13,12,8+7+6)=21$
d	$\max(13+12,7,8+6)=25$	-	$\max(13,7,8+12+6)=26$
e	$\max(13+6,7+12,8)=19$	$\max(13,7+12+6,8)=25$	-

The best one is $(e,1)$. Now the current solution is $x^{(1)}=\{(a,3),(b,1),(c,2),(d,2),(e,1)\}$ and the tabu attributes are:

$$T=[\,1\ -3\ -3\ -3\ 2\,]$$

9.5.8 Exercise 8

Given the following instance of the *Knapsack Problem* with capacity $V=12$:

Objects	a	b	c	d	e	f	g
Prize ϕ	11	2	4	9	8	4	9
Volume v	5	1	2	4	3	1	6

consider the starting solution $x^{(0)}=\{a,b,d\}$. Perform two steps of *Simulated Annealing*, assuming neighbourhood $N_{\mathcal{H}_2}\setminus N_{\mathcal{H}_1}$ (that is, Hamming distance exactly equal to 2).

Select the moves by considering in lexicographic order the pairs of elements whose bit in the incidence vector is complemented. Let the temperature parameter be $T=1/\ln 2$, so that $e^{-\Delta f/T}=2^{-\Delta f}$, and the pseudorandom numbers generated be 0.3, 0.4, 0.7, 0.2, 0.8, 0.1.

Solution The neighbourhood $N_{\mathcal{H}_2}\setminus N_{\mathcal{H}_1}$ includes the feasible solutions obtained complementing the bits of two elements of the ground set in the incidence vector representation. For $x^{(0)}=\{a,b,d\}$, this representation is $\xi^{(0)}=[\,1\,1\,0\,1\,0\,0\,0]$. The pairs of elements are $n(n-1)/2=21$, but many of them are unfeasible.

The abstract scheme of *SA* requires to extract a neighbour solution at random with uniform probability, compute its value and decide probabilistically whether to accept it or not. This requires, in theory, to know the number of neighbour solutions, that is the number of feasible exchanges. There are several possible ways to manage this problem.

The first one is to explore the exchanges and list the feasible ones. The following table lists them, marking the unfeasible exchanges with label NF and the feasible ones with F. The lower half is empty for symmetry.

i (v_i)	a (5)	b (1)	c (2)	d (4)	e (3)	f (1)	g (6)
a (5)		F	F	F	F	F	F
b (1)			F	F	F	F	NF
c (2)				F	NF	NF	NF
d (4)					F	F	F
e (3)						NF	NF
f (1)							NF
g (6)							

This approach requires a full neighbourhood exploration, even if it avoids the computation of the objective function. From this point of view, it is rather contrary to the spirit of *SA*. Since there are $|N_{\mathcal{H}_2} \setminus N_{\mathcal{H}_1}| = 14$ neighbour solutions, pseudorandom number 0.3 selects the fifth, that is (a, f). Removing a and adding f yields $x' = \{b, d, f\}$ with value $f(x') = 2+9+4 = 15$. The value of the starting solution was $f(x^{(0)}) = 11+2+9 = 22$. The new solution is worse and the probability to accept it is $e^{-\Delta f/T} = 2^{-\Delta f} = 2^{-7}$, that is very small. Pseudorandom number 0.4 is larger, and the move is rejected.

A second way to manage the existence of unfeasible exchanges is to generate the exchange at random and estimate its feasibility *a posteriori*, generating another exchange if the first is unfeasible. This is more in line with the spirit of *SA*, but it can be very inefficient if the probability of generating an unfeasible exchange is large. In this case, it is neither large nor small: the advantage of avoiding a full exploration is probably strong enough. In this case, we have 21 exchanges and the first pseudorandom number (0.3) suggests to select the seventh in lexicographic order, that is (b, c). Removing b and adding c yields $x' = \{a, c, d\}$ with value $f(x') = 5+2+4 = 11$.[5] The value of the starting solution was $f(x^{(0)}) = 11+2+9 = 22$. The new solution is worse and the probability to accept it is $e^{-\Delta f/T} = 2^{-\Delta f} = 2^{-11}$, that is very small. Pseudorandom number 0.4 is larger, and the move is rejected also in this case.

A third approach, that we just mention, could be to try and find classes of moves that are unfeasible by construction, so as to reduce the number of exchanges approximating the actual number of neighbour solutions. For example, one could exploit the fact that the $|V \setminus x| \cdot (|V \setminus x| - 1)/2 = 6$ pairs that only add elements are all unfeasible because the residual capacity of the current solution is $V - \sum_{i \in x} v_i = 12-5-1-4 = 2$ and the volume of all pairs of elements in $V \setminus x$ exceed it. Therefore, there are $21-6 = 15$ candidate exchanges, which is a much better approximation of the correct number. With the given pseudorandom numbers, this approach would lead to the same result as the first one (mind that the extraction is made only on the 15 exchanges with at least a deletion).

Since all approaches yielded the starting solution, let us consider the second move with pseudorandom number 0.7. In this case, the three approaches suggest different exchanges. The first one suggests the tenth move, that is (b, f), feasible by construction: removing b and adding f yields $x' = \{a, d, f\}$ with value $f(x') = 11+9+4 = 24$, that is better than the current one, and is therefore directly accepted.

Based pseudorandom number 0.7, the second approach suggests the fifteenth move, that is unfeasible. The following pseudorandom number, 0.2, suggests the fifth move, that is (a, f) and is feasible. Removing a and adding f yields $x' = \{b, d, f\}$ with value $f(x') = 2+9+4 = 15$, that is worse than the current one. We have already met it, and seen that the probability to accept it is $e^{-\Delta f/T} = 2^{-\Delta f} = 2^{-7}$, that is very small. The pseudorandom number used to decide is changed (it is 0.8), but, still, the move is rejected.

Based pseudorandom number 0.7, the third approach suggests the eleventh exchange, that is (a, g). Since we have not explored the neighbourhood, we have to test its feasibility, and find that it is unfeasible. Pseudorandom number 0.2 suggests the third exchange, that is (a, d). Removing both a and d yields $x' = \{f\}$ with value $f(x') = 4$, that is much worse than the current one, and the probability to accept the is $e^{-\Delta f/T} = 2^{-\Delta f} = 2^{-18}$, so small that pseudorandom number 0.8 suggests to reject the move.

[5] ATTENZIONE: no, questi sono i volumi, non i premi, per cui la soluzione non e' migliore. L'esercizio va rivisto.

9.5.9 Exercise 9

Given the instance of the *Capacitated Minimum Spanning Tree Problem*, with root vertex a, weight function $w_v = 1$ for all $v \in V \setminus \{a\}$ and $w_a = 0$, capacity $W = 2$ and cost function:

Cost	a	b	c	d	e	f
a	0	10	5	4	9	8
b	10	0	9	8	11	7
c	5	9	0	12	15	3
d	4	8	12	0	2	13
e	9	11	15	2	0	7
f	8	7	3	13	7	0

Starting from solution $x = \{(a,b),(a,d),(a,f),(b,c),(d,e)\}$, apply one step of *Simulated Annealing* with neighbourhood $N_{\mathcal{S}_1}$. Since most swaps yield unfeasible solutions, split the selection of the move in two parts: first select the added edge, then enumerate the edges whose removal yields feasible solutions and select one. In both steps, sort the candidate edges lexicographically. Set the current temperature to $T = 5$ and use the following pseudorandom number sequence: $r = 0.95, 0.21, 0.45 \ldots$

Solution Neighbourhood $N_{\mathcal{S}_1}(x)$ includes all feasible solutions obtained adding an edge to the current tree and breaking the loop thus formed by removing another edge in the loop. Since the graph has $|V| \cdot (V| - 1) = 15$ edges and $|V| - 1 = 5$ belong to the current solution, the external edges are 10, namely (a,c), (a,e), (b,d), (b,e), (b,f), (c,d), (c,e), (c,f), (d,f) and (e,f). Pseudorandom number 0.95 suggests to select the last one, that is (e,f). Adding edge (e,f) closes the loop (a,d,e,f,a), which can be broken in three ways (keeping the new edge), but only one yields a feasible solution, since the others produce a single branch of three vertices, exceeding the capacity constraint. Therefore, we remove (d,e). It is not necessary, but also not forbidden, to use a pseudorandom number for this decision. The variation of the objective function is $\delta f = c_{ef} - c_{de} = +7 - 2 = +5$. The probability to accept the move is $e^{-\delta f/T} = e^{-1} = 1/e$. Pseudorandom number 0.21 suggests to accept the move (of course, if one used 0.21 to select the removed edge (d,e), one has to apply pseudorandom number 0.45, that suggests to reject the move.

For the sake of completeness, one could consider the approach in which all neighbour solutions are enumerated before selecting one. In this case:

- adding edge (a,c) would require to remove either (a,b) or (b,c);
- adding edge (a,e) would require to remove either (a,d) or (d,e);
- adding edge (b,d) would require to remove either (a,b) or (a,d), but both solutions would be unfeasible;
- adding edge (b,e) would require to remove (a,b), (a,d) or (d,e), but all solutions would be unfeasible;
- adding edge (b,f) would require to remove either (a,b) or (a,f), and only the second removal would be feasible;
- adding edge (c,d) would require to remove (a,b), (a,d) or (b,c), but all solutions would be unfeasible;

- adding edge (c,e) would require to remove (a,b), (a,d), (b,c) or (d,e), but all solutions would be unfeasible;
- adding edge (c,f) would require to remove (a,b), (a,f) or (b,c), and only the last removal would be feasible;
- adding edge (d,f) would require to remove either (a,d) or (a,f) but both solutions would be unfeasible;
- we have already analysed the addition of (e,f) and its only feasible removal.

Therefore, overall 6 solutions are feasible. Pseudorandom number 0.95 suggests the last one, and the exercise proceeds accepting the move with pseudorandom number 0.21.

Part V

Recombination algorithms

CHAPTER 10

Recombination metaheuristics

The last class of solution-based heuristics for combinatorial optimisation problems is composed by the **recombination heuristics**. In general, they adopt memory or random steps, so that they can be straighforwardly defined metaheuristics. Moreover, contrary to constructive and exchange algoritms, this class of methods has no basic scheme to discuss and extend, but only the general idea of **handling many solution at a time** to generate other solutions.

10.1 Introduction to recombination metaheuristics

Recombination heuristics manage a set of solutions, which is generally called the **population**, whereas the single solutions are often called **individuals**. This strongly differs from the case of constructive and exchange algorithms, that most of the time manipulated only a single solution (or subset) at a time. The only exception is provided by the *Ant System*, that actually shares an important aspect with many recombination algorithms: it is based on a natural metaphor. The words "population" and "invidivual", in fact, are commonly used due to the same reason (even if not all recombination algorithms have such an origin).

Recombination algorithms manipulate several "individuals" at a time, recombining them in order to generate new ones. During this process, they often adopt other elements taken from constructive and exchange algorithms, sometimes renaming them (in particular, when the natural metaphor is kept as a guide for the design).

In the following, we present two main families of recombination heuristics: the first one has a mainly deterministic structure, adopting random steps only as a last resort. In some cases, a purely deterministic version, which could be classified as a heuristic, clearly exists. They mainly derive from scholars interested in developing effective optimisation heuristics. The main examples of this family are *Scatter Search* and *Path Relinking*. The second family is based on a strong use of randomisation, which sometimes derives from a biological metaphor, referring to evolutionary mechanisms or the behaviour of living creatures. The randomicity is originally meant to simulate the behaviour of these agents, and the application to optimisation is more of a second thought. Examples are the *genetic algorithms*, the *memetic algorithms* and the *evolution strategies*.

10.1.1 General concepts

The recombination algorithms are based on the idea that many nonoptimal solutions share subsets with the optimal ones: some components of these solutions are identical, or very similars, to components of an optimal solution. The concept of *component* is vague on purpose: in general it is a subset of elements of the ground set, but more specifically it could be a circuit in the

VRP, a truth assignment for a subset of variables in a Max-SAT, and so on; in general, it is a part of the solution whose quality can be measured somehow.

Since different solutions, in general, share different components with the optimal ones, instead of trying to correct the "bad" components of a single solution, one could take several solutions, extract the "good" components from each one and combine them into a better final solution. If one could identify the good components and combine them correctly, one could obtain the optimal solution in a much more efficient way than just manipulate each single solution until it coincides with the optimal one.

A general scheme of recombination heuristics, therefore, consists in building a starting population of solutions and repeatedly modify its elements until a suitable termination condition is satisfied. Each iteration is usually called a **generation**, for the usual metaphorical reason. At each generation, the algorithm extracts a collection of subsets of individuals and applies to each subset some operator to generate new subsets of individuals. Sometimes, the subsets are singletons, and the operator just modifies a single individual, as in exchange heuristics (but working in parallel on a population). In other cases, and this is characteristic of recombination algorithms, the operators work on a subset of more than one individual and produce a new individual or a new subsets of individuals. At the end of the generation, all the generated individuals are collected, and the algorithm applies a *selection rule* to choose those that are worthy of being included in the population at the next generation. According to the algorithm, the individuals of the original population can be admitted or not to the selection. According to the algorithm, if different operators generate identical solutions, these can be admitted to the selection step or reduced to a single one. Usally, the class of more deterministic heuristics keeps the old solutions and avoids generating multiple copies, since losing information or keeping redundant information is seen as a waste of space and time; the more simulation-oriented algorithms usually allow copies and the removal of old solutions (of course, always saving the best known one.

The starting population can be obtained, as the starting solution of an exchange algorithm, randomly or using contrustive (meta)heuristics; of course, they can also be obtained using exchange (meta)heuristics.

Except for the few purely deterministic heuristics that we shall discuss in the following, the termination condition is one of the absolute or relative conditions already discussed for constructive and exchange metaheuristics.

10.2 Scatter search

Scatter search (*SS*) was proposed by Glover, the creator of *TS* and many other optimization methods, in 1977. Its general idea is to recombine solutions drawn from a population, as in all recombination heuristics. The solutions are, however, all locally optimal, since they are the result of an auxiliary exchange procedure. More in detail, the population called **reference set**, $R = B \cup D$, consists of two parts that play complementary roles. Subset B includes the best locally optimal solutions found during the search. Subset D includes the locally optimal solutions that are "farthest" from B and from each other, according to a suitable definition of distance, that typically is the Hamming distance.

The pseudocode of *SS* is provided in Algorithm 20[1]. The algorithm starts with a given population P of solutions (generated as usual, randomly or with constructive heuristics). It applies a suitable steepest descent procedure to each solution in the current population, so as to find a locally optimal solution z. These solutions build the reference set with the following

[1]The pseudocode and its description must be updated with those provided in the slides.

process. First, we try and add z to the subset B of the best solutions, if it has not already been included in a previous step (duplicate solutions are forbidden). If $|B|$ has not yet reached the required cardinality, z is added to B; otherwise, z is compared to the worst solution of B (y_B) and, if it is better, it replaces that solution. If it is not better, and it has not already been included in the subset D of the most diverse solutions in a previous step (duplicate solutions are forbidden also in this case), we try and add it to D. If $|D|$ has not yet reached the required cardinality, we simply add z to D. Otherwise, we determine the less diverse solution in D, y_D, that is the solution with minimum total Hamming distance from all solutions in $B \cup D$, $d(y_D, B \cup D \setminus y_D)$. Then, we compute the vaule of that distance for z, that is $d(z, B \cup D \setminus y_D)$, and compare the two values. If z is better, it replaces y_D; otherwise, it is definitively rejected. Adding z to B or D is an update of the reference set. The values of $|B|$ and $|D|$ are parameters defined by the user. The number of starting solutions required to build R is at least $|B| + |D|$, but it could be much larger, due to the removal of duplicates.

At each iteration, the algorithm scans all pairs of solutions $(x, y) \in B \times (B \cup D)$ and recombines x and y. The purpose is to build new starting points for the exchange heuristic that either combine best solutions ($B \times B$) to intensify the search or best and diverse solutions ($B \times D$) to diversify the search without losing too much quality. The pairs of solutions from $D \times D$ are not considered: the quality would probably be bad and combining distant solutions does not guarantee to generate a distant one. All recombined solutions z are immediately improved with the exchange procedure, to obtain locally optimal solutions that form a new population. Finally, the population undergoes the process described above to update the reference set.

The algorithm iterates on the reference set until a termination condition is verified. If the recombination procedure (and the exchange procedure) are deterministic, a natural termination condition is that the reference subset R did not change in the last iteration. In this case, in fact, it will no longer change, and any further computation is useless. Additional mechanisms can, however, turn this heuristic into a metaheuristic, allowing to apply the usual general termination conditions.

10.2.1 The algorithm

10.2.2 Recombination procedure

The recombination procedure, that is a fundamental element of *SS*, depends on the specific problem considered. However, it is usually based on the manipulation of the two subsets x and y. Recombination procedures with more than two solutions are possible, but much less common.

The basic idea is that the elements included in both x and y are likely to be good. Therefore, in general, the recombined solution z is initialised with such elements:

$$z := x \cap y$$

Then, z is augmented adding elements from $x \setminus y$ or $y \setminus x$. This can be done at random or with a greedy selection criterion (that is limited to $x \setminus y$ or $y \setminus x$, instead of working on the whole of $B \setminus z$). The random approach turns the method into a metaheuristic, and allows to proceed the search also in the case in which an iteration does not update the reference set (even if relying only on this randomisation is not very promising: other diversification mechanisms are advisable). The greedy approach is more likely to provide better solutions, but also more likely to generate duplicates. As well, the augmentation step can draw elements alternatively from the two sources (one from x, one from y, etc... and deciding where to start from is also a parameter)

Algorithm 20 Scatter Search Pseudocode

```
1: procedure SCATTERSEARCH(I, P)
2:     B := ∅
3:     D := ∅
4:     repeat
5:         Stop = true
6:         for each x ∈ P do
7:             z := SteepestDescent(I, x)
8:             y_B := arg max_{y∈B} f(y)
9:             y_D := arg min_{y∈D} d_H(y, B ∪ D \ {y})
10:            if f(z) < f(y_B) then
11:                B := B \ {y_B} ∪ {z}
12:                Stop := false
13:                if f(z) < f(x*) then
14:                    x* := z
15:                end if
16:            else
17:                if d(z, B ∪ D \ {y_D}) > d(y_D, B ∪ D \ {y_D}) then
18:                    D := D \ {y_D} ∪ {z}
19:                    Stop := false
20:                end if
21:            end if
22:        end for
23:        P := ∅
24:        for each (x, y) ∈ B × (B ∪ D) do
25:            P := P ∪ Recombine(x, y, I)
26:        end for
27:    until Stop = True
28:    return (x*, f(x*))
29: end procedure
```

or freely from their union $(x \cup y) \setminus z$. The alternate mechanism is probably appropriate when all solutions have the same cardinality, but limit the search in the opposite case. The free approach has a stronger risk to regenerate one of the two solutions, instead of a proper recombination. At the end of the augmentation phase, in particular (but not only) if z is unfeasible, an auxiliary constructive heuristic can be used to complete the solution with elements drawn from $B \setminus (x \cup y)$ or $B \setminus z$. If this is still insufficient to find feasible solutions, an auxiliary exchange heuristic, called **repair procedure**, can be applied to modify z minimising the infeasibility, in an attempt to obtain a feasible solution.

The definition of the auxiliary procedures and the different choices in the basic steps identifies several possible variants of the method. Let us consider some examples of recombination procedures for different problems.

MDP The recombination might start setting z as the intersection of two given solutions

$$z := x \cap y$$

Since feasibility only requires to include k elements in the final solution, the recombination procedure could augment z with $k - |z|$ points chosen randomly or greedily, alternatively or freely from $x \setminus z$ and y *setminusz*. This is always possible: no repair procedure is required.

Max-SAT Given two solutions, that assign a value to each of the n logical variables, the recombined solution z could be initialised as

$$z := x \cap y$$

and augmented with $n - |z|$ truth assignments, chosen randomly or greedily, alternatively or freely from $x \setminus z$ and $y \setminus z$. No repair procedure is required.

KP Given two solutions, that are subsets of objects, the recombined solution z could be initialised as

$$z := x \cap y$$

and augmented choosing objects from $x \setminus z$ and $y \setminus z$, randomly or greedily, alternatively or freely, but always respecting the capacity constraint. Therefore, at some point one of the two subsets could still contain potential elements while the other does not. No repair procedure is required, but the final solution could still be improved adding objects taken from $B \setminus (x \cup y)$.

SCP Given two solutions, that are subset of columns, the recombined solution z could be initialised as

$$z := x \cap y$$

and augmented choosing columns from $x \setminus z$ and $y \setminus z$, randomly or greedily, alternatively or freely. It makes sense to avoid choosing redundant columns. At some point, it is possible for one of the two subsets to contain potential elements while the other does no longer. No repair procedure is necessary, but some redundant columns could be removed with a destructive procedure.

10.3 Path relinking

The **Path Relinking** procedure, proposed again by Glover in 1989, is generally used as a final intensification procedure, more than as a standalone method. The idea is to take pairs of solutions and try to combine them, in a way that is intrinsically different from Scatter Search, using exchanges moves, instead of construction moves.

Given an exchange heuristic and a neighbourhood N, a **reference set** R is built by collecting the best solutions generated by the auxiliary heuristic. These promising solutions are called *elite solutions*. For each pair of solutions x and y in the reference set R, a path from x to y is built in the search space using the neighbourhood N applying to $z^{(0)} = x$ the auxiliary heuristic. Instead of choosing at each step the best neighbour solution, the heuristic chooses the closest one to the destination y:

$$z^{(k+1)} := \arg \min_{z \in N(z^{(k)})} d(z, y)$$

where d is a suitable metric function on the solutions (usually, the Hamming distance); in case of equal distance, the choise optimises the objective function f. Starting from the obtained solution $z^{(1)}$, the process is repeated, exactly as in a local search algorithm: the neighbourhood of the current solution is explore to select the element that is closest to the final solution. In a connected search graph, a path $x \rightsquigarrow y$ certainly exists. In particularly complex cases, it is possible, however, that no path from x to y allows to decrease the Hamming distance from y at each step. In these cases the method fails (actually, such a path always exists visiting unfeasible subsets: whether this makes sense must be discussed case by case). While building this path, the best solution found is saved:

$$z^*_{xy} := \arg \min_k f(z^{(k)})$$

possibly improved by an exchange heuristic and, if the result is not already in the reference set R and is better than the worst reference solution, R is updated. As in *SS*, duplicate solutions are forbidden.

Notice that the path thus built explores worsening solutions without the risk of a cyclic behaviour, because the Hamming distance of $z^{(k)}$ from the final solution y is strictly decreasing step by step. The length of each path is also clearly limited. Moreover, if we decide to visit also unfeasible subsets, the monotone decrease of the Hamming distance guarantees that there is no risk of getting lost in the unfeasible region: sooner or later feasibility will be regained (in the worst case, when reaching the final solution). The unfeasible solutions visited are not useful in themselves, but allow to open the way to improvements.

10.3.1 General scheme of Path Relinking

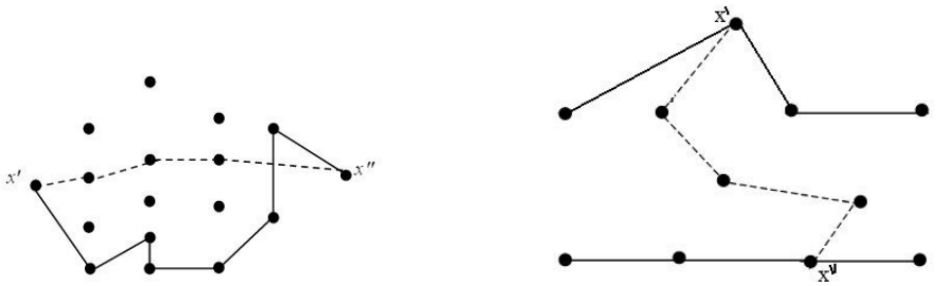
The general scheme of path relinking is reported in Algorithm 21. Given a starting population of locally optimal solutions (generated as in *SS*), the algorithm checks all the pairs of solutions (x, y), starting from x and building a path to y exploring the neighbourhood of the current solution z, finding the neighbour with smallest distance from y (and best objective value, to break ties), and moving to that solution. The best solution found along the path is improved with steepest descent and saved. If the result is not in the reference set, it is tested for insertion in the reference set.

The paths explored in this way clearly intensify the search, because they connect good solutions (as shown in Figure 10.1 (a)). Since, in general, they are different from the path followed by the exchange heuristic, they can also diversify the search. This is especially true

Algorithm 21 Path Relinking Pseudocode

```
1:  procedure PATH RELINKING(I, P)
2:      while P ≠ ∅ do
3:          R := ∅
4:          for each x ∈ P, y ∈ P \ {x} do          ▷ Recombine to build new population
5:              z := x
6:              z* := x
7:              while z ≠ y do                      ▷ Build a path from x to y
8:                  Z := arg min_{z'∈N(z)} d(z', y)
9:                  z̃ := arg min_{z'∈Z} f(z')
10:                 if f(z̃) < f(z*) then
11:                     z* := z̃
12:                 end if
13:                 z := z̃
14:             end while
15:             if z* ∉ P then
16:                 z* := SteepestDescent(I, z*)    ▷ Improve the best solution from the path
17:                 R := R ∪ {z*}
18:             end if
19:         end for
20:         P := Rk                                 ▷ Update the population
21:     end while
22:     return (x*, f(x*))
23: end procedure
```

when the extreme solutions x and y are far away, as shown in Figure 10.1 (b). The method does not provide a clear tuning parameter to control these aspects.



(a) Intensification of the search. (b) Diversification of the search.

Figure 10.1: Execution of Path Relinking.

Variants of Path Relinking

There are several variants of Path Relinking. They usually modify rather secondary aspects of the method: each other:

- **backward path relinking**: the path is not built from x to y, but from y to x (of course, if all ordered pairs are considered, the method applies both the forward and the backward path relinking algorithm).

- **back-and-forward path relinking**: this variant builds both paths, as in Algorithm 21.
- **mixed path relinking**: the path is built making alternate steps from each extreme, and updating the destination: the first step moves from x towards y, finding $z^{(1)}$; the second step moves from y towards $z^{(1)}$, finding $z^{(2)}$; the third step moves from $z^{(1)}$ towards $z^{(2)}$, finding $z^{(3)}$; and so on.
- **truncated path relinking**: instead of building the whole path, the exploration terminates after a few steps (this happens when knowledge about the problem suggests that solutions far away from x and from y will not be promising);
- **external path relinking**: instead of moving from x to y, the algorithm still uses y as a reference solution, but moves in the *opposite* direction, from x far away from y, maximising the distance (this happens when knowledge about the problem suggests that good solutions tend to be far away from each other).

10.4 Genetic algorithms

The **Genetic Algorithms** (*GA*) were originally proposed by Holland in 1975, but they now form a wide family of methods. Such methods belong to the class of recombination algorithms that make a heavy use of randomisation. A basic feature of genetic algorithms is that they are *encoding-based*, meaning that they replace solutions with **encodings**.

10.4.1 Encodings

The idea of encoding-based algorithms is to define a function that transforms each solution in a given solution space into a more compact object, named encoding or representation, that belongs to a suitable space, through a procedure known as **encoding operation**. Figure 10.2 shows a typical case in which the encodings are strings of bits.

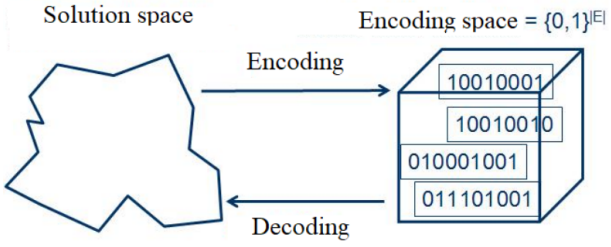


Figure 10.2: Representation of solution encoding.

The recombination algorithms of this family manipulate encodings, instead of solutions. This means that after applying their operators have, the representation has to be **decoded**, in order to retrieve the subset represented by each encoding, and determine whether it is a feasible solution and what is its objective function value.

The distinction between solutions and encodings is not actually clear-cut. Any algorithm that manipulates solutions must represent them with a suitable data structure. These data structure can obviously be considered as encodings themselves. So, the difference between solution and encoding is often vague.

We know from Section 2.5.4 that problems have relations, that can be exploited to solve one by attacking another. Historically, the original purpose of the encoding-decoding process was to

introduce a level of **abstraction** that would allow to transform several different problems into a more general one, that could be attacked by simple and general techniques, instead of studying and solving each problem separately. It was a really ambitious idea that, unfortunately, does not work, at least in its basic form.

10.4.2 General scheme of the genetic algorithm

The genetic algorithm (see the pseudocode in Algorithm 22) is based on the typical idea of all recombination heuristics, to manipulate a population of solutions. The initial population $X^{(0)}$ is obtained, as in the other recombination algorithms, randomly, with constructive heuristics or metaheuristics, possibly improved by exchange heuristics or metaheuristics. Then, it is iteratively subjected, generation after generation, to three fundamental steps:

1. **selection**: generate a new population starting from the current one;
2. **crossover**: recombine subsets of two or more individuals to generate new individuals;
3. **mutation**: modify single individuals.

Each phase can be implemented in different ways, that characterise the specific algorithm. Another fundamental feature that characterises the specific genetic algorithm is the encoding that generates the objects manipulated by the three basic phases.

Algorithm 22 Genetic algorithm Pseudocode

```
1:  procedure GENETICALGORITHM(I, X^(0))
2:      x* := arg min_{x∈X^(0)} f(x)
3:      for g = 1 to n_g do
4:          X^(g) := Selection(X^(g−1))
5:          X^(g) := Crossover(X^(g))
6:          x_c := arg min_{x∈X^(g)} f(x)
7:          if f(x_c) < f(x*) then
8:              x* := x_c
9:          end if
10:         X^(g) := Mutation(X^(g))
11:         x_m := arg min_{x∈X^(g)} f(x)
12:         if f(x_m) < f(x*) then
13:             x* := x_m
14:         end if
15:     end for
16:     return (x*, f(x*))
17: end procedure
```

10.4.3 Features of a good encoding

The performance of a genetic algorithm critically depends on the encoding adopted to represent the solutions of the problem. The following properties should be satisfied by a good encoding (with decreasing importance):

1. each solution should have at least an encoding, different from that of any other solution, in a one-to-many, or at least one-to-one, relation (otherwise, there would be unreachable solutions);

2. vice versa, each encoding should correspond to a feasible solution (otherwise, the population would include useless individuals, that do not represent solutions, wasting time and space);

3. each solution should correspond to the same number of encodings (otherwise, some solutions would be unduly favoured: since mutation and crossover are stochastic operators, the number of associated cases affects the probability to obtain each solution);

4. the encoding and decoding operations should be efficient procedures (otherwise, the overall algorithm will be inefficient);

5. small modifications to the encoding should induce small modifications to the solutions; this property is known as **locality**: it allows to tune the operators controlling the amount of intensification and diversification associated to the mutation and crossover operators (otherwise, one cannot control whether they are "strong" or "weak").

These conditions depend very much on the specific problem considered. Farewell, abstraction!

The first property listed above is nearly fundamental: unless a solution is clearly dominated, it must have an encoding. The second property (all encodings should represent a solution), on the contrary, is far from being general. On the contrary, it is quite typical that encodings produce unfeasible subsets.

We remind that the feasible region X consists of subsets of the ground set B. Therefore, it is a subset of its power set 2^B, that contains all subsets of B. The subsets that are not in X are unfeasible. More specifically, they can be unfeasible because they violate *quantitative* constraints, or *structural* constraints. For example, a solution of the *KP* is unfeasible because its volume exceeds the capacity of the knapsack; this is a quantitative constraint. A solution of the *BPP* can be unfeasible for the same reason (the volume of the objects assigned to a container exceeds its capacity) or because an object is assigned to multiple containers; this is a structural violation. Quite often, the quantitative constraints are easier to repair when violated (for example, removing some objects from the knapsack or moving them to another container). Therefore, they can be relaxed, and the corresponding subsets can be considered acceptable during the execution of an algorithm. Figure 10.3 shows the distinction between the feasible region X, denoted as "feasible solution space", a larger *solution space* that includes both the feasible solutions and subsets that violate only the simpler quantitative constraints (often called "unfeasible solutions"), and the remaining part of 2^B, that includes the strictly unfeasible subsets, not denoted as solutions. For example, in the *BPP*, an assignment of objects to containers can be a feasible solution (if it respects the capacities of the containers and assigns each object exactly to one container), an unfeasible solution (if it assigns each object exactly to one container, but exceeds some capacities) or an unfeasible subset (if some objects are not assigned, or assigned to multiple containers).

The concept of unfeasible solution is an abuse of expression, but it makes sense, because it is possible to build encodings that guarantee to respect some constraints, but not all of them. The corresponding subsets will be solutions, even if not necessarily feasible solutions.

It must be noticed that the distinction between quantitative and structural constraints is largely conventional. In fact, structural constraint can always be reinterpreted as quantitative by introducing suitable auxiliary functions. For example, the number of containers to which

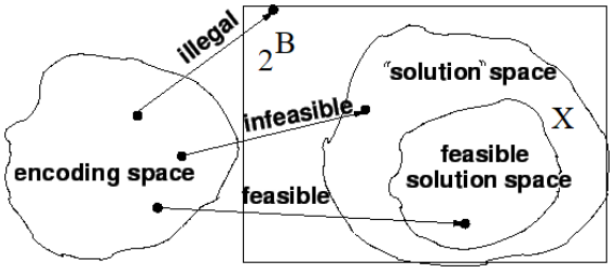


Figure 10.3: Representation of the map between encoding space and power set of the ground set.

each object is assigned is a function that should always assume value 1: if it is too small or too large, the constraint is violated. However, there is usually an intuitive distinction between a constraint that is naturally described as quantitative and a constraint that is modelled as such with some additional function.

In the following, we consider the three most common ways to define an encoding for the solutions of a combinatorial optimization problem.

Encodings: the incidence vector

Since the solutions of a combinatorial optimization problem are subsets of a ground set, the most natural kind of encoding is the incidence vector, that is a binary vector $\xi \in \mathbb{B}^{|B|}$ defined as

$$\begin{cases} \xi_i = 1 & i \in x \\ \xi_i = 0 & i \notin x \end{cases}$$

Let us consider some examples, and briefly discuss whether in each case the basic properties listed above are satisfied.

In the *KP*, a generic binary vector corresponds to a set of objects. It is not necessarily feasible, because its total volume could exceed the capacity. One could define it as an unfeasible solution, anyway.

In the *SCP* a generic binary vector corresponds to a set of columns. In general, it could leave uncovered rows. This looks more like the violation of a structural constraint, even if the number of columns covering a row can, of course, be measured to check if it is violated by defect.

In the *PMSP* and in the *BPP*, a generic binary vector corresponds to a set of assignments of tasks (objects) to machines (containers). In both problems, the structural constraint that exactly one assignment must be made for each task, or object, could be violated. In the *BPP*, also the quantitative constraint related to the capacity of each container could be violated.

In the *TSP*, a generic binary vector corresponds to a subset of arcs. In general, it is quite unlikely that such arcs form a Hamiltonian circuit, so most vectors will not correspond to solutions.

In the *CMSTP* (*VRP*) a generic binary vector corrsponds to a set of edges (arcs), but most of the time it will not form a spanning tree (spanning set of disjoint circuits based on the depot); if it does, it will probably exceed the capacity of some branch, or vehicle.

Encodings: symbol strings

In many problems, the ground set is partitioned into disjoint components

$$B = \bigcup_{c \in C} B_c \text{ with } \forall c \neq c' \; B_c \cap B_{c'} = \emptyset$$

and the feasible solutions must contain exactly one element from each component

$$\forall c |x \cap B_c| = 1$$

The components can correspond to objects, tasks, Boolean variables, vertices, nodes, etc... so that the solution must contain one assignment for each object, task, etc...

An encoding for the solutions of these problems can be obtained defining, for each index $c \in C$, an alphabet of symbols describing the possible elements of each component B_c. For example, in the *BPP*, the components can be associated to the objects and for each object i a symbol can be defined to describe each possible container in which the object can be put. In this case, all components have the same alphabet, but if an object were compatible only with some containers, the alphabets could be different from each other. Given the alphabets, a string of symbols ξ can be built including a symbol from each alphabet: $\xi \in B_1 \times \cdots \times B_{|b|}$, such that

$$\forall c, \xi_c = \alpha \implies x \cap B_c = \{(c, \alpha)\}$$

In other words, each symbol in the string describes an element of the solution. A solution for the *BPP*, for example, can be represented by the sequence of names of the containers that include the various objects, in the same order of the objects. For other examples, see the solved exercises at the end of the chapter.

Encodings: permutations of a set

Another possible encoding for solutions is given by permutations of a given set (that is not necessarily the ground set). In the *TSP*, for example, the feasible solutions are Hamiltonian circuits (that is, subsets of arcs), but they can also be seen as permutations of nodes. A permutation of the node set, therefore, is a natural encoding for the solutions of this problem.

Permutations can be used for every problem whose solution is a partition, by applying the **order-first split-second** method described in Section 8.5. This representation is often biased, in that different solutions correspond to different numbers of encodings, so that some are favoures with respect to other ones.

Permutations can be used also for general problems, in which solutions are not explicitly ordered, by using an auxiliary constructive algorithm. Each step of the algorithm selects an element of the ground set that is an acceptable extension of the current set. The selection rule can be based on the given partition. In the simplest case, given a permutation of the ground set, the algorithm selects the first acceptable extension in the partition. For example, in the *KP* a given permutation of objects can be used to decide which objects to insert in the knapsack, skipping those that exceed the residual capacity. Every feasible solution has such an encoding (actually, many). Each encoding corresponds to a feasible solution. The number of encodings that correspond to the same solution is in generally not uniform. Just permute the objects in a given solution and the objects out of it: the number of encodings is $|x|! \cdot (|O \setminus x|)!$, that favours the solutions with cardinality closer to $|O|/2$.

In more refined cases, the selection rule can combine a greedy criterium and a permutation. For example, several constructive heuristics for the *TSP* first chose a node to insert, and then a position in the circuit. The first choice can be based on a given permutation, while the second is based on the objective function, as usual. In this case, each encoding corresponds to a feasible solution, though probably some encodings correspond to the same solution, and their number is not uniform. Moreover, it is not guaranteed *a priori* that all solutions have an encoding. On the other hand, combining the representation of solutions with the selection could be an advantage, focusing the search on better solutions. If we could prove (or at least reasonably assume) that

at least one globally optimal solution have such an encoding, that could be justify the adoption of this encoding. In these cases, the choice of the set to permute is not obvious: when there is more than one, we could permute each of them (for example, in the *BPP* objects or containers).

10.4.4 Selection

We now consider the first of the three phases of the method. At each generation, the previous population $X^{(g-1)}$ must be transformed in the current population $X^{(g)}$. This is done by extracting $n_p = |X^{(g)}|$ individuals from the current population $X^{(g-1)}$:

$$X^{(g)} := Selection(X^{(g-1)})$$

The extraction is based on a **random mechanism** following two basic principles:

1. an individual can be extracted more than once;
2. better individuals are extracted with higher probability.

$$\varphi(\xi) > \varphi(\xi') \implies \pi_\xi \leq \pi_{\xi'}$$

where $\varphi(\xi)$ is the quality of a solutions (called **fitness**).

Let us represent the decoding process, that turns an encoding back into the original solution as a function $x(\xi)$. For a maximization problem, commonly

$$\varphi(\xi) = f(x(\xi))$$

while for minimization problems

$$\varphi(\xi) = UB - f(x(\xi))$$

where $UB \geq f^*$ is a suitable upper bound on the optimum. Of course, the choice of UB can influence the values of the probabilities. Notice that it is also possible to define $\varphi(\xi) = UB/f(x(\xi))$ (as in the *Ant System*), but this produces numerical issues for problems in which f has large values and distorts in a nonlinear way the relation between objective value and fitness.

Three main mechanisms have been proposed to map fitness to probability.

Proportional selection

The original scheme proposed by Holland in 1975 assumed a probability *proportional to the value of fitness*:

$$\pi_\xi = \frac{\varphi(\xi)}{\sum_{\xi \in X^{(g-1)}} \varphi(\xi)}$$

This is named **roulette-wheel selection** (or spinning wheel selection). In order to implement this scheme, one must compute the fitness for every individual in the population, build the partial sums

$$\Gamma_i = \left(\sum_{k=1}^{i-1} \pi_{\xi_i}, \sum_{k=1}^{i} \pi_{\xi_i} \right]$$

which takes $\mathcal{O}(n_p)$ time, then extract a random number $r \in U(0,1]$ and choose an individual i^* such that $r \in \Gamma_{i^*}$, which takes $\mathcal{O}(\log n_p)$ time. The idea is that number r "lands" in one of the intervals, that is selected.

Figure 10.4 shows an example in which 4 individuals compose a population and random number $r = 0.78$ points to the interval containg r, that is A_4.

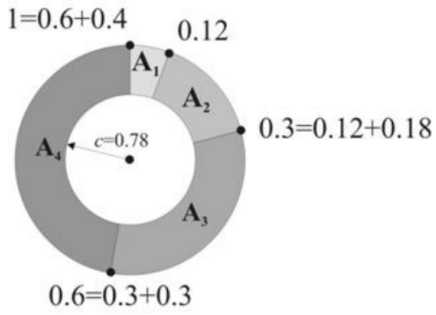


Figure 10.4: Roulette wheel selection.

Rank selection

The proportional selection mechanism has two problems. The first one is **stagnation**: in the long term all individuals tend to have a good fitness, and therefore similar selection probabilities. The second problem is that when the population contains some bad individuals and a lot of *very* bad individuals (for example, in the first generations after a random initialisation), the difference between the two is so large that the selection quickly removes the very bad individuals getting stuck with a bad population. In other words, sometimes the differences between probabilities are too small, and sometimes too large. The scheme gives no control on this aspect.

A way to avoid these problems is to introduce a scheme guaranteeing that two probabilities are neither too similar nor too different from each other. This is obtained creating a fixed profile of decreasing probabilities, exactly as for the selection mechanism in *GRASP*. The **rank selection** method consists in sorting the individuals by nondecreasing fitness

$$X^{(g)} = \{\xi_1, \cdots, \xi_{n_p}\} \text{ with } \varphi(\xi_1) \leq \cdots \leq \varphi(\xi_{n_p})$$

and assigning to each individual a probability linearly increasing with its position

$$\pi_{\xi_j} = \frac{k}{\sum_{k=1}^{n} k} = \frac{2k}{n_p(n_p-1)}$$

This can be computed in $\mathcal{O}(n_p)$ time, as sorting the values of fitness is not actually required: given a random number r, one can compute the corresponding selected index $k(r)$ and apply the linear-time algorithm that finds the k-th element in a set of numbers. The method is slower than proportional selection (due to the multiplying constants), but has the advantages described above. In addition, it does not require to define a fitness function: only a total order relationon solutions must be introduced.

Tournament selection

An efficient scheme consists in extracting n_p random subsets $\bar{X}_1, \cdots, \bar{X}_{n_p}$ of size α, and select the best individual from each subset:

$$\xi_k := \arg\max_{\xi \in \bar{X}_k} \varphi(\xi), k = 1, \cdots, n_p$$

This is done in time $\mathcal{O}(n_p\alpha)$. The parameter α tunes the strength of the selection: $\alpha \approx n_p$ it favours the best individuals, while $\alpha \approx 2$ leaves large chances to the bad individuals, always removing only the worst.

All selection schemes discussed above admit an **elitist variant**, which includes in the new population the best individual of the current one. Of course, also the nonelitist variants save the best individual to return it in the end, but they do not feed it to the other operators.

10.4.5 Crossover

The crossover operator derives from a biological methaphor: the chromosomes of living beings, that are sequences of DNA encoding the features of the individual, sometimes cross each other and "exchange" part of their material during reproduction. In the end, this gives rise to two different individuals that possess part of the genetic code of each of the two parents. This can be easily simulated with the encoding of solutions, and implements the characteristic concept of recombination algorithms, that solutions should be recombined, taking the good parts of several ones in order to build better solutions.

The crossover operator combines $k \geq 2$ individuals generating other k individuals. Most of the time, $k = 2$, but this is not required. Its basic variant (**simple crossover**) extracts a random position p in the encoding, usually with uniform probability, splits the encodings of the two individuals in two parts corresponding to position k, and switches the parts. In Figure 10.5 (a), the two *parents* are split between the fifth and sixth symbol and two *children* encoding are built merging the first part of the black individual with the second part of the red individual and the first part of the red individual with the second part of the black individual.

(a) Simple crossover (b) Double crossover

Figure 10.5: Crossover.

This behaviour can be generalized introducing multiple splits. For example, the **double crossover** randomly extracts two positions, splits the encodings into three parts and exchange the extreme parts of the encodings of the two parents individuals, generating two children as shown in Figure 10.5 (b).

Generalizing, the **α-points crossover** extracts α points at random with uniform probability and splits the two encodings into $\alpha+1$ parts, exchanging the odd parts of the encodings between the two individuals.

Bias

For small values of α, this kind of crossover suffers from a positional bias: symbols that are close in the encoding tend to remain close. For example, consider a *KP* with the incidence vector representation (strings of n bits equal to 1 if the object is inside the knapsack, 0 if it is outside). Suppose that one individual includes objects i and j, whereas another does not include them. The children generated by crossover can include both objects, none of them, or only one. The children of this last group, however, require some of the α positions selected (an odd number, to be precise) to fall between indices i and j. If $j = i+1$, this is very unlikely; if, on the contrary, $j = n$ and $i = 1$, this is very likely. Therefore, some pairs of objects tend to stay together, whereas other pairs tend to be separated. Since the indices of objects are arbitrary, this is not justified.

To cancel this bias, one can adopt the **uniform crossover** scheme, that is less influenced by the biological analogy. This scheme first builds a random binary vector m, called **mask**, made of bits uniformly extracted from $\{0,1\}$ (so $m \in U(\mathbb{B}^n)$). Then, the mask is applied to the two parent individuals: if $m_i = 1$, the symbols in position i of the two individuals are exchanged; if $m_i = 0$, they remain unchanged.

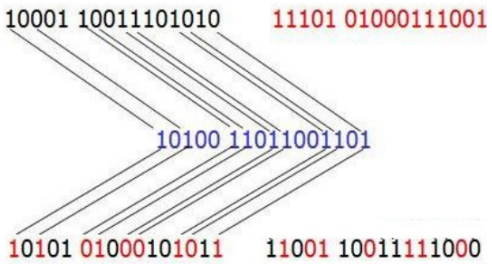


Figure 10.6: Uniform crossover.

Figure 10.6 represents this mechanism: the first position of the mask contains a 1, so the black and red bits in the first position are exchanged (in this case, they are both equal to 1, so nothing changes). The second position of the mask contains a 0, so no exchange occurs and the symbol in the second position of the two children is the same as in their parents, and so on. This destroys the positional bias because the fact of keeping two elements together or separated depends on the values of two random bits in the mask, and on their positions in the encodings.

Crossover versus Scatter Search and Path Relinking

The crossover operator is strictly related to the recombination phase of *SS* and *PR*. It certainly plays the same role as the characterising element of the recombination algorithms. There are, however, important differences:

1. the crossover recombines the symbols of the encodings, instead of
   - recombining elements of the solutions (*SS*)
   - exchanging elements in and out of intermediate solutions (*PR*)
2. the crossover operates on the whole population of current solutions, instead of a selected reference set R;
3. the crossover operates on random pairs of individuals, instead of methodically scanning all pairs of solutions of R;
4. the crossover generates a pair of new individuals, instead of
   - generating a single intermediate solution (*SS*)
   - visiting intermediate solutions and choosing one (*PR*)
5. the new individuals enter the new population, instead of becoming candidates for the reference set.

These differeces can be more or less pronounced, based on the encoding used and the implementation of the crossover.

10.4.6 Mutation

The mutation operator modifies an individual to generate a similar one by scanning the encoding ξ one symbol at a time and deciding, with probability π_m, whether and how to modify that symbol. Small values of probability π_m intensify the search, while large values diversify it.

The kind of modification depends on the encoding. For binary encodings, the only possible modification is a flip, replacing ξ_i with its complementary value $\xi_i' := 1 - \xi_i$, as shown in Figure 10.7.

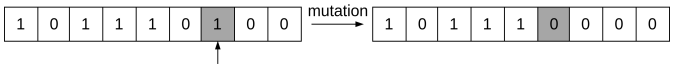


Figure 10.7: Mutation.

When the encoding is a string of symbols, usually each modified symbol ξ_c is replaced by another one, ξ_c', uniformly chosen at random in $B_x \setminus \{\xi_c\}$. In the case of permutations, things are more complex, because the basic structure of the permutation (one different element for each position) must be preserved. A typical approach is to select two random elements in the permutation and exchange them (*swap*), possibly reversing (or not) the stretch between them.

Mutation and exchange heuristics

The mutation operator are strong similarities with the moves of exchange heuristics. They play a similar role: applying minor modifications to a single solution. In fact, quite often mutation operators coincide with the operations used to define neighbourhoods. For example, if a *TSP* solution is represented by a permutation of nodes, exchanging two random positions and reversing the entire stretch inbetween is equivalent to a 2-opt exchange. Not reversing the stretch is equivalent to a swap of two nodes.

Also in this case, the relation can be more or less strict, based on the encoding used and the implementation of mutation, but some basic differences remain:

1. the mutation modifies an encoding, instead of a solution;
2. the mutation operates on random positions in random ways, instead of systematically exploring a set of moves;
3. the mutation operates on a random number of symbols, instead of a fixed number of elements in the solution.

10.5 The feasibility problem

So far, we have always avoided to operate on unfeasible subsets. Even in constructive heuristics, that usually require to handle such subsets in order to reach a final feasible solution, we have introduced the basic concept of acceptable extension, based on the principle that the current subset should not forbid (and possibly should guarantee) to reach a feasible solution. Checking whether or not a subset is feasible, or at least a step towards a feasible solution is a decision problem, that can be called the **feasibility problem**.

This basic requirement could actually be relaxed, provided that the possibility to get back to feasible solutions is not completely destroyed, but doing that implies facing a number of complex problems.

In recombination algorithms, in particular those that make heavy use of randomisation, the existence of unfeasible subsets becomes so important that one can no longer ignore it. We now discuss the problems that derive from accepting in a population what we have already defined as "unfeasible solutions", that is unfeasible subsets that satisfy some basic structural constraint, but violate easier quantitative constraints (keeping in mind that this distinction is largely conventional). The discussion can be applied also to exchange heuristics, relaxing the definition of neighbourhood to include unfeasible solutions and allow the algorithm not only to explore, but also to visit them.

The encoding functions of genetic algorithms are often not invertible, meaning that there are **feasible encodings**, that correspond to feasible solutions, and **unfeasible encodings**, that correspond to legal, but unfeasible solutions. The crossover and mutation operators often generate unfeasible encodings.

This implies several disadvantages:

1. inefficiency: computational time is lost handling useless subsets;
2. ineffectiveness: the algorithm explores less solutions;
3. design problems: the fitness function must be defined on unfeasible subsets, as well.

There are three main approaches to face these problems. The first one is to use special encodings and operators that allow to avoid or, at least, limit the problem. The second approach is to use repair procedures. The third one is to use penalty functions to accept infeasibility, but heavily discourage it.

10.5.1 Special encodings and operators

The idea is to investigate encodings that nearly always yield feasible solutions. This was already discussed while introducing the families of encodings. The incidence vector, for example, often yields unfeasible encodings, whereas the strings of symbols avoid some structural infeasibilities. Permutation encodings, using the order-first split-second approach as a decoding procedure for partition problems such as *CMSTP* and *VRP* guarantee feasibility. Permutation encodings with an auxiliary constructive heuristic to decode them is common in scheduling problems, such as the *PMSP*.

Alternatively, crossover and mutation operators can be designed so as to maintain feasibility. For example, in the *TSP* mutation operators can simulate k-opt exchanges. These methods tend to closely approximate the operators on which the concept of neighbourhood is based. This has clear advantages, as it focuses on the specific problem, while giving up the original idea of abstraction purported by classical genetic algorithms.

10.5.2 Repair procedures

A repair procedure $x_R(\xi)$ is an algorithm that receives an encoding ξ such that the decoding procedure yields an unfeasible subset $x(\xi) \notin X$, and returns a feasible solution $x_R \in X$. Such a procedure can be applied to each unfeasible encoding $\xi \in X^{(g)}$. In some approaches, the repaired solution is encoded again, obtaining $\xi(x_R(x(\xi)))$, that replaces ξ in $X^{(g)}$: thus, the population is maintained fully feasible.

In other approaches, the unfeasible encoding ξ is kept in the population and the repaired solution $x_R(\xi)$ is used only to update the best known result, not to modify the population, that includes both feasible and unfeasible encodings.

The first family of approaches introduces a strong bias in favour of feasible encodings, in particular of the specific feasible solutions that are easier to obtain with the repair procedure. This probably reduces the diversity of the population, possibly generating many duplicate individuals. On the other hand, it intensifies the search.

10.5.3 Penalty functions

The third approach to deal with unfeasibility is to introduce penalty functions. In this case, instead of avoiding unfeasibility or translating it into feasibility, one simply accepts unfeasibility while highly discouraging it through the selection operator. If the objective function is extended to unfeasible subsets $x \in 2^B \setminus X$ (a point already discussed in Section 2.1.3), the fitness function $\varphi(\xi)$ automatically extends to any encoding. The problem is that many unfeasible subsets have a fitness larger than the optimal solution. For example, a solution of the *KP* that exceeds the capacity of the knapsack often has a value higher than a solution that respects it. Hence, the selection operator tends to favour such unfeasible subsets!

To avoid this situation, the fitness function must combine the objective value $f(x(\xi))$ with a measure of unfeasibility $\psi(x(\xi))$ such that

$$\begin{cases} \psi(x(\xi)) = 0 & x(\xi) \in X \\ \psi(x(\xi)) > 0 & x(\xi) \notin X \end{cases}$$

Quite frequently, the constraints of a problem are expressed by equalities or inequalities. In these cases, $\psi(x)$ can be defined as a weighted sum of the violations of such constraints. The problem now is how to tune the weights, in particular whether to make them fixed, variable according to a fixed profile, or adaptively variable depending on the results of the algorithm.

Definition of the fitness function

The simplest way to combine objective and penalties is the **absolute penalty**, that is mainly suitable for the rank and tournament selection schemes, since they only require to compare pairs of solutions (and possibly sort them). Given two encodings, ξ and ξ', if both are feasible, the one with smaller f is better; if exactly one is feasible, the feasible one is better than the other; finally, if none is feasible, the one with the smaller value of the penalty ψ is better than the other. The proportional selection, that requires to assign a numerical value to the penalty, can be implemented considering only the feasible encodings and their objective values. If none exists, it can be implemented using the values of the penalty function.

Another approach is the use of a **proportional penalty**, where the fitness is a linear combination of the objective (with a positive sign for maximisation problems and a negative sign for minimisation ones) and the penalty (with a negative sign):

$$\varphi(\xi) = \begin{cases} f(x(\xi)) - \alpha\psi(x(\xi)) + UB & \text{for maximisation problems} \\ f(x(\xi)) - \alpha\psi(x(\xi)) + UB & \text{for minimisation problems} \end{cases}$$

where the offset UB guarantees that $\varphi(\xi) \geq 0$ for all encodings. Of course, its specific value influences the results: the larger it is, the smoother the difference of probabilities between good and bad solutions.

Since objective and penalty have different units of measure, the coefficient α is a conversion coefficient, which must be tuned somehow, beside being a relative weight of the unfeasibility with respect to the cost. Moreover, since several constraints can be given, the penalty function

$\psi(\cdot)$ itself can be a linear combination of specific terms, with weights that describe both the conversion between units of measure and the relative importance of the different terms.

A third approach, that does not require coefficients, adopts a **penalty based on repair**: the fitness of an unfeasible encoding is given by the fitness of the feasible solution obtained repairing the unfeasible set that corresponds to the encoding:

$$\begin{cases} f(x_R(\xi)) & \text{for maximisation problems} \\ UB - f(x_R(\xi)) & \text{for minimisation problems} \end{cases}$$

In general, in fact, $f(x_R(x(\xi))) \geq f(x(\xi))$.

Proportional penalty functions: weight tuning

Experimentally, it is better to assign penalties the smallest effective value. If the penalty is too large, in fact, the search is confined within a part of the feasible region. On the other hand, if the penalty is too small, once the search enters the unfeasible region of the solution space, it might never get out of it, because many unfeasible subsets have very good values of fitness. So, too few feasible solutions are found.

A method proposed by Glover, called *strategic oscillation*, suggests that the search should repeatedly cross the boundary between the feasible and the unfeasible region, oscillating between feasible and unfeasible solutions. This is thought for problems whose search space has a complex shape, with many adjacent feasible and unfeasible solution, so that visiting the unfeasible ones can be the quickest and easiest way to to reach other feasible solutions, whereas paths fully made of feasible solutions should be long and devious.

The problem is deciding how much to step inside the unfeasible region: this requires a good tuning of parameter α (and of the different weights of the components of the unfeasibility $\psi(x)$, if there are many).

The **dynamic methods** modify parameter α during the execution obeying a fixed rule in time, as the temperature decrease in *SA*. In that case, the aim was to diversify the search initially and intensify it later. The same aim can be obtained progressively increasing α, so that at first the search can freely visit the unfeasible region, and, as time goes by, it gets back to the feasible region.

The **adaptive methods** tune α adaptively depending on the current situation and the recent results, as the tenure in the adaptive *TS*. The parameter, could be increased when unfeasible encodings dominate the current population, in an attempt to reduce them and avoid losing time; α could be decreased when feasible encodings dominate, and the visit of unfeasible encoding could introduce a useful diversification.

Finally, the **evolutionary methods** encode the value of α in each individual, so that different individuals behave in different ways with respect to unfeasibility and evolve different behaviours generation after generation. The method, then, does not refine only the solutions, but also the parameters of the algorithm. The capacity of evolving is sometimes called *evolvability*).

10.6 Other recombination metaheuristics approaches

10.6.1 Memetic algorithms

An approach strongly related to genetic algorithms is provided by the **memetic algorithms**. These are inspired by the concept of *meme*, coined by Richard Dawkins in 1976 and extended

to algorithmics by Moscato in 1989. The idea is that the cultural evolution of ideas follows, somehow, lines similar to the evolution of genetic characters, with operators similar to selection, crossover and mutation, but with a basic difference: individuals can *change* their ideas, besides inheriting them. This yields a sort of Lamarckian evolution in which the memes are not only selected, destroyed or modified blindly, but can also be modified with a purpose some reason inside the individuals who propagate them.

Outside the metaphor, memetic algorithms combine *genotypic* operators that manipulate the encodings, such as crossover and mutation, and *phenotipyc* operators that manipulate the solutions, such as local search. In short, the solutions are improved with exchanges and reencoded. Of course, this can be done in many different ways, with parameters that determine how to apply local search, how often (at every generation or after a while), to which individuals (all of them, the best ones, the most diverse ones), for how long (until a local optimum is found, stopping in advance, proceeding after local optima) and with what method (steepest descent, *VNS*, *TS*, and so on).

10.6.2 Evolution strategies

The **evolution strategy**, proposed by Rechenberg and Schwefel in 1971, was not originally designed for combinatorial optimization problems, but to engineering design problems, such as the design of engines. These problems have a number of real parameters and are really hard to solve using optimization algorithms. The idea was to start from a possible design, estimate by simulation how good it is, and modify it slightly to see if a better model could be found. This can be done not necessarily with a steepest descent algorithm, and even allowing worse designs to get out of local optima. The idea was later extended to combinatorial optimisation.

The evolution strategy encodes solutions into real vectors (vectors of real numbers). It maintains a population of μ such individuals (typically of a smaller size than the typical population of genetic algorithms; in fact, originally, $\mu = 1$). These individuals generate λ candidate descendants with a mutation operator that adds to each real number in the given encoding a disturbance, that is a random value distributed according to a normal distribution with average 0 and variance σ:

$$\xi' := \xi + \delta \text{ with } \delta \in N(0, \sigma)$$

The new vectors ξ' form a population of descendants; the original individuals and the descendants compete to build the population of the next generation:

- in the (μ, λ) strategy the best μ descendants replace the original population, even if some are dominated;
- in the $(\mu + \lambda)$ strategy the best μ individuals overall (predecessors or descendants) survive in the new population.

The crossover operator was not used in the original method, but it can be added as a postprocessing procedure.

10.7 Exercises

Note: Some of the following exercises require to propose an encoding for combinatorial optimisation problems, and to discuss its possible advantages and disadvantages, along the lines introduced in Section 10.4.3. In principle, the binary encoding, that is a vector of bits associated with the elements of the ground set B, is always possible. Proposing it and showing it is a good encoding is an acceptable answer. It is insufficient to simply propose it and state that it is bad, even explaining why, because the reason is nearly always the same: most encodings are unfeasible. In the following answers, however, we will discuss all the available options, for the sake of completeness and to show that in general different answers are acceptable.

10.7.1 Exercise 1

Propose an encoding for the *SCP* problem, discussing whether it satisfies well or not the basic requirements listed in the lecture notes.

Solution Given an instance of the *SCP*, with m rows and n columns, the binary encoding is, of course, always possible for any solution. Most of the time, a binary vector corresponds to an unfeasible solution, because it leaves uncovered rows. It is therefore not a very promising encoding.

Strings of symbols do not seem to be a particularly natural encoding.

A permutation of columns could be used with the following encoding, based on an auxiliary constructive heuristic: instead of selecting at each step the column that minimises the ratio of the cost and the number of additional covered rows, we can select the first column in the partition that covers at least one additional row. This is equivalent to select the first column in the partition and add it to the solution if it covers additional rows or simply removing it, if it does not. This always provides a feasible solution (if any exists). It could produce redundant solutions, that could be processed by a destructive postprocessing algorithm. The number of solutions that correspond to each encoding is probably different (proving it would require an example, but solutions of a size closer to $n/2$ tend to correspond to more encodings, thanks to the permutations of the internal and the external columns). The encoding of a solution takes $O(n)$ time. The decoding procedure take $O(mn)$ time, because each column is evaluated marking the covered rows and, if accepted, updating the marks.

10.7.2 Exercise 2

Propose an encoding for the *Max-SAT* problem, discussing whether it satisfies well or not the basic requirements listed in the lecture notes.

Solution A binary vector on the pairs (variable,value) is not reasonable, as most of the encodings correspond to unfeasible solutions, violating the structural constraint that the assignment must be complete and consistent.

A string of n Boolean values, one for each logical variable, providing the corresponding logical value creates a one-to-one correspondence between solutions and encodings. It can be equivalently seen as a binary vector, or a string of binary symbols. Both the encoding and decoding procedures take linear time in the number of variables.

Permutations do not seem to be a particularly natural encoding.

10.7.3 Exercise 3

Propose an encoding for the *PMSP* problem, discussing whether it satisfies well or not the basic requirements listed in the lecture notes.

Solution A binary vector on the pairs (task,machine) is not reasonable, as most of the encodings correspond to unfeasible solutions, violating the structural constraint that each task must be assigned to exactly one machine.

A string of n machine labels, one for each task, provides a one-to-one correspondence between solutions and encodings. Both the encoding and decoding procedures take linear time in the number of elements.

A permutation of tasks can be used with the following encoding, based on an auxiliary constructive heuristic: each task is assigned to the machine that yields the smallest completion time (with some additional rule to manage ties). This always provides a feasible solution. The number of solutions corresponding to each encoding is probably different (proving it would require an example. Let n be the number of tasks and m the number of machines. The decoding procedure takes $O(mn)$ time, because each task is assigned evaluating the completion time. The encoding of a solution is not trivial, because it is not obvious how to permute the tasks so as to obtain it. Is it actually required to encode solutions[2]?

10.7.4 Exercise 4

Propose an encoding for the *VRP* problem, discussing whether it satisfies well or not the basic requirements listed in the lecture notes.

Solution A binary vector is not reasonable, neither on the arcs nor on the pairs (node,vehicle): most of the encodings correspond to unfeasible solutions. In the former case, they do not form circuits, in the latter they do not assign nodes to exactly one vehicle.

A string of symbols associated to vehicles, one for each task, provides an encoding that satisfies the structural constraint that each node is assigned to exactly one vehicle, but it can violate the capacity of the vehicles. While computing the encoding of a solution is easy, the decoding procedure requires to solve a *TSP* on each vehicle. Doing it heuristically is a possibility.

A permutation of nodes can be used with the order-first split-second method to decode in time proportional to the number of arcs of the auxiliary graph (possibly $O(n^2)$, where n is the number of nodes). If a feasible solution compatible with the permutation exists, the method finds it. The encoding of a solution is not trivial: one can sort somehow the vehicles and the nodes assigned to each vehicle, but this does not guarantee that the corresponding solution is the original one. Is it actually required[3]?

[2] My answer is probably not, apart from the generation of the starting population, but I have not yet thought about this point.

[3] My answer is probably not, apart from the generation of the starting population, but I have not yet thought about this point.

Encodings based on auxiliary constructive heuristics can also be used, drawing inspiration from the *TSP* heuristics, but with a feasibility check on the capacity of the vehicles. These are not guaranteed to find a feasible solution, even if one exists, because the greedy choice of the insertion point could be incorrect. The complexity depends on the specific heuristic adopted. The encoding of a solution is not trivial, because it is not obvious how to permute the tasks so as to obtain it. The same question as above can be posed.

10.7.5 Exercise 5

Propose an encoding for the *TSP* problem, discussing whether it satisfies well or not the basic requirements listed in the lecture notes.

Solution A binary vector on the arcs of the graph is not reasonable, because most of the encodings violate the constraint of forming a Hamitonian circuit.

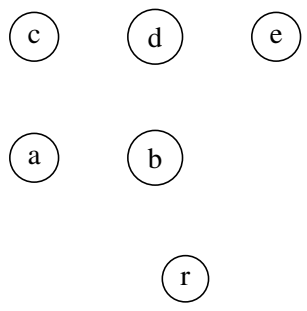
A string of symbols could associate to each node the index of the following one (or the previous one, equivalently). Unfortunately, this also does not guarantee to respect the structural constraints of the problem.

A permutation of nodes is the most natural encoding: in a complete graph, each encoding corresponds to a feasible solution and each solution corresponds to n encodings (the cyclic permutations) or to a single one, if we fix a conventional starting node. Encoding or decoding a solution takes linear time in the number of elements.

Another permutation that could be used is the permutation of the first n positive numbers, associating each node to its position in the Hamiltonian circuit. The properties are the same as for the previous permutation.

10.7.6 Exercise 6

Consider the following instance of the Capacitated Minimum Spanning Tree (*CMST*) problem, that is a complete graph (the edges are not reported for the sake of clarity) with 5 vertices of unitary weight $w_v = 1$ for all $v \in \{a,b,c,d,e\}$, a root vertex, capacity $V = 2$.



and the following cost matrix

	r	a	b	c	d	e
r	-	14	11	25	26	24
a	14	-	15	13	16	27
b	11	15	-	15	12	18
c	25	13	15	-	14	20
d	26	16	12	14	-	10
e	24	27	18	20	10	-

Given solutions

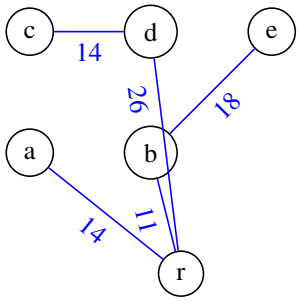
- $x' = \{(r,a),(r,b),(r,d),(b,e),(c,d)\}$
- $x'' = \{(r,a),(r,b),(r,e),(a,d),(b,c)\}$

apply the following recombination procedures:

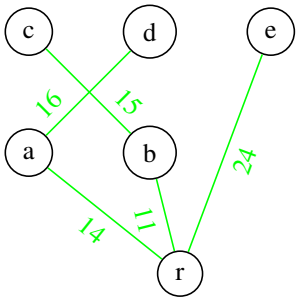
a) alternated greedy extraction based on Kruskal's algorithm (cheapest edge not inducing cycles or unfeasible trees) starting from x';

b) alternated random extraction with uniform probability assuming the pseudorandom number sequence $0.1, 0.6, 0.8, 0.5, 0.3$.

Solution The following picture represents the two given solutions.



$f(x') = 83$

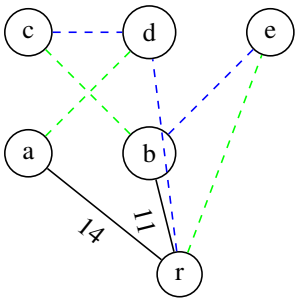


$f(x'') = 80$

Both recombination procedures suggest to start from $z^{(0)} = x' \cap x'' = \{(r,a),(r,b)\}$.

Part a) The alternate greedy recombination considers as acceptable all edges that produce forests (no cycles) whose trees have weight not larger than $V = 2$, and that belong, respectively, to $x' \setminus z^{(t)}$ in the odd iterations and to $x'' \setminus z^{(t)}$ in the even iterations. Consequently, the operations comply with the following sequence:

1. start from $z^{(0)} = x' \cap x'' = \{(r,a),(r,b)\}$;

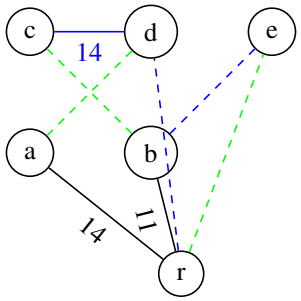


$z^{(0)} = x \cap x' \Rightarrow f(z^{(0)}) = 25$

2. all edges of $x' \setminus z^{(0)} = \{(r,d),(b,e),(c,d)\}$ are compatible with the constraints (forests with trees of weight $\geq V = 2$); add the cheapest one, that is (c,d);

Add	Evaluation
(r,d)	$\delta f = 26$
(b,e)	$\delta f = 18$
(c,d)	$\delta f = 14$

now $z^{(1)} = \{(r,a),(r,b),(c,d)\}$.

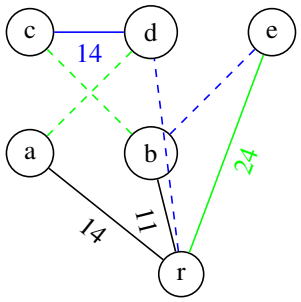


$z^{(1)} = z^{(0)} \cup \{(c,d)\} \Rightarrow f(z^{(1)}) = 39$

3. in set $x'' \setminus z^{(1)} = \{(r,e),(a,d),(b,c)\}$ only edge (r,e) is compatibile with the constraints, because (a,d) and (b,c) would yield trees of weight 3;

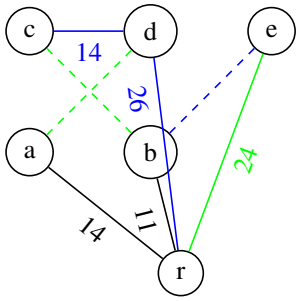
Add	Evaluation
(r,e)	$\delta f = 24$
(a,d)	Unfeasible (capacity)
(b,c)	Unfeasible (capacity)

now $z^{(2)} = \{(r,a),(r,b),(c,d),(r,e)\}$.



$z^{(2)} = z^{(1)} \cup \{(r,e)\} \Rightarrow f(z^{(2)}) = 63$

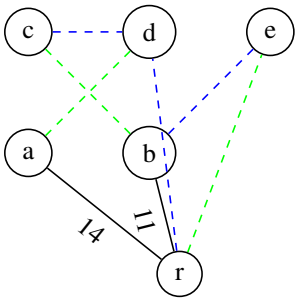
4. finally, in set $x' \setminus x = \{(r,d),(b,e)\}$ only edge (r,d) is compatibile with the constraints, because (b,e) closes a loop; now $z^{(3)} = \{(r,a),(r,b),(c,d),(r,e),(r,d)\}$ is a feasible solution.



$z^{(3)} = z^{(2)} \cup \{(r,d)\} \Rightarrow f(z^{(2)}) = 89$

Part b) The alternate random strategy extracts one edge from x' and one from x'' alternatively, selecting it based on the given pseudorandom number sequence. If this generates an incomplete solution, further edges from the overall graph must be added, possibly with a greedy constructive heuristic. In the specific case:

1. start from $z^{(0)} = x' \cap x'' = \{(r,a),(r,b)\}$;

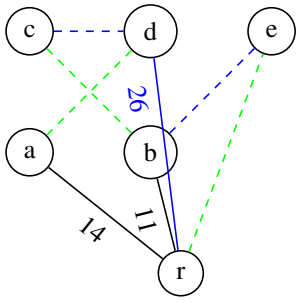


$z^{(0)} = x \cap x' \Rightarrow f(z^{(0)}) = 25$

2. in $x' \setminus z^{(0)} = \{(r,d),(b,e),(c,d)\}$ all edges are compatible with the constraints; we select (r,d) because there are three alternatives and 0.1 selects the first one (the probabilities are uniform);

Add	Evaluation	Cumulated probability
(r,d)	$\delta f = 26$	$0.\bar{3}$
(b,e)	$\delta f = 18$	$0.\bar{6}$
(c,d)	$\delta f = 14$	1.0

now $z^{(1)} = \{(r,a),(r,b),(r,d)\}$.

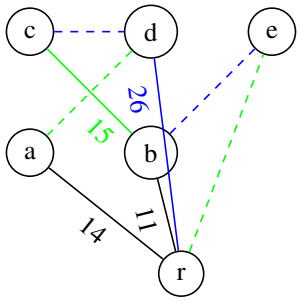


$z^{(1)} = z^{(0)} \cup \{(r,d)\} \Rightarrow f(z^{(1)}) = 51$

3. in $x'' \setminus z^{(1)} = \{(r,e),(a,d),(b,c)\}$ only the edges (r,e) and (b,c) are compatible with the constraints because (a,d) would close a loop, and 0.6 selects the second one;

Add	Evaluation	Cumulated probability
(r,e)	$\delta f = 24$	0.5
(a,d)	Unfeasible (cycle)	
(b,c)	$\delta f = 15$	1.0

now $z^{(2)} = \{(r,a),(r,b),(r,d),(b,c)\}$.



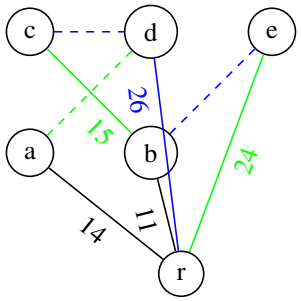
$z^{(2)} = z^{(1)} \cup \{(c,d)\} \Rightarrow f(z^{(1)}) = 66$

4. in $x' \setminus z^{(2)} = \{(b,e),(c,d)\}$ no edge is compatible with the constraints, because (b,e) would yield a tree with weight $> V = 2$ and (c,d) would close a loop;

Add	Evaluation
(b,e)	Unfeasible (capacity)
(c,d)	Unfeasible (cycle)

the solution does not change: $z^{(3)} = z^{(2)} = \{(r,a),(r,b),(r,d),(b,c)\}$.

5. in $x'' \setminus z^{(3)} = \{(r,e),(a,d)\}$ only edge (r,e) is compatible with the constraints because (a,d) would close a loop; now $z^{(4)} = \{(r,a),(r,b),(r,d),(b,c),(r,e)\}$ is a feasible solution.



$z^{(4)} = z^{(3)} \cup \{(r,e)\} = z^{(2)} \cup \{(r,e)\} \Rightarrow f(z^{(1)}) = 90$

10.7.7 Exercise 7

Consider the following instance of the *Parallel Machine Scheduling problem* (*PMSP*) with $|M| = 3$ machines:

Tasks	a	b	c	d	e	f	g
Durations	8	6	7	12	5	13	10

Apply the recombination step of *Scatter Search* to solutions[4] $x = \{\{a,b,c\},\{d,e\},\{f,g\}\}$ and $x' = \{\{a,b,g\},\{c,d\},\{e,f\}\}$ to generate a new solution, choosing the elements in a greedy way, alternatively, first from x then from x'.

Do the same applying a random recombination with free choice of the elements from x and x', using the following sequence of pseudorandom numbers: 0.2, 0.9, 0.4, 0.1...

Assume that the subset of the best solutions consists only of x and x' ($B = \{x,x'\}$), the subset of the diverse solutions D consists of $y = \{\{b,g\},\{a,d,f\},\{c,e\}\}$ and $y' = \{\{a,e,f\},\{b,c\},\{d,g\}\}$, that both are full and that the recombination phase generated solutions $z = \{\{a,f\},\{b,e,g\},\{c,d\}\}$ and $z' = \{\{b,c\},\{d,g\},\{a,e,f\}\}$. Update the reference set R with the new solutions.

Solution At first, the procedure computes the intersection of the two solutions:

$$z^{(0)} = x \cap x' = \{\{a,b\},\{d\},\{f\}\}$$

which implies the following total execution times for the three machines: $(14,12,13)$.
The remaining objects are c, e and g. The greedy alternating procedure considers their assignment in x and chooses the one that provides the best completion time:

- assigning c to the first machine implies $f(z^{(0)} \cup \{(c,1)\}) = \max(21,12,13) = 21$;
- assigning e to the second machine implies $f(z^{(0)} \cup \{(e,2)\}) = \max(14,17,13) = 17$;
- assigning g to the third machine implies $f(z^{(0)} \cup \{(g,3)\}) = \max(14,12,23) = 23$.

The procedure selects $(e,2)$, obtaining $z^{(1)} = \{\{a,b\},\{d,e\},\{f\}\}$ with $f(z^{(1)}) = 17$.
Now we consider the assignments of c and g in x':

- assigning c to the second machine implies $f(z^{(1)} \cup \{(c,2)\}) = \max(14,24,13) = 24$;
- assigning g to the first machine implies $f(z^{(1)} \cup \{(g,1)\}) = \max(24,17,23) = 24$.

Since they are equivalent, we choose lexicographically the first, obtaining $z^{(2)} = \{\{a,b\},\{c,d,e\},\{f\}\}$ with $f(z^{(2)}) = 24$.
Since there is a single remaining assignment, the last step considers the assignment of g in x and obtains $z^{(3)} = \{\{a,b\},\{c,d,e\},\{f,g\}\}$ with $f(z^{(3)}) = \max(24,17,23) = 24$.

If we apply the random free procedure, we still start from the intersection $z^{(0)} = x \cap x' = \{\{a,b\},\{d\},\{f\}\}$, with $f(z^{(0)}) = \max(14,12,13) = 14$. The remaining elements must be chosen in

$$(x \cup x') \setminus z^{(0)} = \{(c,1),(c,2),(e,2),(e,3),(g,1),(g,3)\}$$

that is sorted lexicographically by convention. Pseudorandom number 0.2 suggests to select the second term, $(c,2)$. Now we could proceed keeping all pairs and neglecting those that have become unfeasible, but they are so few that it is not complicated even by hand to remove them (and

[4]For the sake of briefness, we can represent the solutions of the *PMSP* also as a collection of $|M|$ subsets of tasks, instead of a subset of pairs (task,machine).

it is more correct). The new set of options is $\{(e,2),(e,3),(g,1),(g,3)\}$. Pseudorandom number 0.9 suggests to select the fourth term, $(g,3)$, which leaves only $\{(e,2),(e,3)\}$. Finally, pseudorandom number 0.4 suggests to select $(e,2)$. The final solution is $z^{(3)} = \{\{a,b\},\{c,d,e\},\{f,g\}\}$ with $f(z^{(3)}) = \max(14,24,23) = 24$.

The current subset B consists of two solutions costing $f(x) = \max(21,17,23) = 23$ and $f(x') = \max(24,19,18) = 24$. The current subset D consists of two solutions whose Hamming distances from the other ones in the reference set and whose costs are:

$$d_H(y,R\setminus y) = d_H(y,x) + d_H(y,x') + d_H(y,y') = 10+6+14 = 30 \quad f(y) = \max(13,33,12) = 33$$

and

$$d_H(y',R\setminus y') = d_H(y',x) + d_H(y',x') + d_H(y',y) = 10+10+14 = 34 \quad f(y') = \max(26,13,22) = 26$$

The "worst" element of D (with respect to the distance) is, therefore, y.
Solution z has a value equal to $f(z) = \max(21,21,19) = 21$, that is better than the worst in B, since $f(x') = 24$ (actually, it is also better than the best). Therefore, z replaces x', yielding $B = \{x,z\}$. In principle, this implies that the distances used in the diverse set should be recomputed:

$$d_H(y,R\setminus y) = d_H(y,x) + d_H(y,z) + d_H(y,y') = 8+12+14 = 34$$

and

$$d_H(y',R\setminus y') = d_H(y',x) + d_H(y',z) + d_H(y',y) = 10+6+14 = 30 \quad f(y') = \max(26,13,22) = 26$$

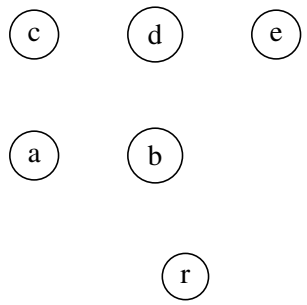
The "worst" element of D (with respect to the distance) is, therefore, y'.
Solution z' has a value equal to $f(z' = \max(13,22,26) = 26$, that is worse than the worst in B. We therefore compute its distance from $R\setminus y'$, that is

$$d_H(z',R\setminus y') = d_H(z',x) + d_H(z',z) + d_H(z',y) = 6+12+8 = 26$$

smaller than $d_H(y',R\setminus y')$. Therefore, D is unchanged.

10.7.8 Exercise 8

Consider the following instance of the Capacitated Minimum Spanning Tree (*CMST*) problem, that is a complete graph (the edges are not reported for the sake of clarity) with 5 vertices of unitary weight $w_v = 1$ for all $v \in \{a,b,c,d,e\}$, a root vertex, capacity $V = 2$.



and the following cost matrix

	r	a	b	c	d	e
r	-	14	11	25	26	24
a	14	-	15	13	16	27
b	11	15	-	15	12	18
c	25	13	15	-	14	20
d	26	16	12	14	-	10
e	24	27	18	20	10	-

Given solutions

- $x' = \{(r,a),(r,b),(r,d),(b,e),(c,d)\}$
- $x'' = \{(r,a),(r,b),(r,e),(a,d),(b,c)\}$

apply a Path Relinking procedure from x' to x'' using the one-swap neighbourhood $\mathcal{N}_{S_1}$ with respect to the edges of the graph.

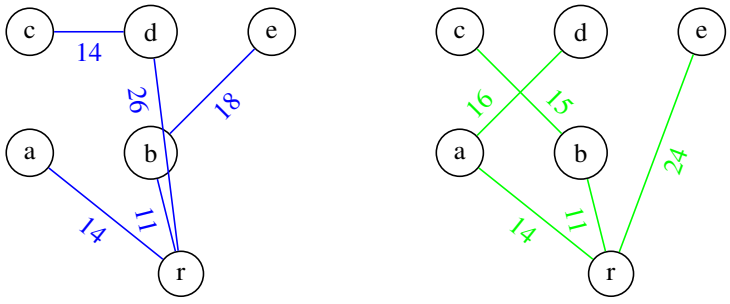
Solution The problem and the two starting solutions are the same as for the *Scatter Search* exercise. In order to reduce the Hamming distance from the current solution x to the final one, x'', every exchange must:

- delete an edge from $z \setminus x''$;
- add an edge from $x'' \setminus z$.

At the first step, $z^{(0)} = x'$ and the possible swaps require to:

- delete an edge from $z^{(0)} \setminus x'' = \{(r,d),(b,e),(c,d)\}$;
- add an edge from $x'' \setminus z^{(0)} = \{(r,e),(a,d),(b,c)\}$.

The two extreme solutions are represented in the following picture.

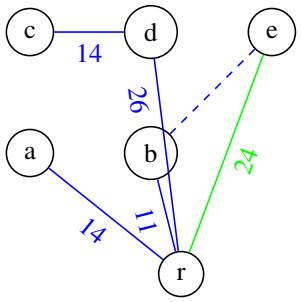


$f(x') = 83$ $f(x'') = 80$

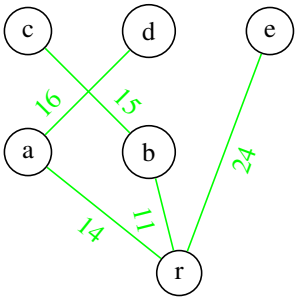
The possible swaps are evaluated in the following table.

Add	Delete	Evaluation
(r,e)	(r,d)	Unfeasible (cycle and disconnection)
	(b,e)	$\delta f = 24-8 = 16$
	(c,d)	Unfeasible (cycle and disconnection)
(a,d)	(r,d)	Unfeasible (capacity)
	(b,e)	Unfeasible (cycle and disconnection)
	(c,d)	Unfeasible (cycle and disconnection)
(b,c)	(r,d)	Unfeasible (capacity)
	(b,e)	Unfeasible (cycle and disconnection)
	(c,d)	Unfeasible (capacity)

Since only swapping (r,e) and (b,e) is feasible and reduces the Hamming distance, the next solution is $z^{(1)} = \{(r,a),(r,b),(r,d),(r,e),(c,d)\}$ and its cost is $f\left(z^{(1)}\right) = 89$



$z^{(1)} = x \cup \{(r,e)\} \setminus \{(b,e)\} \Rightarrow f(z^{(2)}) = 90$



$f(x'') = 80$

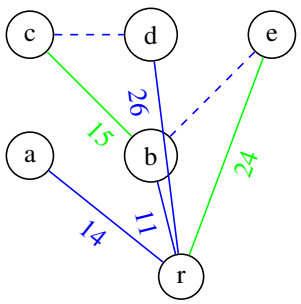
The possible swaps from $z^{(1)}$ towards x'' require to:

- delete an edge from $z^{(1)} \setminus x'' = \{(r,d),(c,d)\}$;
- add an edge from $x'' \setminus z^{(1)} = \{(a,d),(b,c)\}$.

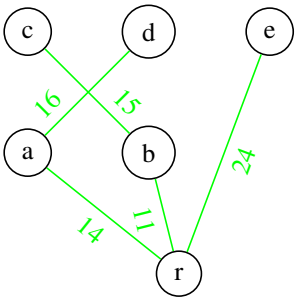
and are evaluated in the following table.

Add	Delete	Evaluation
(a,d)	(r,d)	Unfeasible (capacity)
	(c,d)	Unfeasible (cycle and disconnection)
(b,c)	(r,d)	Unfeasible (capacity)
	(c,d)	$\delta f = 15-14 = 1$

The only feasible exchange yields $z^{(2)} = \{(r,a),(r,b),(r,d),(r,e),(c,d)\}$, whose cost is $f\left(z^{(2)}\right) = 90$.



$z^{(2)} = z^{(1)} \cup \{(b,c)\} \setminus \{(c,d)\} \Rightarrow f(z^{(1)}) = 89$



$f(x'') = 80$

A final swap between (a,d) and (r,d) would allow to reach x'', but this is not required because that solution is already known.

The best solution found along the path is $z^{(1)}$, and it will be tested for insertion in the reference set. Probably, it is not a very promising starting point for further search, but this is mainly due to the very small size of the instance.

10.7.9 Exercise 9

Consider the following instance of the *Parallel Machine Scheduling problem* (*PMSP*):

Tasks	a	b	c	d	e	f	g
Durations	6	4	5	10	3	11	8

Apply the *Path Relinking* mechanism from solution $x = \{\{a,b,c\},\{d,e\},\{f,g\}\}$ to solution $x' = \{\{a,b,g\},\{c,d\},\{e,f\}\}$ to generate new solutions and evaluate them, using as a basic neighbourhood $N_{\mathcal{S}_1}$ (that is, the swap between two tasks assigned to different machines). What can be observed with respect to the objective function?

Solution The *PR* procedure considers only the exchanges on solution x that reduce the Hamming distance from solution x'. Since there are three tasks assigned to different machines, the distance is $d_H(x,x') = 6$. There are three exchanges that reduce it:

- swap (c,e) generates $\{\{a,b,e\},\{c,d\},\{f,g\}\}$, whose Hamming distance from x' is 4 and whose cost is $\max(13,15,19) = 19$;
- swap (c,g) generates $\{\{a,b,g\},\{d,e\},\{c,f\}\}$, whose Hamming distance from x' is 4 and whose cost is $\max(18,13,16) = 18$;
- swap (e,g) generates $\{\{a,b,c\},\{d,g\},\{e,f\}\}$, whose Hamming distance from x' is 4 and whose cost is $\max(15,18,14) = 18$.

The best ones are the second and third; we choose lexicographically (c,g), obtaining $z^{(0)} = \{\{a,b,g\},\{d,e\},\{c,f\}\}$ with $f\left(z^{(0)}\right) = 18$.

Now, there is a single swap that reduces the Hamming distance from x', that is (c,e) and it directly generates x', so it is not necessary to consider it.

10.7.10 Exercise 10

Consider the following instance of the *Bin Packing problem* (*BPP*):

Items	a	b	c	d	e	f	g
Volumes	8	6	3	9	5	12	10

Bin capacity
21

Consider the binary vector encoding of the solutions of the *BPP*, where each index is associated with an element of the ground set $O \times C$, that is the assignment of an object to a container. Write the encodings of the two solutions $x = \{\{a,b,c\},\{d,e\},\{f,g\}\}$ and $x' = \{\{a,b,g\},\{c,d\},\{e,f\}\}$, and compute the Hamming distance between them.

Solution Rather than vectors, binary matrices offer a clearer (and fully equivalent) representation:

$$\xi = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \qquad \xi' = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

The Hamming distance is $d_H(x,x') = 6$, as 3 objects are assigned to different containers.

10.7.11 Exercise 11

Consider a population of 4 individuals $X = \{x_1, x_2, x_3, x_4\}$, characterised by the following fitness values

$$\phi(x) = [\ 10\ 18\ 9\ 13\ ]$$

and assume that the pseudorandom number generator provides the following sequence: $r = (0.47, 0.33, 0.80, 0.71, 0.12, 0.93)$. Generate a new population of individuals with

a) the roulette wheel selection mechanism;

b) the rank selection mechanism;

c) the tournament selection mechanism on the following subsets of individuals: $(1,3)$, $(1,4)$, $(2,3)$ and $(2,4)$;

Solution **Part a)** Roulette wheel selection assigns to each individual a probability proportional to the value of its fitness. Since the sum of all fitnesses is $10+18+9+13 = 50$, the corresponding probabilities and their partial sums are reported in the following table.

Individual i	x_1	x_2	x_3	x_4
$\phi(\xi(x))$	10	18	9	13
$\pi_i = \dfrac{\phi_i}{\sum_j \phi_j}$	0.20	0.36	0.18	0.26
$\sum_{j=1}^{i} \pi_j$	0.20	0.56	0.74	1.00

The sequence of extractions is the following (the extracted elements are not removed, so that they can be extracted any number of times):

1. $r = 0.47$ falls in the second interval, so that we select x_2;
2. $r = 0.33$ falls in the second interval, so that we select x_2;
3. $r = 0.80$ falls in the fourth interval, so that we select x_4;
4. $r = 0.71$ falls in the third interval, so that we select x_3.

The final population is $X = \{x_2, x_2, x_3, x_4\}$.

Part b) Rank selection assigns to each individual a probability proportional to its index in a fitness nondecreasing order. Since $\phi(\xi(x_3)) < \phi(\xi(x_1)) < \phi(\xi(x_2)) < \phi(\xi(x_4))$, solution x_3 has index 1, solution x_1 has index 2 and so on, producing the following table of indices, probabilities and partial sums of probabilities.

Individual i	x_1	x_2	x_3	x_4
Index k	2	4	1	3
$\pi_i = \frac{2k}{n(n+1)}$	0.20	0.40	0.10	0.30
$\sum_{j=1}^{i} \pi_j$	0.20	0.60	0.70	1.00

The sequence of extractions is the following (the extracted elements are not removed, so that they can be extracted any number of times):

1. $r = 0.47$ falls in the second interval, so that we select x_2;
2. $r = 0.33$ falls in the second interval, so that we select x_2;
3. $r = 0.80$ falls in the fourth interval, so that we select x_4;
4. $r = 0.71$ falls in the fourth interval, so that we select x_4.

The final population is $X = \{x_2, x_2, x_4, x_4\}$.

Part c) Tournament selection extracts $n_p = 4$ subsets of $\alpha = 2$ individuals, given in the text of the exercise, and selects the individual with the largest fitness in each subset.

The sequence of selections is:

1. from (x_1, x_3), select x_1;
2. from (x_1, x_4), select x_4;
3. from (x_2, x_3), select x_2;
4. from (x_2, x_4), select x_2.

The final population is $X = \{x_1, x_2, x_2, x_4\}$.

10.7.12 Exercise 12

Consider the following instance of the *Knapsack Problem* with capacity $V = 12$:

Objects	a	b	c	d	e	f	g
Prize ϕ	11	2	4	9	8	4	9
Volume v	5	1	2	4	3	1	6

Adopt the classical incidence vector encoding for $x = (a, b, d)$ and $x' = (b, e, g)$, and apply a *uniform crossover* operator to them, generating the binary mask with the following pseudorandom number sequence: 0.1, 0.3, 0.7, 0.9, 0.6, 0.8, 0.2, ... where lower values correspond to 0 (no exchange) and higher values to 1 (exchange).

Solution The two encoding are

$$\xi(x) = [\,1\,1\,0\,1\,0\,0\,0\,]$$

and

$$\xi(x') = [\,0\,1\,0\,0\,1\,0\,1\,]$$

The binary mask corresponding to the pseudorandom number sequence is

$$m = [\,0\,0\,1\,1\,1\,1\,0\,]$$

meaning that the first, second and last bit should remain unchanged, whereas the other ones should be exchanged between the two vectors.

Applying the mask to the original two encodings produces the following modified ones:

$$\xi = [\,1\,1\,0\,0\,1\,0\,0\,]$$

and

$$\xi' = [\,0\,1\,0\,1\,0\,0\,1\,]$$

where the fourth and fifth element are explicitly exchanged, whereas the third and sixth apparently do not change because they are the same, and the first, second and last are unaffected by the mask. The resulting solutions are $x(\xi) = \{a,b,e\}$ and $x(\xi) = \{a,b,e\}$ with $f(x(\xi)) = 21$ and $x(\xi^{prime}) = \{b,d,g\}$ with $f(x(\xi')) = 20$. They are both feasible.

Part VI

Laboratory sessions

APPENDIX A

Generalities

The laboratory sessions of the course on heuristic algorithms aim to illustrate the practical aspects of the design, implementation and evaluation of heuristic algorithms for combinatorial optimisation problems.

These lessons assume a basic background on C language programming and on fundamental algorithms and data structures. The lessons, therefore, will not go into technical details on this topics, but will just briefly recall the existence of instructions, algorithms and data structures that allow to apply the fundamental operations of the heuristic algorithms considered.

For the sake of simplicity, the lessons refer to a single combinatorial optimisation problem, that is the *maximum diversity problem* (in short, *MDP*). This problem has been chosen among the other treated in the course because its definition is very simple and its solutions can be represented and manipulated in a rather simple way. On the other hand, the problem is strongly NP-complete and does not admit any constant factor approximation guarantee. Therefore, it is rather difficult to solve it to optimality. Finally, it presents several interesting aspects concerning the effectiveness and the efficiency of the most common algorithmic procedures (insertions, exchanges and recombinations) and not excessively related to specific features of the problem.

Section A.1 defines the problem and describes the data structure and the basic procedures (implemented as C libraries) that will be used to manipulate the instances and the solutions. Section B describes the implementation and evaluation of some constructive and destructive heuristics.

A.1 The maximum diversity problem

A.1.1 Definition

The *maximum diversity problem* (*MDP*) is defined by:

- a set P of *points* in an abstract space (we define $n = |P|$);
- a *distance* function $d : P \times P \to \mathbb{N}$, that associates each pair of points to a nonnegative integer distance;
- a positive integer number $k \in \mathbb{N}$ with $0 < k < |P|$.

The problem consists in determining a subset $x \subset P$ such that

- the sum of the pairwise distances between the points of x is maximized, that is

$$\max_{x \subseteq P} f = \sum_{i \in x} \sum_{j \in x} d_{ij}$$

- the cardinality of x is equal to k ($|x| = k$).

Some minor remarks allow to restrict the possible data sets without affecting the generality of the problem. To start with, in practical applications the distances could be real numbers, but computers will always represent them with a finite precision, so that they can be considered as rational numbers. Moreover, rational values can always be transformed into integer ones by changing the unit of measure. Then, it can always be assumed that

$$d_{ij} = d_{ji} \text{ for all } i, j \in P$$

In fact, if for all pairs of points (i, j) we replace d_{ij} and d_{ji} with their arithmetic mean $\left(d_{ij} + d_{ji}\right)/2$, the value of any solution x is unchanged, since the sum $\sum_{i \in x} \sum_{j \in x} d_{ij}$ contains either both distances or none. We can say that

$$d_{ii} = 0 \text{ for all } i \in P$$

In fact, if every solution contains exactly k points, the sum which provides the value of the objective contains exactly k terms d_{ij} for each $i \in x$, one of which is the term d_{ii}; setting d_{ii} to zero and summing $d_{ii}/(k-1)$ to each term d_{ij} the overall value of the objective does not change; the same operation can be done on the terms d_{ij} for $i \notin x$, given that they do not occur in the objective. Finally, we can say that

$$d_{ij} \geq 0 \text{ for all } i, j \in P$$

In fact, if every solution contains k points, the value of the objective is always a sum of k^2 terms; summing a constant value $\bar{d}$ to every term so as to make them nonnegative, the objective function increases by $k^2\bar{d}$ for every solution, so that the optimal solution does not change; therefore, the distances can be considered as nonnegative integer numbers.

By contrast, in general the triangle inequality is not satisfied:

$$d_{ij} + d_{jk} \geq d_{ik} \text{ for all } i, j, k \in P$$

There are *MDP* instances which enjoy it and instances which do not.

A.1.2 Benchmark instances

The literature provides several classes of benchmark instances for the *MDP*. Since most of them have a fixed size, but we aim to discuss also the dependence of the results on size, we will generate another benchmark set, inspired by the available ones, but with a larger range of different sizes. The main features of these instances are:

- set P ranges from $n = 100$ to $n = 1\,000$ points by steps of 100;
- the integer number k is equal to $0.1n$, $0.2n$, $0.3n$, $0.4n$;
- the values d_{ij} of the distance function (with $i < j$) are random integer numbers uniformly distributed in $\{1, \ldots, 100\}$;

- the values for $i \geq j$ derive from the assumptions made in the previous section.

Overall, they are 40 instances (ten different sizes times four different values of k).

They are neither particularly significant nor realistic instances, but they are hard enough not to be trivially solved to optimality by any method, but easy enough to assume that the best known solution is probably close to the optimum (even if this has not been proved) and small enough to require a reasonable processing time from any polynomial algorithm.

The data are provided in text files. The name of each file reports the features of the instance: file `n[%n]k[%k].dat` corresponds to an instance with a set P of `%n` points, and a required cardinality equal `%k` for the solution.

The file adopts the AMPL format, a standard format to represent Integer Programming problems, used by general-purpose modelling languages and solvers for that family of problems. The format is rather self-evident (and anyway does not concern us, given that library `data.h` provides a function to load the data into the memory).

```
param n := 100 ;
param k := 10 ;
param D :=
[1,1] 0 [1,2] 42 [1,3] 10 [1,4] 75 [1,5] 53 ...
[2,1] 42 [2,2] 0 [2,3] 13 [2,4] 33 [2,5] 84 ...
...
```

A.2 Instance representation

The C library `data.h` provides the data structure

```
typedef struct data_s
 {
  int n;   /* cardinality of the set of points P */
  int k;   /* cardinality of the feasible solutions x \in X */
  int **d; /* distance matrix between the points */
 } data_t;
```

that we will use to represent each given instance I of the *MDP*, since it consists of three simple components: a set P, a metric d and an integer number k.

We will represent set P with the natural numbers from 1 to $n = |P|$, since this set has no other specification besides the metric. We do not adopt the classical C language convention that represents integer number sets starting from zero, in order to keep consistent with the data available in the literature, and to possibly use index 0 for special operations.

We will represent metric d with a square integer matrix. We could save space exploiting the symmetry of the metric, for example by representing only the values d_{ij} with $i < j$, but this would require to test at each access to d_{ij} whether $i < j$ or not, in order to exchange the two indices if the result of the test is negative. Since we expect to access the data a huge number of times, we choose to represent all of them, preferring time efficiency over space efficiency.

The distance matrix is dynamic, and will be allocated when loading the data from a text file and deallocated at the end of the algorithm. The C library `alloc.h` is already available, and provides functions to allocate integer vectors (given the actual number of elements) and matrices (given the actual number of rows and columns).

```
/* Allocate a vector of n int */
int *int_alloc (int n);

/* Allocate a matrix of (n1,n2) int */
int **int2_alloc (int n1, int n2);
```

To manage the instances, library `data.h` provides functions to load the data from a text file (in the standard AMPL format), to deallocate the struct `data_t` described above and to print the instance on the screen (once again in AMPL format).

```
/* Load from the AMPL file data_file the instance *pI */
void load_data (char *data_file, data_t *pI);

/* Deallocate the instance *pI */
void destroy_data (data_t *pI);

/* Print the instance *pI in AMPL format */
void print_data (data_t *pI);
```

As observed above, a point is just an abstract object with no associated information, represented by an integer number ranging from 1 to P. With some excess of zeal, we will however introduce a level of abstraction and distinguish:

- on the one hand, the numerical index of a point, that is an `int`;
- on the other hand the abstract point, with all its associated information (if any), that is a `point`.

For the sake of rigour, we shall use `int` variables to scan the indices (for example, in the distance matrix) and `point` variables to scan the actual points (for example, the elements of a solution x and of its complement $P \setminus x$, as will be described in the following section). As long as the two concepts coincide, we will identify them with the type definition:

```
typedef int point;
```

allowing future developments, in which a point could be associated to auxiliary information (e. g., coordinates, text strings, etc...) besides its numerical index. This approach allow more easily to modify the data structures without modifying the algorithms already implemented.

The disadvantage is that, in order to switch between points and indices, we must use the functions:

```
/* Get the index of point i in instance *pI */
int get_index (point i, data_t *pI);

/* Get the point of index id in instance *pI */
point get_point (int id, data_t *pI);
```

If the data structure will need to be modified, these functions will take care of the corresponding type conversions, but currently they just return in output the same value received in input. This is an inefficiency, that can be controlled by adopting suitable technological solutions, such as inline functions or macros to implement the conversion.

A.3 Solution representation

The solutions of the *MDP* are subsets of the ground set, as for any other Combinatorial Optimisation problem. There are two main ways to represent a subset:

1. with an *incidence vector*, which associates each $i \in P$ to a boolean value

$$x_i = \begin{cases} \texttt{true} \text{ when } i \in x \\ \texttt{false} \text{ when } i \in P \setminus x \end{cases}$$

2. with a *list of elements*, which allows to scan only the elements of the solution $i \in x$

The choice between the two representations depends on the type of operations that the algorithm needs to perform: the inclusion test of a point in the solution is efficient in the first representation ($O(1)$), inefficient in the second one ($O(n)$); the opposite holds for the operation of scanning only the elements of the solution, or the elements out of it. The algorithms we want to implement mainly use operations on lists, both the internal and external points. For example, each step of a constructive algorithm scans the set $\Delta_A^+(x)$, that in the *MDP* is the complement of the solution, $P \setminus x$, whereas the computation of the value of a solution, $f(x)$, requires to scan its elements. However, we will adopt both representations to keep as flexible as possible. We will also represent the complementary set of the solution, $P \setminus x$ as another list, because many algorithms require to scan its points. If one of the two representations is not actually used, we can decide to remove it *a posteriori*. The C library `solution.h` provides the struct

```
typedef struct solution_s
{
  int f;          /* solution value */

  bool *in_x;     /* incidence vector: in_x[i] = true if i \in x,
                                       in_x[i] = false if i \notin x */

  /* Lists of points in solution x and in the complement P \setminus x */
  int head_x;     /* sentinel of the list of points in x */
  int head_notx; /* sentinel of the list of points in P \setminus x */
  int *next;      /* next     element for each point i in either list */
  int *prev;      /* previous element for each point i in either list */

  int card_x;     /* cardinality of the solution */

} solution_t;
```

The value f of the solution is saved in member `f` and kept up to date, so that it can be accessed in constant time $O(1)$, instead of recomputed every time. Its recomputation, in fact, would require $O(n^2)$ time if done scanning the incidence vector, $O(k^2)$ time if done scanning list x.

Note: since the distance matrix is symmetric and integer, $f(x)$ is certainly an even number, as it is the sum of pairs of equal terms. It is a common convention to report in `f` half of the overall sum. We shall discuss this point again later.

The boolean vector `in_x` represents the incidence vector. The `boolean` type, with its two values `false` and `true` is declared in the library `defs.h`.

```
typedef enum _bool bool;
enum _bool {false = 0, true = 1};
```

and dynamic vectors of booleans can be allocated thanks to the already cited library `alloc.h`, that provides function

```
/* Allocate a vector of n bool */
bool *bool_alloc (int n);
```

The two lists that represent the solution x and its complement $P \setminus x$ are *doubly-linked circular lists with sentinel*, so that every fundamental operation (insertion, extraction, etc...) can be performed in constant time, at the cost of a larger memory occupation. Briefly, such lists can be scanned in both directions and are never physically empty, because they always contain the fictitious element known as *sentinel* (by convention, an empty list is a list containing only the sentinel). This removes the need for different ways to operate on different parts of the list (the beginning, inner positions or the end).

We will adopt the implementation of the two lists with vectors and indices, instead of the implementation with pointers and dynamically allocated structures, because all possible points are defined once for all at the beginning of the algorithm, and only their positions change dynamically during the execution. Moreover, since the two lists do not intersect (they are complementary), we will exploit the same vectors and indices `next` and `prev` for the two lists; only the heads `head_x` and `head_notx` will be different. The sentinel of list x has index `0`, the sentinel of list $P \setminus x$ has index `card_N+1`. Only the intermediate values correspond to regular indices. The following example shows how the solution $x = \{1,3,7\}$ of an instance with point set $P = \{1,2,3,4,5,6,7\}$ will be represented:

```
f        46

           1  2  3  4  5  6  7
in_x     [ 1  0  1  0  0  0  1 ]

head_x    0
head_notx 8

         0  1  2  3  4  5  6  7  8
next   [ 1  3  4  7  5  6  8  0  2 ]
prev   [ 7  0  8  1  2  4  5  3  6 ]

card_x  3
```

The cardinality of the solution $|x|$ (`card_x`) should be fixed to k, but reporting it explicitly allows the structure to represent also general subsets of P, and in particular *partial solutions* of cardinality $< k$. In fact, the set of all partial solutions is the search space $\mathcal{F}_A$ for all the constructive algorithms considered in the following.

In order to avoid programming technicalities, we will hide many implementation details using library functions to access the data. This also allows, if necessary, to modify the low-level implementation without affecting already implemented algorithms. It can, however, imply some time inefficiencies, because it requires function calls instead of the simple direct access to data structures. Such inefficiencies can be easily overcome by using macros (or `inline` definitions in C++), so we accept them, but we do not describe how to do that to avoid technicalities.

The C library `solution.h` provides some functions to manage solutions:

```
/* Create an empty solution for a problem of size n */
void create_solution (int n, solution_t *px);

/* Deallocate the solution *px */
void destroy_solution (solution_t *px);

/* Turn a solution into the empty set for a problem of size n */
void clean_solution (int n, solution_t *px);

/* Copy solution *px_orig into solution *px_dest */
void copy_solution (solution_t *px_orig, solution_t *px_dest);

/* Print by increasing indices solution *px for a problem of size n */
void print_sorted_solution (solution_t* px, int n);
```

The creation of an empty solution corresponds to the typical initial step of a constructive heuristic, that starts from the empty subset. The deallocation is performed at the end of the algorithms. Cleaning a solution allows to restart an algorithm without deallocating and reallocating the memory. The copy function is useful to update the best known solution when the current one improves it. The print function is useful to analyse the results: it reports on the screen on a single row the name of the data file, the value of the objective and the list of points in solution x.

Another block of functions allow to access the solution x and its complement $P \setminus x$, avoiding any explicit reference to their concrete implementation. List x can be scanned with the following library functions:

```
/* Return the first and the last point of solution *px */
point first_point_in (solution_t *px);
point last_point_in (solution_t *px);

/* Return the point following and preceding i in solution *px */
point next_point (point i, solution_t *px);
point prev_point (point i, solution_t *px);

/* Indicate whether i is a regular point or a sentinel */
bool end_point_list (point i, solution_t *px);
```

In order to scan the complementary list $P \setminus x$, the functions to access the first and last point change, because the sentinel has a different index:

```
/* Return the first and the last point of the complement of solution *px */
point first_point_out (solution_t *px);
point last_point_out (solution_t *px);
```

but the functions that return the next and the previous point and the function that indicates whether the point is regular or the sentinel remain the same for both lists, because they share the same vectors and follow the same rules.

For example, given two points:

```
point i, j;
```

in order to scan solution x from the first to the last point, one can perform the loop

```
for (i = first_point_in(&x); !end_point_list(i,&x); i = next_point(i,&x))
```

and to scan the complement $P \setminus x$ from the first to the last point, one can perform the loop

```
for (j = last_point_out(&x); !end_point_list(j,&x); j = prev_point(j,&x))
```

The main manipulations of a solution in a constructive algorithm is the addition of a point, that requires to move it from the complementary list to the solution list, while at the same time updating the incidence vector and the value of the objective. It is adviseable to define a specific function for this basic operation, to make it as efficient as possible and to guarantee the consistency of the data structures. Functions operating on a single list would not make much sense for the overall problem and could easily introduce inconsistencies in the representation of the solution.

```
/* Add point i to solution *px */
void add_point (point i, solution_t *px, data_t *pI);

/* Delete point i from solution *px */
void delete_point (point i, solution_t *px, data_t *pI);
```

These functions must keep all components of the data structure consistent and up to date [1]. Function `add_point`:

1. adds to the objective function the sum of all distances of the newly added point from the previous ones (but not the reverse distances because `f` reports only half of the objective); this operation requires the distance matrix;
2. increases by one the cardinality `card_x`;
3. finds the index `id` of point `i` and sets the corresponding value of `in_x` to `true`;
4. extracts point `i` from list $P \setminus x$;
5. adds point `i` to list x.

Considering the previous example, adding point 4 to the solution yields the following data structure:

```
f        90

           1  2  3  4  5  6  7
in_x     [ 1  0  1  1  0  0  1 ]

head_x    0
head_notx 8
```

[1] It would be worth discussing whether these functions should require point `i` or index `id`. I guess the former is more likely in general, but the current implementation makes it indifferent.

```
        0  1  2  3  4  5  6  7  8
next  [ 1  3  5  7  0  6  8  4  2 ]
prev  [ 4  0  8  1  7  2  5  3  6 ]

card_x  4
```

The library also provides a function to remove a point from the solution, moving it from the solution list to the complementary list, and correspondingly updating the incidence vector and the value of the objective.

```
/* Delete point i from solution *px */
void delete_point (point i, solution_t *px, data_t *pI);
```

This function:

1. subtracts from the objective function the sum of all distances of the newly removed point from the remaining ones; this operation requires the distance matrix;

2. decreases by one the cardinality `card_x`;

3. finds the index `id` of point `i` and sets the corresponding value of `in_x` to `false`;

4. extracts point `i` from list x;

5. adds point `i` to list $P \setminus x$.

Removing point 1 from the solution previously augmented yields the following data structure:

```
f        72

           1  2  3  4  5  6  7
in_x     [ 0  0  1  1  0  0  1 ]

head_x    0
head_notx 8

        0  1  2  3  4  5  6  7  8
next  [ 3  8  5  7  0  6  1  4  2 ]
prev  [ 4  6  8  0  7  2  5  3  1 ]

card_x  3
```

All these operations require constant time, except for the update of the objective function, which requires $O(|x|)$ time for the addition and $O(n-|x|)$ time for the removal. This will be useful to implement destructive algorithms.

A.3.1 Consistency check

Notice that the use of a double representation for the solution implies a computational overhead and an additional effort to keep the two representations up to date. This choice should be justified by a better efficiency gained somewhere else. Moreover, this choice allows the risk to lose the consistency between different elements of the two representations. Such a risk is limited by the use of clearly defined functions to manipulate the solutions, but it is anyway impossible to remove completely. In our case, the data structure `solution_t` includes five potentially inconsistent components: objective value, cardinality, incidence vector, solution list and complementary list. The manipulation functions should guarantee the consistency between the five components. Even assuming that they are consistent at the beginning, every subsequent modification (for example, the introduction of new fields in the solution to allow other operations or to perform the same operations more quickly) could introduce inconsistencies, and therefore errors.

When implementing a heuristic algorithm, it is therefore a very good practice to write and maintain a function to check the internal consistency of the data structures. This function usually assumes one of the components as valid *a priori*, and recomputes the other ones, checking whether their current values are correct or not. The choice of the valid component is arbitrary, provided that it is sufficient to derive all of the other ones. In general, one uses the simplest component, that it that which is less likely to be incorrect. In our case, the available check function starts from the incidence vector, and derives from it the objective value, the cardinality and the two lists.

```
/* Check the internal consistency of solution *px based on instance *pI,
   starting from the incidence vector */
bool check_solution (solution_t *px, data_t *pI);
```

If the function finds an inconsistency, it return the value `false`, and the user can decide whether to terminate the execution to correct the code. Of course, the check function is used only during the implementation of the algorithm, and does not appear in its final version.

A.4 The main function

The `main` function in all the algorithms presented in the following sections manages the parsing of the command line (that is, the interpretation of the parameters of each algorithm), the loading of the data, the allocation and deallocation of the data and the solution, the execution of the algorithm, the determination of the computational time and the print of the result on the screen. Its general structure can be described as follows.

```
parse_command_line(argc,argv,data_file,&param);

load_data(data_file,&I);

create_solution(I.n,&x);

start = clock();
...
end = clock();
```

```
CPUtime = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6lf ",CPUtime);
print_sorted_solution(&x);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

where `data_file` stands for the name of the text file reporting the instance and `param` is a specific structure for each family of algorithms tested, that collects the parameters that identify a single algorithm. `I` is the instance of the problem and `x` the solution obtained. The starting time, ending time and overall duration of the computation are `start`, `end` and `CPUtime`. Finally, the dots (`...`) represent the actual call of the algorithm considered. The print of the results occurs on a single line of the screen, so that several calls to the algorithm can be collected in a *script*. This allows to run the algorithm on several benchmark instances, or with several different parameters, appending the results in a single file, one row for each run.

APPENDIX B

Laboratory on constructive heuristics

B.1 General scheme

The constructive heuristics usually apply the following simple general scheme:

Algorithm Greedy(I)
$x := \emptyset$; $x^* := \emptyset$;
If $x \in X$ *then* $f^* := f(x)$ *else* $f^* := +\infty$;
While $\Delta_A^+(x) \neq \emptyset$ *do*
    $i := \arg \min_{i \in \Delta_A^+(x)} \varphi_A(i,x)$;
    $x := x \cup \{i\}$;
    If $x \in X$ and $f(x) < f^*$ *then* $x^* := x$; $f^* := f(x)$;
Return (x^*, f^*);

Let us adapt this scheme to the specific case of the *MDP*. First, the only feasible subset visited by the algorithm is the last one. This allows to remove any reference to x^* and f^* and simply return (x, f) at the end of the algorithm. Second, the extremely simple structure of the feasible solutions (the only constraint is the fixed cardinality) suggests to define the search space as the set of the partial solutions, that is of the subsets with at most k points.

$$\mathcal{F}_A = \{x \subseteq P : |x| \leq k\}$$

This implies that the set $\Delta_A^+(x)$ of all possible extensions for a given partial solution x coincides with the complement of the latter (except in the last step, when it is empty):

$$\Delta_A^+(x) = \begin{cases} P \setminus x & \text{for } |x| < k \\ \emptyset & \text{for } |x| = k \end{cases}$$

Moreover, as the *MDP* is a maximisation problem, it is more natural to consider also the selection criterium as a function to maximise.

This transforms the general scheme as follows:

Algorithm GreedyMDP(I)
$x := \emptyset$;
While $|x| < k$ *do*
    $i := \arg\max_{i \in P \setminus x} \varphi_A(i,x)$;
    $x := x \cup \{i\}$;
Return (x, f);

This scheme can be easily implemented with the available functions, plus one that maximises the selection criterium:

```
void greedy (data_t *pI, solution_t *px)
{
  point i;

  while (get_card(px) < pI->k)
  {
    i = best_point_to_add(px,pI);
    add_point(i,px,pI);
  }
}
```

The instruction $x := \emptyset$ should correspond to `create_solution(pI->n,px)`, but we prefer to move it out of the algorithm, in the `main` function, under the form `create_solution(I.n,&x)` and to pass the empty solution thus obtained as an argument to function `greedy`. The advantage of this structure is that function `greedy` now can be used not only to generate a solution from scratch, but also to complete a possible partial solution obtained in any other way[1].

Function `best_additional_point(px,pI)` must be implemented to determine the best point `i` to add to solution `*px` based on the features of instance `*pI` according to the selection criterium $\varphi_A(i,x)$, which we have not yet defined. Different definitions will give rise to different constructive algorithms.

B.1.1 The basic constructive heuristic

Since the objective function can be easily extended to any subset of points, the simplest definition for the selection criterium is the value of the objective, that is

$$\varphi_A(i,x) = f(x \cup \{i\}) = \sum_{j \in x \cup \{i\}} \sum_{k \in x \cup \{i\}} d_{jk}$$

Computing it from scratch requires $O\left(|x|^2\right)$ time, but is not actually necessary, because it is enough to update it step by step choosing the point that maximises it. To achieve this result, one can consider the variation $\delta f(x,i) = f(x \cup \{i\}) - f(x)$

$$\delta f(x,i) = \sum_{j \in x} d_{ji} + \sum_{j \in x} d_{ij} + d_{ii} = 2 \sum_{j \in x} d_{ji}$$

[1] This would be useful if a preliminary reduction procedure or some manipulation of a model of the problem could prove (or suggests heuristically) the opportunity to include a promising subset of points.

which can be computed in $O(|x|)$ time. Also remind that we are updating and optimizing $f(x)/2$, so that the factor 2 can be removed from the expression of $\delta f(x,i)$.
The previous remark allows to implement the operation

$$i := \arg\max_{i \in P \setminus x} f(x \cup \{i\});$$

with the simple call

```
i = best_additional_point(px,pI);
```

of the following function

```
// Find the best point to add to solution *px based on the instance *pI
point best_additional_point (solution_t *px, data_t *pI)
{
  point i, i_max;
  int d, d_max;

  d_max = -1;
  i_max = NO_POINT;
  for (i = first_point_out(px); !end_point_list(i,px); i = next_point(i,px))
  {
    d = dist_from_x(i,px,pI);
    if (d > d_max)
    {
      i_max = i;
      d_max = d;
    }
  }

  return i_max;
}
```

which computes for each point `i` of list $P \setminus x$ the variation of the objective function, $\delta f(i,x)/2 = \sum_{j \in x} d_{ij}$, obtained adding `i` to solution x, that is the total distance of `i` from the points of x. This value is computed by function `dist_from_x(i,px,pI)`. The function returns the point `i_max` that yields the maximum increase. The result is the basic constructive algorithm.

B.2 Empirical evaluation

The benchmark considered is rather small and too specific to allow a truly meaningful analysis. However, it is sufficient to illustrate the process and to make some interesting remarks. Let us run the algorithm on the whole benchmark set. The *script* `greedy_solve.bat` applies the algorithm redirecting its output from the screen to the text file `report.txt`.

```
echo "File T_A f_A x_A" > report.txt
./main_greedy data/n0100k010.txt >> report.txt
./main_greedy data/n0100k020.txt >> report.txt
./main_greedy data/n0100k030.txt >> report.txt
...
```

The first line creates a header with four elements, that are potential labels in a table: the name of the instance file, the computational time T_A required by the algorithm, the value f_A of the objective function and the list of points in the solution x_A found. This header is redirected by directive > on the text file `report.txt`. Each following line applies the algorithm and redirects the output in *append* (with directive >>) on the same text file, so as to obtain a very regular summary, with the results of a single instance in each row.

```
File T_A f_A x_A
data/n0100k010.txt   0.000150    3308   1   33   70   31   72   ...
data/n0100k020.txt   0.000448   12120   1   72   61   12   66   ...
data/n0100k030.txt   0.000780   26115   1   96   46    4   57   ...
...
```

We are particularly interested in the columns reporting the computational time T_A and the result ($f_A(x)$).

B.2.1 Computational time analysis

Figure B.1 reports the *RTD diagram* for the whole benchmark. It is a good example of a "scientific-looking", but insignificant diagram, because the benchmark includes instances of different size, the computational time strongly depends on the size, and the diagram actually describes the specific benchmark more than a property of the algorithm, or the problem in general. The parametric *RTD* diagrams for fixed size (see Figure B.2) account for this aspect, but each one refers to only four instances, so they are also nearly meaningless. The fact that they are more and more spaced as the size increases suggests a more than linear dependence of the time on the size.

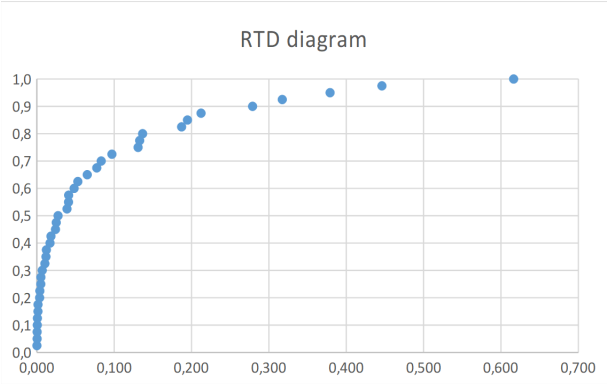


Figure B.1: *RTD* diagram for the greedy algorithm on the benchmark

The correct tool to describe the dependence of the computational time T_A on the instance size P is the *scaling diagram* (see Figure B.3). The first remark is that T_A is rather low, even for large instances ($n = 1\,000$), and often nearly "zero" for $n = 100$. For a fixed value of P, they appear distributed on a rather large "fan". An interesting question is whether this distribution is due to an important secondary feature of the instances or to wide random variations of the computational time over different instances. A useful hint is provided by the distribution of the points on the scaling diagram, that are rather clearly clustered in four profiles, corresponding to the different values of parameter k. The theoretical analysis confirms that k plays a significant

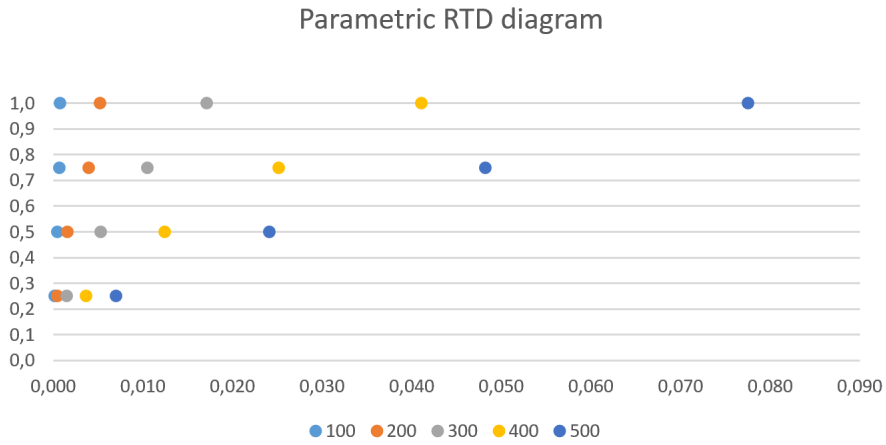


Figure B.2: Parametric *RTD* diagrams for the greedy algorithm on the benchmark

role. In fact, the general scheme implies k iterations, each of which searches for the best additional point scanning the $n-|x|$ external points and computing in $O(|x|)$ time the distance of each point from the current solution. The other operations are clearly faster, even if adding the new point to the solution takes $O(|x|)$ time to update the objective function value. Overall, the complexity is

$$T_A(n,k) = \sum_{i=1}^{k} O(n-i)\,O(i) = O\left(nk^2\right)$$

Since in the benchmark $k = \alpha n$, with $\alpha \in \{0.1, 0.2, 0.3, 0.4\}$, the theoretical estimate amounts to $T_A \in O\left(n^3\right)$.

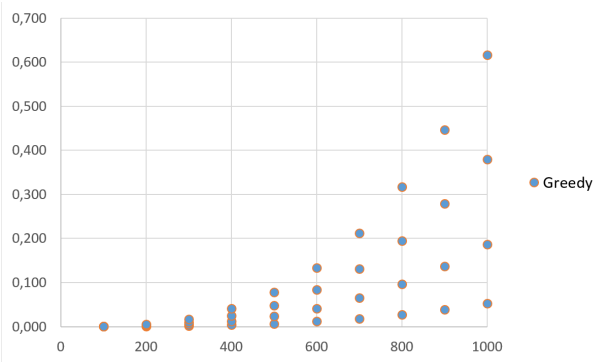


Figure B.3: Scaling diagram for the greedy algorithm on the benchmark

Let us verify whether it is correct by drawing the scaling diagram in a logarithmic scale:

$$T_A = \beta n^{\alpha} \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

Indeed, the graph seems to be remarkably linear. The linear interpolation suggests that $\alpha \approx 2.771$ and $\beta \approx 10^{-9}$. The value of β depends on constant multiplying factors, among which the technical parameters of the specific computer employed. The value of α suggests that the algorithm is actually less than cubic. This could be due to actual overestimates in the theoretical analysis (not likely in this case, given its rather simple structure) or to the fact that the instances

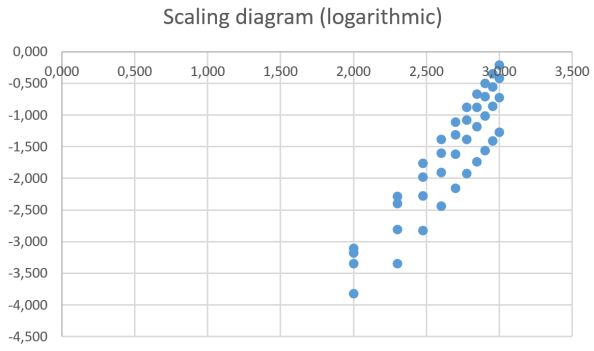


Figure B.4: Scaling diagram in logarithmic scales for the greedy algorithm on the benchmark

considered are not large enough to exhibit a full dominance of the main complexity factor. In the present case, there is a number of quadratic terms that indeed could still measurably affect the computational time (the update of the objective function, and possibly of the best additional point).

Solution quality analysis

In order to evaluate the quality of the results, it is adviseable to compute the gap (relative difference) with respect to the optimum, so that the values obtained from different instances could be compared in a more reasonable way. However, the optimum is not known, due to the hardness of the problem, and the gap should be replaced by an estimate. Two estimates are possible[2]:

$$\frac{LB - f_A(x)}{LB} \leq \delta_A(x) = \frac{f^* - f_A(x)}{f^*} \leq \frac{UB - f_A(x)}{UB}$$

In the case of the *MDP*, the best known upper bounds are of rather scarse quality, whereas the lower bounds seem to be closer to the optimum. The first estimate is therefore probably tighter, even if unfortunately they do not provide a quality guarantee (the real gap is larger).

Table B.1 reports the results of the basic greedy algorithm. None of the instances is solved to the optimum. Is this a good or a bad result? Of course, the answer would depend on a comparison with alternative algorithms, but in general it does not seem to be a strikingly good outcome. In order to understand what is going on, we can watch the step-by-step behaviour of the algorithm, in order to check whether it makes some obviously ineffective operation, or we can watch its solutions, in order to check whether they have something strange. The following rows show the solutions of the first instances:

```
"File T_A f_A x_A"
data\n0100k010.txt   0.000000     3308    1    6   21   22   31...
data\n0100k020.txt   0.000000    12120    1    2   12   13   15...
data\n0100k030.txt   0.000000    26115    1    4    7   16   17...
data\n0100k040.txt   0.000000    44037    1    2    4    7   12...
data\n0200k020.txt   0.001000    13139    1    3    4    5   35...
data\n0200k040.txt   0.002000    48040    1   13   14   22   23...
```

[2]Their expressions are different from those discussed in the theoretical lessons, because the problem is a maximisation one, but the basic idea is exactly the same.

I	f_A	f^*	δ_A
n100k10	3308	3561	7.10%
n100k20	12120	12541	3.36%
n100k30	26115	26642	1.98%
n100k40	44037	45445	3.10%
n200k20	13139	13489	2.59%
n200k40	48040	48866	1.69%
n200k60	102535	103266	0.71%
n200k80	175407	177263	1.05%
n300k30	27891	29208	4.51%
n300k60	104130	106272	2.02%
n300k90	225757	227346	0.70%
n300k120	388035	391901	0.99%
n400k40	49333	50593	2.49%
n400k80	180929	184820	2.11%
n400k120	394012	397695	0.93%
n400k160	681948	689552	1.10%
n500k50	75918	77937	2.59%
n500k100	279418	285776	2.22%
n500k150	610721	616986	1.02%
n500k200	1062600	1072953	0.96%
n600k10	107626	110064	2.22%
n600k20	400101	407113	1.72%
n600k30	876249	885531	1.05%
n600k40	1521578	1532111	0.69%
n700k20	144837	148024	2.15%
n700k40	544500	550806	1.14%
n700k60	1185633	1197512	0.99%
n700k80	2063246	2078232	0.72%
n800k30	186327	190962	2.43%
n800k60	704682	713263	1.20%
n800k90	1549915	1558378	0.54%
n800k120	2690094	2707534	0.64%
n900k40	235297	240114	2.01%
n900k80	886151	899843	1.52%
n900k120	1948726	1959910	0.57%
n900k160	3393602	3413499	0.58%
n1000k50	288336	293587	1.79%
n1000k100	1091266	1102515	1.02%
n1000k150	2389100	2407636	0.77%
n1000k200	4190140	4207633	0.42%

Table B.1: Results of the basic greedy algorithm

```
data\n0200k060.txt   0.003000   102535    1    5    9   10   11...
...
```

We immediately notice an interesting phenomen: all solutions include point 1. This is certainly strange, given that the different instances of the benchmark have a similar structure, but have been generated independently. It is strange enough to suggest revising the behaviour of the algorithm, possibly step by step. The answer is trivial when performing this revision: in the first iteration of the main loop, which selects the first point of the solution, the selection criterium assigns the same value, equal to zero, to all points. This is because each point $i \in P$ yields a solution $x^{(1)} = \{i\}$ whose value is $f(\{i\}) = 0$, as the sum of the reciprocal distances is necessarily zero. Something is clearly wrong in the selection criterium φ, at least at the first step.

B.3 Alternative constructive heuristics

In order to solve the intrinsic defect of the basic constructive heuristic, we can try to modify something in its design, going back to the basics: the construction graph and the selection criterium. Let us consider three possible proposals:

1. *farthest-pair heuristic*: keep the same selection criterium, but modify the construction graph at the first level (where it is defective), skipping directly from the empty set to pairs of points, instead of singletons; in other words, start with the two reciprocally farthest points, then go back to adding a single point at a time:

$$\varphi(B,x) = \begin{cases} \max\limits_{i,j \in P} d_{ij} & \text{when } x = \emptyset \\ f(x \cup \{i\}) & \text{when } x \neq \emptyset \end{cases}$$

   The asymptotic worst-case complexity increases from $O\left(nk^2\right)$ to $O\left(nk^2 + n^2\right)$, due to the search for the pair of farthest points, but its order does not change unless k is very small.

2. *farthest-point heuristic*: adopt a special selection criterium for the first point $i^{(1)}$: for example, it could be the point farthest away from the other ones:

$$\varphi(i,x) = \begin{cases} \sum\limits_{j \in P} d_{ij} & \text{when } x = \emptyset \\ f(x \cup \{i\}) & \text{when } x \neq \emptyset \end{cases}$$

   The asymptotic worst-case complexity increases from $O\left(nk^2\right)$ to $O\left(nk^2 + n^2\right)$, due to the computation of the total distance from each point to all the other points; however, its order does not change unless k is very small.

3. *try-all heuristic*: use the first point as a parameter and run the algorithm P different times, changing the first point at each repetition. This algorithm strictly dominates the other three, because it includes them: one of the runs certainly makes the same starting choices (point 1, or one of the two farthest points and therefore also the other, or the point at maximum total distance from the other ones), and consequently proceeds in the same way, hitting the same final result. It is however also much more expensive, becase its asymptotic worst-case complexity grows to $O\left(n^2k^2\right)$.

The scaling diagrams for the three new algorithms are given in Figure B.5 in a logarithmic scale, that shows how the last heuristic clearly has a larger slope than the other ones, whereas these have more or less the same. This diagram also describes a basic destructive heuristic that we are going to implement before proceeding with an experimental comparison of all the algorithms.

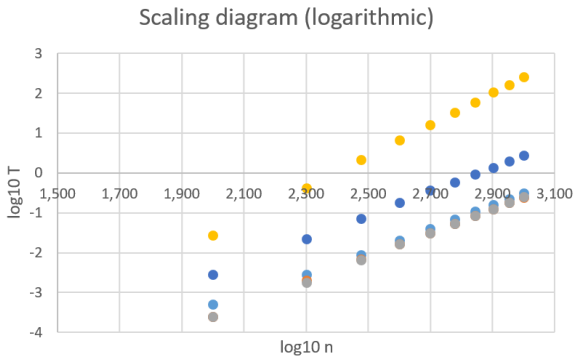


Figure B.5: Scaling diagram for the four greedy algorithms variants (and the stingy one) on the benchmark

B.4 The basic destructive heuristic

A destructive heuristic starts from the overall ground set P and iteratively removes one element at a time, according to a suitable selection criterium, so as to remain inside a suitable search space, until a final solution is found.

If we adopt the objective function as a selection criterium, we can adapt the general scheme of the destructive heuristic to the *MDP* as follows, based on remarks similar to those made for the basic constructive one:

Algorithm StingyMDP(I)
$x := P$;
While $|x| > k$ *do*
    $i := \arg\max_{i \in x} f(x \setminus \{i\})$;
    $x := x \setminus \{i\}$;
Return (x, f);

Notice that maximising $f(x \setminus \{i\})$ corresponds, through computations similar to the ones seen for greedy constructive heuristics, but with a reversed sign, to minimise the (absolute value of the) variation of the objective function $\delta f(x,i) = f(x) - f(x \setminus \{i\})$

$$\delta f(x,i) = \sum_{j \in x} d_{ji} + \sum_{j \in x} d_{ij} + d_{ii} = 2 \sum_{j \in x} d_{ji}$$

which means that the point to be deleted is the one with the maximum total distance from the current subset x.

The first iteration of this algorithm is not problematic as it is for the constructive algorithm: in general, in fact, the removal of different points yields sets with a different value of the objective.

The computational time is predictably larger, because the number of iterations is $n-k$, and each iteration requires to check the total distance of $|x|$ points from other $|x-1|$ points, with $|x|$ decreasing from n to $k+1$. This is approximately $O\left(n^3\right)$, that is of the same order as nk^2, but with larger multiplying factors. In fact, the scaling diagram of Figure B.5 confirms this prediction.

The quality of the results is experimentally better than that of the greedy algorithms (with the obvious exception of the one that tries all starting points). This is rather surprising, because stingy algorithms often tend to perform worse than the greedy ones, due to the larger number of iterations in which they risk to take bad choices.

The reason for this opposite outcome is not clear, but one could conjecture that, in the special case of the *MDP*, choosing the first points to be included in a good solution is actually much more misleading than choosing the firsts to be removed from it. It is a phenomenon that would deserve a more detailed study.

It is however to be remarked that the stingy algorithm cannot be simply considered better than the greedy one because its computational time is much longer.

B.5 Experimental comparison

Solution quality diagrams The SQD diagram is not very significant in itself, because the benchmark considered is rather small and specific. It allows however to compare the different algorithms. See Figure B.6 for the comparison, which clearly confirms the strict dominance of the last heuristic on the previous ones, the probabilistic dominance (at least on the given benchmark) of the stingy heuristic on the greedy ones, whereas the three greedy heuristics are more or less equivalent (with a slight predominance of the farthest point heuristic, especially concerning the maximum gap).

The diagram also shows a very strange phenomenon: the farthest pair heuristic has the same diagram as the basic greedy heuristic. In fact, they not only have the same distribution, but exactly the same results. Is this an unpredicted property of the two algorithms? It does not seem to be necessarily so, and yet this is what can be empirically observed.

The reason is very peculiar and depends on the specific structure of the benchmark. Hence, it is not instructive for the *MDP* in general, but for the need to keep an open eye on the generation of the benchmark. We have extracted the distance values from $\{1,\ldots,100\}$. An instance with n points has exactly $n(n-1)/2$ independent distance values, if we take into account the assumptions made in the introductory section. This means that each point i is very likely to admit at least another point j at distance $d_{ij}=100$. In particular, it is very likely that point 1 admits such another point. But this implies that the farthest pair heuristic will chose a pair including point 1. On the other hand, the basic heuristic, after choosing point 1 will certainly proceed choosing the farthest point from it. Therefore, the two heuristics are very likely to start with the same pair, and, being deterministic, to proceed in the same way and obtain the same final result. This is not intrinsic in the two algorithms, but it depends on the structure of the benchmark instances.

Statistical indices and boxplots A more compact description of the same information can be given by the boxplots of the five heuristics, that are reported in Figure B.7.

Statistical tests Wilcoxon's test can be applied to pairs of algorithms to determine whether any of the two dominates significantly the other one. Since the basic greedy heuristic and the

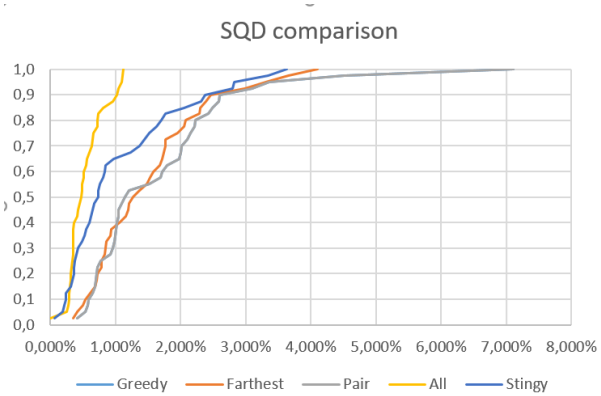


Figure B.6: Solution Quality Distribution diagram for the four greedy algorithms variants (and the stingy one) on the benchmark

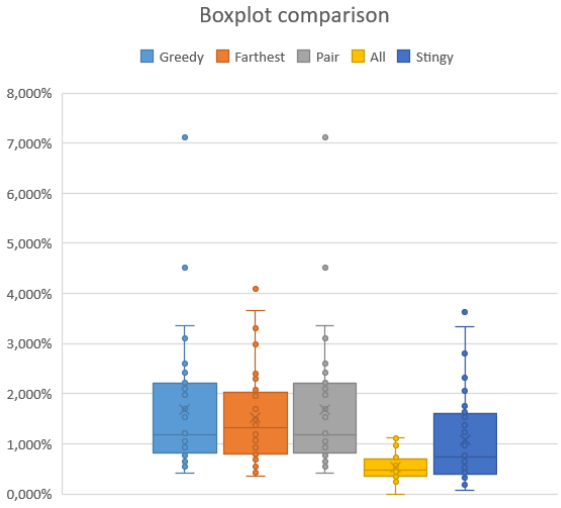


Figure B.7: Boxplots for the four greedy algorithms variants (and the stingy one) on the benchmark

farthest pair heuristic have exactly the same results, it does not make sense to test them. Let us first compare the basic greedy heuristic and the farthest point heuristic. The latter appears slightly better according to Figure **??**, in particular concerning the worst cases (4% versus 7%). Building a text file with two columns reporting the results of the two algorithms allows to run the `SRtest.pl` Perl script that performs Wilcoxon's test. The result is:

```
W+ = 413, W- = 407, N = 40, p <= 0.9732
```

that suggests very similar ranks for the two algorithms, and a very high probability to get such results (or more unbalanced ones) under the null hypothesis that the two algorithms are actually equivalent: $p = 97.32\%$. Therefore, we conclude that the two algorithms are probably equivalent.

Considering the greedy basic heuristic and the stingy heuristic:

```
W+ = 30, W- = 790, N = 40, p <= 3.385e-007
```

the ranks of the two algorithms look very different, with negative ranks prevailing, meaning that the second column (stingy) tends to include larger (that is, better) values. Since the p-value is very small ($3.385 \cdot 10^{-7}$), it looks likely that the stingy heuristic is actually better than the greedy one.

Finally, comparing the stingy heuristic with the "try all" heuristic:

```
W+ = 570, W- = 250, N = 40, p <= 0.03204
```

the latter looks better (positive ranks prevail) with a significant, but not very strong, p-value (3.204% is only slightly lower than the classical 5% threshold).

Other constructive heuristics? The above experiments and remarks open the way to a large variety of possible algorithms, based on more refined definitions of the construction graph or of the selection criterium. Just to mention one, we could take into account the fact that in the final solution each point i will relate with $k-1$ other points. Therefore, estimating its contribution as the distance with respect to the $|x|$ points currently included is certainly incorrect. An estimate of the distances of i from the other $k-1-|x|$ points could be useful. These points are unknown, of course, but it is likely that they are far from x and from i. Therefore, we could consider the points with the largest current total distance from x and from i. This still neglects the reciprocal distances between such points, but could anyway provide a better estimate, and consequently a more effective choice. Of course, computing that information has a computational cost, which must be minimised and weighed with respect to the improvement in the final result.

Open questions Is there a dependence of the solution quality on n and k (see Figure B.8)? Is it different for different algorithms?

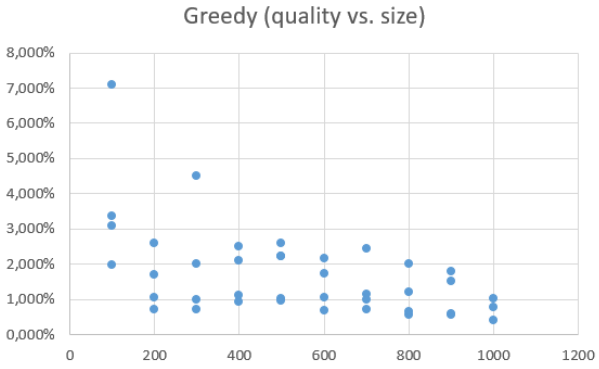


Figure B.8: Scaling diagram for the quality of the greedy algorithm with respect to the size of the instances of the benchmark

APPENDIX C

Laboratory on constructive metaheuristics

This chapter discusses the application of constructive metaheuristics to the *Maximum Diversity Problem* (*MDP*). Constructive metaheuristic try to improve the results of a basic constructive heuristic by running it repeatedly with the introduction of mechanisms that modify its final result. In the end, of course, the algorithm returns the best of the solutions found during the process. The main mechanisms used by metaheuristic algorithms to enhance a constructive heuristic are:

1. the use of different selection criteria, typical of *multi-start* algorithms;
2. the use of *random choices*, typical of *GRASP*;
3. the use of *memory*, typical of the *Ant System*

In the following, we will implement *GRASP* and *Ant System* algorithms for the *MDP*, based on the constructive heuristics discussed in the previous chapter[1]. In the literature, these two approaches require the introduction of exchange procedures to improve the solutions generated by the constructive mechanism. In order to focus on the latter, however, we will avoid them.

Correspondingly, the `main` function allows to choose from the command line which of the two algorithms to apply (with option `-grasp` for the *GRASP* and `-grasp` for the *Ant System*) and to provide the numerical values of the following parameters:

- for the *GRASP* heuristic:
  - the total number of iterations ℓ
  - the randomness parameter μ
- for the *Ant System* heuristic:
  - the total number of iterations ℓ
  - the randomness parameter q
  - the oblivion parameter ρ

plus the seed required to initialise the pseudorandom number generator.

The other operations (loading the data, allocating and deallocating the data and the solution, determining the computational time and printing the results on the screen) are the same as for

[1] Presently, the chapter only includes the *GRASP* algorithm.

the constructive heuristics, except for the fact that also the parameters are printed, so that the report keeps trace of how each single solution was obtained to help guarantee the reproducibility of the results[2]

```
parse_command_line (argc,argv,data_file,algo,&iterations,&mu,&q,&rho,&seed);

load_data(data_file,&I);
//print_data(&I);

create_solution(I.n,&x);

inizio = clock();
if (strcmp(algo,"-grasp") == 0)
  grasp(&I,&x,iterations,mu,&seed);
else if (strcmp(algo,"-as") == 0)
  ant_system(&I,&x,iterations,q,rho,&seed);

fine = clock();
tempo = (double) (fine - inizio) / CLOCKS_PER_SEC;

printf("%s ",data_file);
for (arg = 2; arg < argc; arg++)
  printf("%s ",argv[arg]);
printf("%10.6lf ",tempo);
print_solution(&x);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

C.1 Greedy randomized adaptive search procedure

The *Greedy Randomized Adaptive Search Procedure* (*GRASP*) is a development of the classical *semigreedy* algorithm, that we will actually implement. Its basic idea is to modify the scheme of constructive algorithms by replacing the deterministic choice of the element that provides the best value of the selection criterium

$$i^* := \arg \min_{i \in \Delta^+(x)} \varphi_A(i,x)$$

with a stochastic choice $i^*(\omega)$. This requires to define a probability distribution on set $\Delta^+(x)$, that should be biased so as to favour the best elements over the worst ones:

$$\varphi_A(i,x) \leq \varphi_A(j,x) \Leftrightarrow \pi_A(i,x) \geq \pi_A(j,x)$$

The following pseudocode provides the basic scheme of *GRASP* for maximisation problems, adapted to the specific application to the *MDP* by translating the termination condition into a

[2]Of course, the code could change and yield different results for the same parameter values, but this should no longer be the case when the implementation has reached a sufficiently stable status.

check on the cardinality of the current subset, by returning the last visited subset (as it is the only feasible one) and by replacing the search for a minimum cost solution with the search for a maximum value one. The search procedure after each constructive phase is also neglected:

Algorithm 23 GRASP

```
1:  procedure GRASP(I, ℓ, μ)
2:      x* := ∅; f* := 0;                          ▷ Best solution found so far
3:      for l := 1 to ℓ do
4:          x := ∅;
5:          while |x| < k do                       ▷ Randomised constructive procedure
6:              φ_i := Σ_{j∈x} d_ij for all i ∈ P \ x;   ▷ φ_i := (f(x∪{i}) − f(x))/2
7:              i* := BiasedRandomExtraction(P \ x, φ, μ);
8:              x := x ∪ {i*};
9:          end while
10:         if f(x) > f* then
11:             x* := x; f* := f(x);
12:         end if
13:     end for
14:     return (x*, f*);
15: end procedure
```

The algorithm performs ℓ iterations, where ℓ is a number chosen by the user, based on the available time. It is easy to recognise in the inner **while** a modified version of the basic constructive heuristic presented in the previous chapter. The algorithm still computes all values of $\varphi_A(i,x)$, but, instead of choosing the largest one, it selects one at random based on a biased probability distribution (with parameter μ) that favours the largest ones. At the end of each iteration, the current solution possibly updates the best known one.

The scheme is quite similar to the one used for the greedy "try-all" heuristic, as it requires to create and iteratively fill and empty a current solution, while updating the best known one, that will be returned in the end. We just replace the function `best_point_to_add(px,pI)` used in the basic greedy heuristic, with a function `biased_random_point_to_add(px,pI,pseed,pmu)` that performs the evaluation of the selection criterium $\varphi_A(i,x)$ based on the current solution `x` and on the instance `I`, the extraction of a pseudorandom number depending on the `seed` and the biased stochastic selection of a new point `i` to add based on the parameter `mu` provided by the user.

```
create_solution(pI->n,&x);
for (iter = 1; iter <= iterations; iter++)
{
  while (get_card(&x) < pI->k)
  {
    i = biased_random_point_to_add(&x,pI,mu,pseed);
    add_point(i,&x,pI);
  }

  if (x.f > px->f) copy_solution(&x,px);
  clean_solution(&x,pI->n);
}
```

```
destroy_solution(&x);
```

C.1.1 Choice of the basic constructive heuristic

In the previous chapter, we have considered four alternative (though very similar) constructive heuristics, and a destructive one. we now discuss which of the heuristics should be adopted as the core of the *GRASP* approach.

We will leave aside the destructive heuristic, because it is conceptually different from the other ones, though it would indeed be interesting to compare the constructive and destructive approaches in a metaheuristic allowing to give them equal time[3]. We will also exclude the try-all heuristic, that was the best-performing one, but also the slowest, and in a sense it is already a sort of multi-start metaheuristic, using the initial point as a parameter whose value changes at every iteration. Since a metaheuristic is intrinsically less efficient than the basic heuristic on which it is based, we prefer to choose a simple and fast mechanism, rather than a slow and complex one, at least for a first experiment. There is always time to introduce complications and refinements, if justified by theory or by experience.

The basic heuristic had a strong drawback: it could only generate solutions including point 1. Is this drawback still present in a randomized version? The answer depends on the probability distribution selected. We remind that at the first step the selection criterium is equal to zero for all points. Therefore:

- a scheme based on a *Heuristic Biased Stochastic Sampling* (*HBSS*) would choose any point, but it would assign larger probabilities to the first points;
- a scheme based on a *Restricted Candidate List* (*RCL*) would choose with uniform probability one of the first points.

In the first case, all solutions can be obtained, but there is a bias towards those including the points with small indices, and such a bias is not justified by any good reason. In the second case, the points with larger indices could even be impossible to reach for some instances (that depends also on the values of the distance function). Since for the sake of simplicity, we are going to test only a *RCL* approach, the basic greedy heuristic is not a good choice, unless with some additional correction.

The farthest-point and the farthest-pair heuristic still introduce a bias, or a deterministic advantage, in favour of points with a large total distance, or pairs of very distant points. Such a bias is less unreasonable, but still not provably justified. Moreover, in the case of point pairs, since the number of possible distances is not huge, several pairs of points could have the same distance, and therefore the discrimination between them would end up being based on their indices, and that would not be reasonable.

A very simple idea to avoid an index-based bias at the first iteration of the procedure could be to select the first point at random with uniform probability. At the following step[4], the choice would be stochastically biased in favour of the point that is farthest from a first one selected uniformly at random. That is similar to the farthest-pair heuristic, but different in that at least one of the two indices would be selected at random, without any index-based bias. For the sake of simplicity, we will apply this idea. This means that when x is empty the procedure directly

[3] That's for a future laboratory, but it could be a good exercise.

[4] The following discussion is unrevised brainstorming: I do not expect it to be very clear, or even correct, but I think that the elements discussed are indeed relevant for the behaviour of the algorithm.

selects one of the points with probability $\pi_i = 1/n$ without computing the selection criterium (that would be trivially equal to zero for all points $i \in P$).

When x is not empty, we compute the selection criterium for all external points and proceed to a biased random extraction from the available alternatives. Instead of simply saving the maximum of these values, we save them progressively in a vector `phi` and the corresponding points in a vector `P`, returning the final length `num` of the two vectors, that corresponds to the number of possible extensions $|\Delta^+(x)| = P \setminus x$[5].

```
if (get_card(px) == 0)
  i = get_point(rand_int(1,pI->n,pseed),pI);
else
{
  P = point_alloc(pI->n+1);
  phi = int_alloc(pI->n+1);

  num = compute_selection_criterium(px,pI,P,phi);
  i = biased_random_extraction(P,phi,num,mu,pseed);

  free(P);
  free(phi);
}

return i;
```

The selection criterium is still the value of the objective, that is

$$\varphi_A(i,x) = f(x \cup \{i\}) = \sum_{j \in x \cup \{i\}} \sum_{k \in x \cup \{i\}} d_{jk}$$

replaced for the sake of efficiency by half of its variation $\delta f(x,i) = (f(x \cup \{i\}) - f(x))/2$

$$\delta f(x,i) = \sum_{j \in x} d_{ji}$$

It is therefore computed with the function `dist_from_x(i,px,pI)` that was implemented in the previous chapter.

```
cnt = 0;
for (i = first_point_out(px); !end_point_list(i,px); i = next_point(i,px))
{
  cnt++;
  P[cnt] = i;
  phi[cnt] = dist_from_x(i,px,pI);
}

return cnt;
```

[5]Strictly speaking, this value is therefore already known, so that it would not be necessary to retrieve it from the computation.

C.1.2 Pseudorandom number extraction

In order to generate random numbers, we exploit a classical pseudorandom number generator (the `ran1` generator described in the *Numerical recipes in C*). This is a function that receives in input a *seed*, that is a negative integer number, modifies that number (this is why the seed is passed by reference) and returns in output a real number ω that tends to assume a uniform distribution in the range $[0;1]$ as the function is repeatedly called. The first value of the seed is selected by the user, fed to the algorithm in the command line and determines the overall sequence of numbers generated. This is why the numbers of the sequence are denoted as *pseudorandom*. The ability to generate the same sequence in all runs is fundamental for the repeatability of the approach and the reproducibility of the results. In short, it is a basic condition for the scientific investigation of the problem.

C.1.3 Biased point selection

Given the pseudorandom number ω, it is necessary to determine the corresponding point in $\Delta^+(x) = P \setminus x$ with a biased scheme that favours the points with larger values of ϕ_i. This mechanism depends on the probability distribution adopted. In the following, for the sake of simplicity, we will adopt a *value-based Restricted Candidate List* (*RCL* scheme, in which:

1. a real parameter $\mu \in [0;1]$ is used to fix an intermediate threshold $\bar{\varphi}(x,\mu)$ between the minimum and maximum values available for ϕ_i;

2. the points i such that ϕ_i is better than (i.e., above) the threshold enter the *RCL*;

3. a point is selected from the *RCL* with uniform probability.

Other schemes commonly adopted in *GRASP* heuristics employ a cardinality-based *RCL*, a linearly decreasing or an exponentially decreasing probability profile on all external points. All of them assign decreasing probabilities to the possible choices $i \in \Delta^+(x)$ sorted by nondecreasing values of ϕ. In short, if i_r is the element in position $r = 1,\ldots,|\Delta^+(x)|$ in the ranking, $\phi(i_1,x) \geq \phi(i_2,x) \geq \ldots \geq \phi(i_{|\Delta^+(x)|},x)$. Moreover, the typical schemes adopted in *GRASP* heuristics define probabilities based on the ranking of the choices, and not on the absolute values of the selection criterium, that is $\phi(i_r,x)$ can be expressed as a function of r and $|\Delta^+(x)|$.

C.1.3.1 Value-based *RCL*

This scheme computes an adaptive threshold depending on the values of the selection criterium on the available extension.

$$\bar{\varphi}(x,\mu) = (1-\mu)\,\varphi_{\min} + \mu\,\varphi_{\max}$$

where

$$\varphi_{\min}(x) = \min_{i \in \Delta_A^+(x)} \varphi_A(i,x) \quad \text{and} \quad \varphi_{\max}(x) = \max_{i \in \Delta_A^+(x)} \varphi_A(i,x)$$

and defines a *RCL* as

$$\mathrm{RCL}(x,\mu) = \{i \in \Delta^+(x) : \varphi(i,x) \geq \bar{\varphi}(x,\mu)\}$$

Then, it assigns the elements keeping under the threshold a uniform probability, and the following ones a zero probability:

$$\pi_{i_r} = \begin{cases} \dfrac{1}{|\mathrm{RCL}(x,\mu)|} & \text{if } r \leq |\mathrm{RCL}(x,\mu)| \\ 0 & \text{if } r > |\mathrm{RCL}(x,\mu)| \end{cases}$$

The parameter $\mu \in [0;1]$ tunes the randomness of the choice, with $\mu = 0$ yielding a deterministic heuristic and $\mu = 1$ a random walk.

In order to implement this scheme, the procedure `biased_random_extraction` scans the vector `phi` a first time to determine its minimum and maximum values $\varphi_{\min}$ and $\varphi_{\max}$. Then, it computes the threshold based on parameter μ that discriminates from the other points the elements of the *RCL*, according to condition[6]

$$\varphi_i \geq (1-\mu)\,\varphi_{\max} + \mu\,\varphi_{\min}$$

Scanning again the *RCL* allows to determine the number of its elements and to move them at the beginning of the vector of points `P`. For the sake of efficiency, the points are copied overwriting the previous elements of the vector, because there is no need to keep the whole original content: only the elements of the *RCL* are necessary.

C.1.3.2 Identification of the selected point

Now, in order to extract one of them with uniform probability, it is enough to compute $\lceil \omega\,|\mathrm{RCL}(x,\mu)| \rceil$ and take the corresponding element of vector `P`. In *RCL*-based schemes, it is enough to multiply ω by the size of the *RCL* and round up the result. This provides the position of the point in the list, and therefore allows to access it directly.

The other schemes mentioned above require to first translate the pseudorandom number ω into a ranking position, and then to identify the point corresponding to that position in a sorted vector. Let us focus on the first phase. It is always possible to identify the ranking position that corresponds to ω by summing the probabilities associated to the subsequent rankings, and stopping when the sum reaches ω: the corresponding position is the required one. In practice, however, it is usually not necessary to perform this sum explicitly, since the structure of the values allows to compute the correct ranking more quickly, just as in the case of the *RCL* it was enough to multiply ω by the size of the list[7]

After computing the ranking position, one must still identify the corresponding point according to the order by nondecreasing values of ϕ. It is never required to fully sort vector `phi`: the extraction of the k-th largest element from a vector can in fact be performed in linear time applying suitable algorithms. We do not give details here since we limit our experiments to the *RCL* scheme.

```
phiMin = INT_MAX;
phiMax = -1;
```

[6]The condition is complementary to the one given in the slides, because the selection criterium must be maximised, but μ still measures randomness.

[7]I need to work on this point, that is not clearly discussed in any paper or textbook at the best of my knowledge. I am pretty sure that a linear probability profile allows to give a closed-form quadratic expression of the cumulated probability in each ranking position r, and therefore to find the position corresponding to ω in constant time solving a second-order equation. For the exponential profile, something more sophisticated is required.

```
for (cnt = 1; cnt <= num; cnt++)
{
  if (phi[cnt] < phiMin) phiMin = phi[cnt];
  if (phi[cnt] > phiMax) phiMax = phi[cnt];
}

barphi = (1-mu) * phiMax + mu * phiMin;

RCLsize = 0;
for (cnt = 1; cnt <= num; cnt++)
  if (phi[cnt] >= barphi)
  {
    RCLsize++;
    P[RCLsize] = P[cnt];  /* overwriting P for the sake of efficiency */
  }

cnt = rand_int(1,RCLsize,pseed);

return P[cnt];
```

C.1.4 Empirical evaluation

We can now evaluate the performance of the *GRASP* heuristic. Contrary to what we have done in the previous chapter, having acquired a certain understanding of what the heuristics are doing, we will try to avoid producing meaningless diagrams.

C.1.4.1 Computational time analysis

An *a priori* worst-case asymptotic analysis of the computational time can be based on the similar analysis made for the basic deterministic greedy heuristic. First of all, the constructive heuristic is run for a given number of iterations ℓ. This number is a relevant parameter, that could constant and totally unrelated to the size of the problem, but could also be chosen depending on it, with the idea that larger instances could require more iterations to be explored properly, or vice versa that larger instances allow less iterations because thery require a longer time for each run, and the overall time is limited. In general, therefore, the expression of the time complexity will include ℓ. The basic deterministic heuristic required time $O(nk^2)$ for each run. The randomised version requires additional time for the generation of the pseudorandom number (that can be assumed as constant), for the identification of the minimum and maximum values of the selection criterium and the construction of the *RCL* (that can be assumed as linear), for the biased random selection of the new point (constant time). Consequently, we can estimate an additional linear time per each of the k iterations of the constructive method: the resulting $O(nk)$ term is dominated asymptotically. Other additional terms are given by the comparison of each of the ℓ solutions obtained with the best known one and, possibly, the update of the latter. All these terms are asymptotically dominated, but they could have a perceivable influence on the empirical evaluation.

For the first experiment, we set $\ell = n$. The choice is clearly arbitrary, but it is motivated by the idea to allow each of the n points to be chosen with a reasonable probability as the starting point of the constructive heuristic. It also aims to obtain a computational time comparable to that of

the "try-all" heuristic, which also had n repetitions of the basic constructive scheme. This should allow to better estimate the impact of the additional operations required by the randomisation (not the maintenance of the best known solution, that occurs also in the deterministic heuristic). Figure C.1 reports the *scaling diagram* for the whole benchmark, obtained setting $\mu = 0.1$. It can be noticed, however, that according to the theoretical analysis parameter μ should have very little influence on the computational time. The detailed reports show a limited slow-down as μ increases, but the difference is not significant. The diagram shows the expected increase of the computational time with size, and its logarithmic version in Figure confirms its polynomiality. The $O\left(\ell n k^2\right)$ theoretical estimate, with $\ell = n$ and $k \propto n$, suggests an overall $O\left(n^4\right)$ complexity, that is confirmed by the linear interpolation:

$$T_A = \beta n^\alpha \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

with $\alpha \approx 3.969$ and $\beta \approx 3.4 \cdot 10^{-10}$.

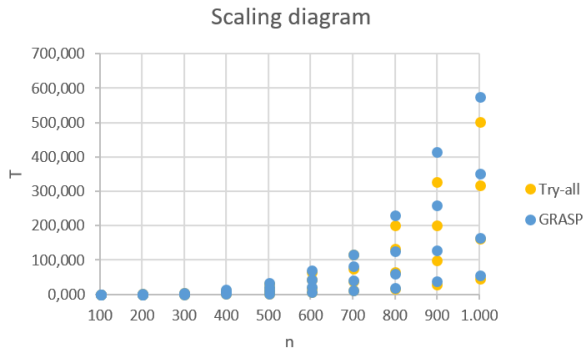


Figure C.1: Scaling diagram for the greedy algorithm on the benchmark

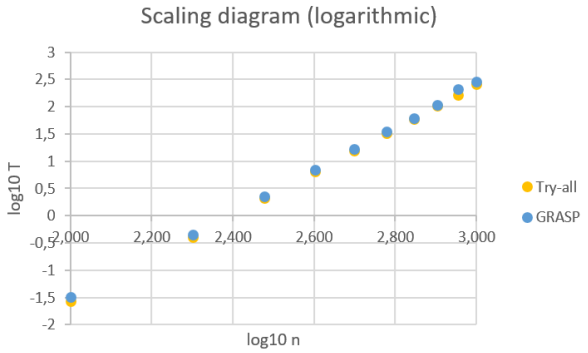


Figure C.2: Scaling diagram in logarithmic scales for the greedy algorithm on the benchmark

The two figures report also the scaling diagram of the greedy "try-all" heuristic, that was expected to have a very similar performance with respect to the computational time. In fact, the two profiles are very similar, with the *GRASP* heuristic only slightly higher, confirming that the additional operations to randomise the selection of points affect very little the overall complexity.

C.1.4.2 Solution quality analysis

Without forgetting that the benchmark is rather small and specific, we can now draw the *SQD* diagram (see Figure **??**), to compare the different parameter tunings with one another. We consider the following tunings: $\mu \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$, after some preliminary experiments showed that larger values provided worse results. The same diagram can be used to compare *GRASP* with the algorithms already developed. In particular, the figure shows the profile of the "try-all" heuristic, that is the most similar one in terms of behaviour (applying a sequence of n different constructive heuristics) and computational time. The diagram is not very clear, but the versions with larger values of the randomness coefficient μ seem to perform slightly worse (the trend becomes clearer for larger values). Indeed, the performance of the "try-all" heuristic is similar to that of the smaller values of μ and better than the other ones. This is rather disappointing, and poses the question whether the values tested are too large and should be reduced. In order to be sure, one would need to check the typical length of the *RCL* during the search. A rough estimate, based on the (unproved, but not unreasonable) assumption that the values of the selection criterium are uniformly distributed between $\varphi_{\min}$ $\varphi_{\max}$ is that the typical size decreases from μn to $\mu(n-k)$. For $\mu = 0.01$, n ranging from 100 to 1000 and k ranging from $0.1n$ to $0.4n$, this corresponds to sizes ranging from 1 to 9, that do not seem so large.

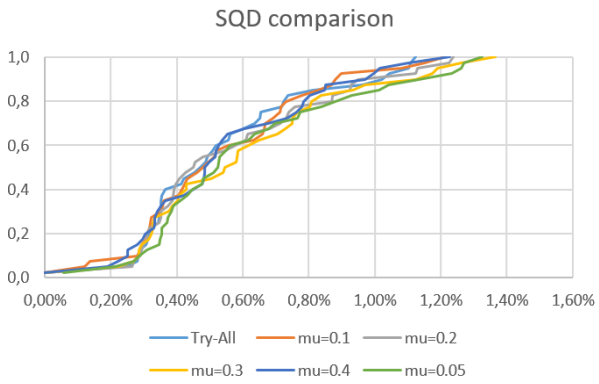


Figure C.3: Solution Quality Distribution diagram for the *GRASP* algorithm (with $\mu \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$) and the greedy "try-all" heuristic

The boxplots reported in Figure C.4 provide a similar, perhaps slightly clearer, intuition. It should anyway be noticed that the *GRASP* heuristic can be prolonged for more than $\ell = n$ iterations, probably improving the final results, whereas the deterministic heuristic cannot. As well, we could experiment with shorter runs, to determine whether the results obtained actually could require a lower number of iterations. Anyway, the results remain unpromising.

C.1.4.2.1 Statistical tests We now apply Wilcoxon's test to determine whether it can discriminate between the results of different parameter tuning of *GRASP* and the deterministic competitor. The results are:

- for $\mu = 0.01$

```
W+ = 338.50, W- = 402.50, N = 38, p <= 0.6478
```

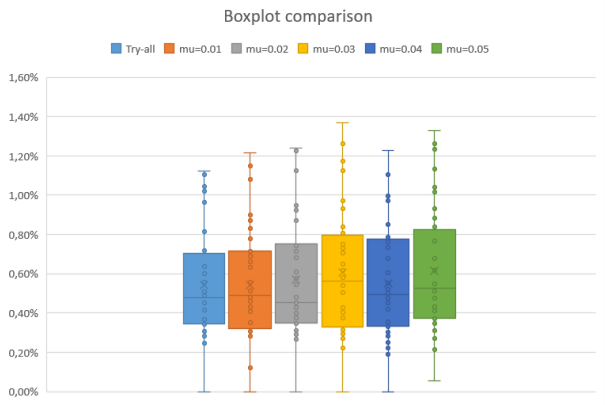


Figure C.4: Boxplots for the *GRASP* algorithm (with $\mu \in \{0.1, 0.2, 0.3, 0.4\}$) and the greedy "try-all' heuristic'

- for $\mu = 0.02$

```
W+ = 430.50, W- = 310.50, N = 38, p <= 0.3882
```

- for $\mu = 0.03$

```
W+ = 523.50, W- = 256.50, N = 39, p <= 0.06345
```

- for $\mu = 0.04$

```
W+ = 455.50, W- = 324.50, N = 39, p <= 0.3644
```

- for $\mu = 0.05$

```
W+ = 634.50, W- = 145.50, N = 39, p <= 0.0006617
```

In short, the difference, at first not statistically significant, tends to become more significant as μ increases (with an exception for $\mu = 0.04$. The only comparison that seems to exhibit a true dominance is the one between the deterministic heuristic and $\mu = 0.05$. The message is clearly not to exceed with randomicity.

C.1.4.2.2 Influence of the random seed An aspect that must be analysed when using random steps is the influence that randomness has on the final result obtained. If we run the algorithm a single time, in fact, the quality of the results achieved could be easily due to a lucky, or unlucky, choice of the random seed. In order to estimate the role of randomness, we should run the algorithm for several times, with different random seeds, and compare the results thus obtained. A simple index is given by the average quality of the solution with respect to a sufficiently large number of runs (at least 10, possibly more), but other indices of distribution are certainly relevant: the maximum and minimum values obtained, or the medians and quartiles.

The description is very similar in principle to that we have given with respect to the benchmark instances (numerical indices, boxplots, *SQD* diagrams), but it considers single instances and variable seeds, instead of a single seed and random instances.

In order to give an idea of this kind of investigation, we select a single instance, that we consider as significant for some reason, and we run the algorithm for a given number of times with different seeds. We will consider the instance `n0600k060`, because it is the instance on which the *GRASP* algorithm with $\mu = 0.01$, that is the best performing one on average (though with nonsignificant differences with other ones) obtains the largest gap: $\delta_A(I) = 1.21\%$ (see the maximum of the second boxplot in Figure C.4. The question investigated is whether this bad result was typical or derived from a particularly unlucky, or particularly lucky, choice of the random seed. In order to establish this, since the algorithm takes about 5 seconds to solve the instances, we run it 100 times with different random seeds, ranging from -1 to -100. Figure C.5 provides the *SQD* with respect to the random seed. The gap δ is never huge, but indeed it varies in a rather large range (between 0.6% and 1.4%). The red line in the picture, that corresponds to the single run of the previous phase of experiments, suggests that its result was indeed rather unlucky. This suggests that the *GRASP* algorithm (at least on this instance) is rather unstable, that its results could be in practice better (but also worse) than the ones discussed above, that were obtained in a single (not necessarily representative) run, and that the conclusions drawn from such results should be handled with much care.

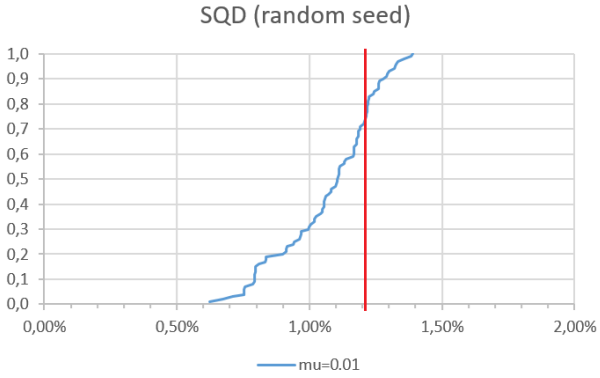


Figure C.5: Solution Quality Distribution diagram for the *GRASP* algorithm with $\mu = 0.01$ on instance `n0600k060` with 100 different random seeds: the red line corresponds to the single run of the previous phase of experiments.

C.2 Ant System

The *Ant System* (*AS*) is a development of the classical *cost perturbation* algorithm, as well as of the *semigreedy* algorithm. Its basic idea is to modify the scheme of constructive algorithms by replacing the deterministic choice of the element that provides the best value of the selection criterium with a stochastic choice, influenced by additional information provided by the memory of previously found solutions. In addition, the *AS* considers a *population* of algorithms that work in parallel, iteration by iteration. Among the several variants of *AS*, we are going to implement the one that:

- defines the *visibility* of a point $i \in P \setminus x$ as the selection criterium used in the constructive and the *GRASP* heuristic, that is
$$\eta(i,x) = \varphi(i,x) = \sum_{j \in x} d_{ji}$$
- maintains in a suitable vector a *trail* function $\tau(i)$ that depends only on the point i to be added, and is progressively updated during the execution;
- combines visibility and trail by multiplying them, as they both tend to associate larger values to better options;
- tunes the randomness of the choice with a parameter q, so that the choice is made selecting with probability $1-q$
$$i^* = \arg\max_{i \in P \setminus x} \varphi(i,x)\,\tau(i)$$
and with probability q a random point with probability distribution
$$\phi_i = \frac{\varphi(i,x)\,\tau(i)}{\sum_{j \in P \setminus x} \varphi(j,x)\,\tau(j)}$$
- applies a local update to the trail to diversify the search after each individual has built a solution;
- applies a global update to the trail to intensify the search on the points that belong to the best known solution found in the whole process.

The following pseudocode provides the scheme of this variant of the *AntSystem* adapted to the *MDP* as already done for *GRASP*. Also in this case the search procedure that should be run to improve the solutions built is neglected:

The algorithm performs ℓ iterations, in each of which it generates h different solutions; ℓ and h are numbers chosen by the user, based on the available time. It is easy to recognise in the inner **while** a modified version of the basic constructive heuristic presented in the previous chapter. The algorithm still computes all values of $\varphi_A(i,x)$, but, instead of choosing the largest one, it selects one at random based on a biased probability distribution (with parameter μ) that favours the ones with largest values of φ and τ. At the end of each iteration, the current solution possibly updates the best known one.

Algorithm 24 AntSystem

procedure ANTSYSTEM(I, ℓ, q, ρ)
  $x^* := \emptyset$; $f^* := 0$; ▷ Best solution found so far
  $\tau_i = \tau_0$ for all $i \in P$;
  for $l := 1$ **to** ℓ **do**
    for $g := 1$ **to** h **do**
      $x := \emptyset$;
      while $|x| < k$ **do** ▷ Randomised constructive procedure
        $\varphi_i := f(x \cup \{i\}) - f(x)$ for all $i \in P \setminus x$;
        $i^* :=$ BiasedRandomExtraction$(P \setminus x, \varphi, \tau, \mu)$;
        $x := x \cup \{i^*\}$;
      end while
      if $f(x) > f^*$ **then**
        $x^* := x$; $f^* := f(x)$;
      end if
      $\tau :=$ LocalTrailUpdate(x, τ, ρ);
    end for
    $\tau :=$ GlobalTrailUpdate(x^*, τ, ρ);
  end for
  return (x^*, f^*);
end procedure

APPENDIX D

Laboratory on exchange heuristics

asdasdas

This chapter discusses the application of exchange heuristics to the *Maximum Diversity Problem* (*MDP*). Exchange heuristic start from a given feasible solution $x^{(0)}$ (typically obtained with a constructive heuristic, or metaheuristic, or a random generation process) and try to improve the current solution x iteratively by adding a suitable subset A and deleting a suitable subset D of elements of the ground set. Of course, A consists of external elements ($A \subseteq B \setminus x$) and D of internal ones ($D \subseteq x$). The possible pairs of subsets are determined by a rule that takes the form of a *neighbourhood* function $N : X \to 2^X$, associating each feasible solution $x \in X$ with a subset of feasible neighbour solutions $N(x)$. The choice of the *incumbent*, that is the neighbour solution that replaces the current one, is done optimising a suitable selection criterium, that nearly always is the objective function value.

In the following, we will consider some alternative initialisation procedures (namely, the farthest point and the try-all constructive heuristics and a purely random generation). We will also adopt the most natural neighbourhood for the *MDP*, that is the *single-swap neighbourhood* N_{S_1}, which includes all the subsets obtained from x deleting a single element i and adding a single element j:

$$N_{S_1}(x) = \left\{x' = x \setminus \{i\} \cup \{j\} \text{ with } i \in x, j \in P \setminus x\right\}$$

Notice that $N_{S_1}(x) = N_{H_2}(x)$, that is, it coincides with the collection of feasible subsets having Hamming distance equal to 2 from x. However, the collection of all subsets at Hamming distance equal to 2 also includes the ones obtained adding or deleting two points, which are unfeasible. The single-swap neighbourhood, on the contrary, automatically satisfies the cardinality constraint that characterises the *MDP*, and this implies the strong advantage that the feasibility of the subset obtained with any swap operation is guaranteed *a priori* and needs not be verified. The exploration of the neighbourhood, therefore, simply consists in the computation of the objective function for each neighbour solution.

Thanks to the cardinality constraint, and to the lack of other complicating constraints, neighbourhood N_{S_1} always includes exactly $k(n-k)$ solutions, and this induces a strong relation between the number of neighbourhood explorations and the computational time (at least, if the neighbourhood is fully explored).

Finally, concerning the selection criterium, we will adopt the objective function, thus implementing the basic exchange heuristic known as *steepest ascent* (for maximisation problems as the *MDP*). We will discuss its theoretical and empirical computational complexity, and we will improve it with a standard trick to allow the evaluation of quadratic objective functions in cardinality-constrained problems. We will compare the results of the different initialisation

procedures and we will tune the size of the neighbourhood with the adoption of the *first-best* exploration strategy as opposed to the *global-best* one.

The `main` function, then, allows to choose from the command line which of the three initialization procedures to apply (with option `-gp` for the *farthest-point*, `-ga` for the try-all heuristic and `-r` followed by a negative integer seed for the random initialisation), and which of the two neighbourhood exploration strategies to apply (with option `-gb` for the *global-best* and `-fb` for the *first-best* strategy). The steepest ascent heuristics also returns the number of neighbourhood explorations performed, because we are going to investigate the influence of the exploration stategy on this value and its relation with the computational time. Apart from printing the number of iterations, all other operations are the same introduced for the previous heuristics.

```
parse_command_line(argc,argv,data_file,init_algo,visit_strategy,&seed);

load_data(data_file,&I);

create_solution(I.n,&x);

start = clock();
if (strcmp(init_algo, "-gf") == 0)
  greedy_farthest(&I, &x);
else if (strcmp(init_algo, "-ga") == 0)
  greedy_tryall(&I, &x);
else if (strcmp(init_algo, "-r") == 0)
  generate_random_solution(&I, &x, &seed);
steepest_ascent(&I,&x,visit_strategy,&niter);
end = clock();
tempo = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6f ",tempo);
printf("%8d ",niter);
print_sorted_solution(&x,I.n);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

D.1 The steepest ascent heuristic

The *steepest ascent* heuristic chooses the following solution from the neighbourhood of the current one by optimising a selection criterium $\phi(x,i,j)$ that is simply the value of the objective function $f(x \cup \{i\} \setminus \{j\})$ or, to be more precise, half of its variation:

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = \sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

This implies the following adaptation to the *MDP* of the general scheme:

Algorithm SteepestDescentMDP$\left(I, x^{(0)}\right)$

$x := x^{(0)}$;

Stop := *false*;

While Stop = *false do*

$\tilde{x} := \arg \max_{i \in x, j \in P \setminus x} \left(\sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij} \right)$;

If $f(\tilde{x}) \leq f(x)$ *then* Stop := *true*; *else* $x := \tilde{x}$;

EndWhile;

Return $(x, f(x))$;

The implementation of this algorithm is nearly straightforward. The main difference is that, for the sake of efficiency, the procedure `explore_neighbourhood` that selects the incumbent returns a pair of points (i^*, j^*) to be exchanged, and the resulting variation of the objective, instead of a whole solution $\tilde{x}$. Therefore, if the incumbent improves the current solution (that is, the variation is negative), the update operation $x := \tilde{x}$ is obtained performing the exchange suggested with procedure `swap_points`.

```
*pniter = 0;
do
{
  explore_neighbourhood(px,pI,visit_strategy,&p_in,&p_out,&delta_f);
  if (delta_f > 0)
  {
    (*pniter)++;
    swap_points(p_in,p_out,px,pI);
  }
} while (delta_f > 0);
```

It could be remarked that swapping points j and i is equivalent to deleting point j and adding point i, so that we do not actually need an additional procedure. However, implementing this procedure separately has the advantage to avoid any instruction that is useless when the two operations must be performed together (to give a trivial example, the cardinality of the solution remains unvaried, instead of being decremented and incremented).

The exploration of the neighbourhood trivially consists in two nested loops, as j scans the current solution x and i scans its complement, taking advantage of the corresponding lists. For each pair of solutions, the procedure estimates the variation of the objective function $\delta f(x, i, j)$ calling a suitable procedure `evaluate_exchange` and saves the best exchange and the corresponding variation of the objective.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px);
      p_in = next_point(p_in,px))
  for (p_out = first_point_out(px); !end_point_list(p_out,px);
        p_out = next_point(p_out,px))
  {
    delta_f = evaluate_exchange(p_in,p_out,px,pI);
```

```
    if (delta_f > *pdelta_f)
    {
      *pdelta_f = delta_f;
      *pp_in = p_in;
      *pp_out = p_out;
    }
  }
```

The procedure that evaluates each single exchange does not physically perform it. *Do not perform moves only to evaluate them* is a specific application of a general fundamental principle in the design of heuristic algorithms: *avoid all useless operations*. It simply computes

$$\sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

as follows[1].

```
delta = 0;

delta  = dist_from_x(p_out,px,pI);
delta -= dist_from_x(p_in,px,pI);
delta -= pI->d[get_index(p_in,pI)][get_index(p_out,pI)];

return delta;
```

D.1.1 Time complexity estimation

The computational complexity of the steepest ascent heuristic derives from three main sources:

1. the number of neighbourhood explorations $t_{\max}$ performed to reach the local optimum in which the search terminates;
2. the number of neighbour solutions (or, in general, subsets) whose objective value (and, in general, feasibility) is evaluated;
3. the computational time required to evaluate the objective value (and, in general, the feasibility) of each neighbour solution (or subset).

The first term is in general unknown and hard to estimate (unless for upper estimates such as the total number of feasible solutions, that are very loose). For the single-swap neighbourhood N_{S_1}, the number of neighbour solutions is exactly $k(n-k)$. The feasibility is automatically guaranteed, and the evaluation of the objective requires to compute the distance of two points from the current solution, that is $O(k)$ time. The resulting overall estimate is $O\left(t_{\max}(n-k)k^2\right)$.

D.1.2 Empirical evaluation

We can now evaluate the performance of the *steepest ascent* heuristic with the global-best strategy.

[1] This is just a detail, but it is probably better to add and remove d_{ij} rather than checking at every step whether $k = j$ or not.

D.1.2.1 Computational time analysis

From the detailed results, we can remark that the overall computational time (including both the initialisation constructive procedure and the following exchange procedure) ranges from fractions of a second to a couple of minutes. This is two orders of magnitude larger than the time required for the constructive heuristic alone (up to half a second), and therefore mostly depends on the exchange phase. It is comparable to the try-all heuristic only for the smaller instances, and one order of magnitude smaller for the larger ones (in fact, the try-all heuristic repeats the basic constructive heuristic n times, with n ranging from 100 to 1 000). The exchange heuristic is therefore less efficient than the constructive (and destructive) ones, but more efficient than the try-all heuristic.

Figure D.1 reports the semilogarithmic *scaling diagram* for the computational time of the steepest ascent heuristic on the whole benchmark. The diagram shows the expected polynomial increase with size. The $O\left(t_{\max} n k^2\right)$ theoretical estimate, with $k \propto n$, suggests an overall $O\left(t_{\max} n^3\right)$ complexity. The linear interpolation:

$$T_A = \beta n^\alpha \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

suggests that $\alpha \approx 4.3$ and $\beta \approx 5.75 \cdot 10^{-12}$. If we assume a cubic complexity for the neighbourhood exploration, this would imply that $t_{\max}$ increases slightly more than linearly with n. To test more precisely this conclusion we can compute and plot the ratio T/t_{max} of the total time T on the number of neighbourhood explorations t_{max} (see the yellow graph in Figure D.1) and make an interpolation on it. Since $\alpha \approx 2.7$ and $\beta \approx 3.6 \cdot 10^{-9}$, it seems that the cubic estimate for the time required to explore a single neighbourhood is excessive, and that $t_{\max}$ is more than linear in n, though not quadratic. One can also notice that the first diagram is much more irregular than the first one, meaning that $t_{\max}$ is not strictly dependent on n. Of course, we could also directly interpolate t_{max} as a function of size: a quick look at the detailed results for each fixed value of n suggests that t_{max} indeed strongly depends on k, increasing more than linearly: it becomes about 10 times larger as k goes from $0.1\,n$ to $0.4\,n$.

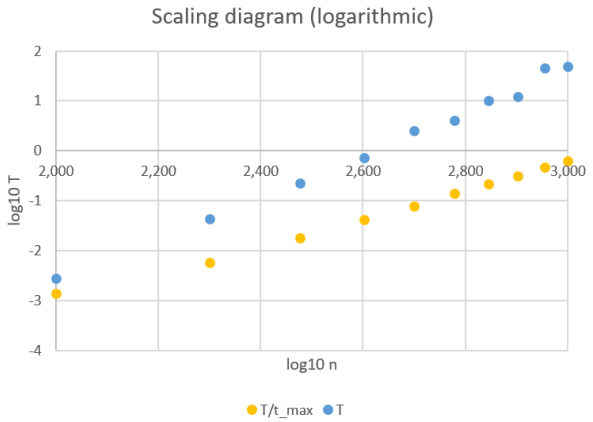


Figure D.1: Scaling diagram in logarithmic scales for the steepest ascent algorithm on the benchmark

D.1.2.2 Solution quality analysis

Figure D.2 reports the *SQD* diagram, compared with that of the initialisation farthest-point procedure. Of course, the former strictly dominates the latter, as it receives the solution in input and proceeds by improving it with local search. What is interesting is the amount of the improvement, that is strong, but not huge: the average gap δ decreases from 1.52% to 0.97%. This suggests that the *MDP* has many local optima of various quality and with small basins of attraction. In fact, the try-all heuristic still performs better than the steepest ascent. Of course, as it takes much more time, we can't say that the latter is dominated. However, it is still impossible to dismiss the try-all heuristic as a viable approach. One can notice that the steepest ascent heuristic has a larger probability of finding very small gaps (below 0.3%), though not of finding the best known result. This suggests a region of stronger stability, possibly corresponding to the capacity of improving good initial results.

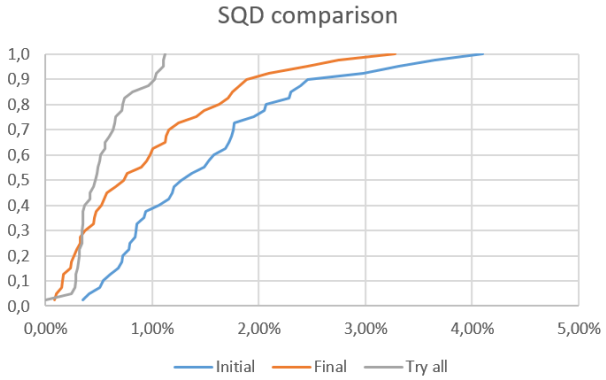


Figure D.2: Solution Quality Distribution diagram for the *steepest ascent* and the *farthest point* heuristics

The boxplots reported in Figure D.3 provide the same information.

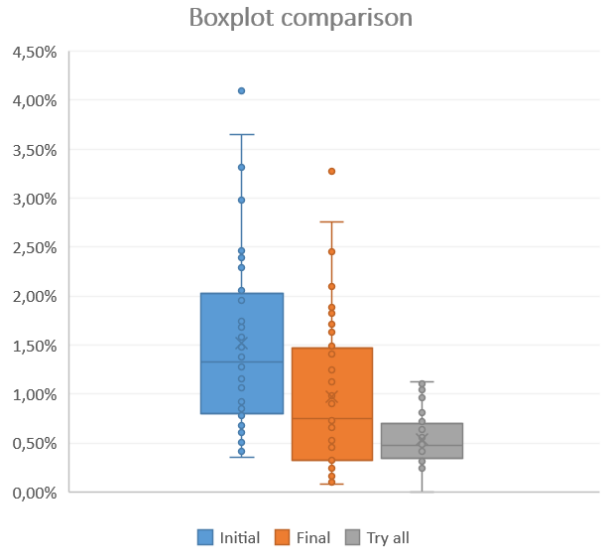


Figure D.3: Boxplots for the *steepest ascent* and the *farthest point* heuristics

We do not apply statistical tests to this comparison, because by definition the dominance of steepest ascent with respect to the farthest point heuristic is strict and the tests would not add anything to this fundamental information.

D.1.3 Constant-time neighbour evaluation

From theory we know that the variation of a quadratic objective function implied by a simple swap of elements can be estimated in constant time exploiting the formula

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = \sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

by saving in a suitable vector D_i the total distance of each point $i \in P$ from the current solution x, both for internal and external points. In fact, given this vector

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = D_j - D_i - d_{ij}$$

can be computed in two operations. Of course, whenever the current solution x changes into $x' = x \setminus \{i\} \cup \{j\}$, the vector D must be updated. This takes time $O(n)$ applying the following formula

$$D_k := D_k - d_{ik} + d_{jk} \qquad \text{for all } k \in P$$

Overall, this reduces the time to explore a single neighbourhood from $O\left((n-k)k^2\right)$ to $O\left((n-k)k\right)$, at the cost of adding an $O(n)$ term, that is dominated.

From the implementation point of view, we need to decide where to store vector D. The most natural approaches are either to keep it explicitly as a variable in procedure `steepest_ascent` or to "hide" it in solution x. In the former case, we will have to pass it as an argument to procedure `explore_neighbourhood`. In the latter case, we will have to update all the functions that manipulate objects of type `solution_t`. The choice mainly depends on whether we think that the vector will be used outside of the exchange heuristic or not. As we are going to use it also in the exchange metaheuristics, we will adopt this approach. Moreover, it make sense to go back to constructive heuristics and metaheuristics and evaluate whether the computational trick would provide an advantage also in those algorithms. Indeed, the $O\left(nk^2\right)$ complexity of the constructive heuristics depends on applying k times the basic constructive step, in which for all external points (hence, the $O(n)$ term) the distance from the current solution is estimated in time $O(k)$. The introduction of vector D would remove that term, and therefore the total time to $O(nk)$ at the cost of an additional $O(n)$ term, that is dominated. All the required work auxiliary work has been done in library `solution2`, that we will use in the following instead of `solution`. This will also require to modify accordingly the inclusion directive in file `local_search.h`.

Adding vector D to the `solution_t` data structure under the form of a dynamic integer vector (`int *`) requires to update the creation, destruction, copy and check procedures, as well as the manipulation procedures (`add_point`, `delete_point` and `swap_points`). After this update, it is possible to implement the evaluation of the exchange simply as follows.

```
delta  = px->D[get_index(p_out,pI)];
delta -= px->D[get_index(p_in,pI)];
delta -= pI->d[get_index(p_in,pI)][get_index(p_out,pI)];

return delta;
```

On the other hand, the `swap_points` procedure must also update the elements of vector D (in $O(n)$ time), but it can save $O(k)$ time for the update of the objective value.

Solving again the whole benchmark leads to exactly the same results as above, as expected. The computational time is however much smaller: in Figure D.4, the logarithmic scale clearly shows a decrease in the slope, corresponding to a reduction of the exponent in the polynomial dependence of the computational time on the number of points[2]. Just to have an intuitive idea of the improvement, the computational times now range from fractions of a second to 1 second, instead of two minutes. This impressive result derives from having reduced the theoretical worst-case complexity from $O(t_{\max} nk^2)$ to $O(t_{\max} nk)$. The time is not strictly reduced by a factor of k (from 10 to 400 in the benchmark) because the contribution of secondary terms previously overwelmed by the evaluation of the value for the neighbour solutions now can becomes perceivable.

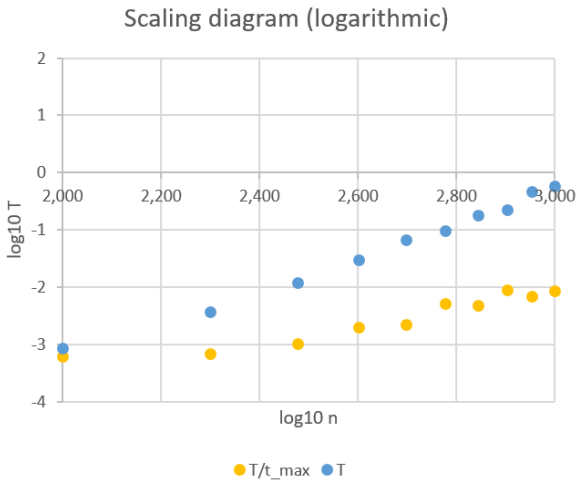


Figure D.4: Scaling diagram in logarithmic scales for the steepest ascent algorithm with the constant-time or the linear-time evaluation of neighbours

D.1.4 Comparison of initialisation procedures

We now investigate whether the initial solution $x^{(0)}$ exerts a long-term influence on the quality of the final solution returned by the exchange heuristic; in other words, whether good starting solutions tend to fall into the basin of attraction of good local optima. This would recommend the adoption of a good constructive procedure, provided that its computational time is not excessive. To investigate also this aspect, we consider a third variant in which the initial solution is obtained with the "try-all" heuristic[3].

Figure D.5 shows the three *SQD* diagrams for the initial solutions and for the final ones. The diagrams are in semilogarithmic scale because the gaps are widely distributed (part of the dia-

[2] Actually, α decreases from 4.3 to 2.9 for the overall time and from 2.7 to 1.3 for the time per iteration, which seems rather too much, but I have had no time to check the numbers and to investigate the reason of this behaviour.

[3] It is clear that this heuristic could be fully exploited by running an exchange procedure on each of the starting solutions it provides, but we are now focusing on the improvement power of a single run of an exchange heuristic. Applying the exchange heuristic to each solution generated by the "try-all" heuristic would be more on the line of a multi-start exchange metaheuristic. The same can be said about the classical *GRASP* mechanism that applied an exchange procedure to each solution generated by the semigreedy algorithm.

grams is missing because zero values cannot be represented on a logarithmic axis). This allows to appreciate the relative improvement obtained by the exchange procedure with respect to each different starting point This is particularly strong in the case of the random initialisation, whereas it becomes smaller for the constructive initialisations. In particular, it can be noticed that the worse starting solutions are only slightly improved (in the case of the try-all heuristic, the second-worse solution is not improved, so that the diagram appears to reach the upper bound nearly for the same gap). It is however clear that better starting solutions tend to be associated with better final solutions: the exchange heuristic is not strong enough to overcome the initial advantage.

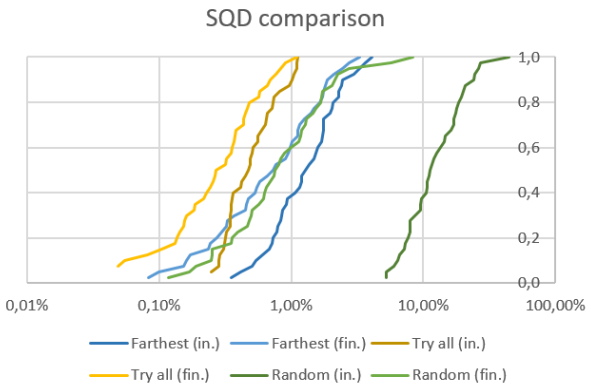


Figure D.5: Comparison of different initialisation procedures

Of course, the computational times are also relevant for the choice. Notice that we can apply the try-all heuristic only because the constant-time evaluation procedure can be extended to the constructive phase, so that this heuristic no longer takes several minutes to provide the starting solution for the larger instances. Indeed, its overall time requirement (constructive and improvement phase) ranges from fractions of a second to about 5 seconds, as opposed to about 1 second for the farthest point initialisation and 1.6 seconds for the random initialisation. The try-all initialisation is slower due to the more refined constructive phase; in fact, the number of neighbourhood explorations before reaching the local optimum is on average 15. The random initialisation has an extremely fast initialisation, but an average of 100 iterations. That implies the final longer time with respect to the farthest point heuristic, which only makes 25 iterations before hitting the local optimum.

Figure D.6 provides the *RTD* diagrams on the benchmark. We remind that such a diagram makes little sense for benchmarks collecting instances of different size, but it allows meaningful comparisons between different algorithms.

This suggests that a random initialisation is not a good idea (at least for the steepest ascent heuristic), whereas it is still open whether the farthest point or the try-all initialisation (or other non fully random ones) could be more effective.

D.1.5 Neighbourhood tuning: *global-best* versus *first-best*

Finally, we experiment with the idea of tuning the exploration of the neighbourhood, that is terminating it as soon as an improving solution is found with respect to the current one. This is known as *first-best* strategy, as opposed to the classical *global-best* strategy that visits the whole neighbourhood and returns the overall best solution it contains. The rationale is to accept

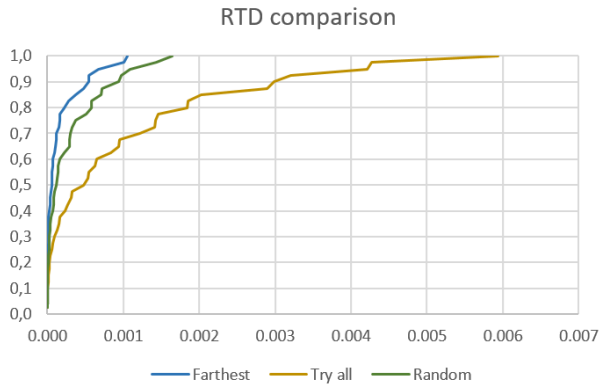


Figure D.6: Comparison of the overall running time of the exchange heuristic for different initialisation procedures

a smaller improvement in each step of the search, in exchange for a much smaller computational time, that allows to perform many more steps, possibly getting earlier to the same local optimum. It must be noticed that, changing the rule that determines the following visited solution in general also changes the basins of attraction, and therefore the final local optimum reached. This could be worse or better than in the global-best case. To make things even more complex, the local optimum returned also depends on the order in which each neighbourhood is visited, a fact that could be perhaps exploited somehow (possibly, starting with the most promising elements based on the distance D from x), with a possible increase of the computational cost. We are not going to explore this line of research: the neighbourhood will be explored scanning x and $P \setminus x$ with the corresponding lists, exactly as in the global-best strategy.

In order to impose the first-best strategy, an extremely simple modification must be made to the neighbourhood exploration procedure: as soon as an exchange with a positive effect is found, the procedure terminates returning that exchange.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px);
          p_in = next_point(p_in,px))
  for (p_out = first_point_out(px); !end_point_list(p_out,px);
          p_out = next_point(p_out,px))
  {
    delta_f = evaluate_exchange(p_in,p_out,px,pI);
    if (delta_f > *pdelta_f)
    {
      *pdelta_f = delta_f;
      *pp_in = p_in;
      *pp_out = p_out;
      if ( (delta_f > 0) && (strcmp(visit_strategy,"-fb") == 0) ) break;
    }
  }
```

For the sake of simplicity, we apply the farthest-point initialisation heuristic. The detailed results show that the number of iterations tends to be larger with respect to the global-best strategy,

as expected, but the difference is not strong (30 versus 25 on average). In fact, the computational time is shorter, and the results comparable. Figure D.7 shows that the two algorithms have rather similar performance with respect to the solution quality (perhaps, the first-best strategy is slightly better).

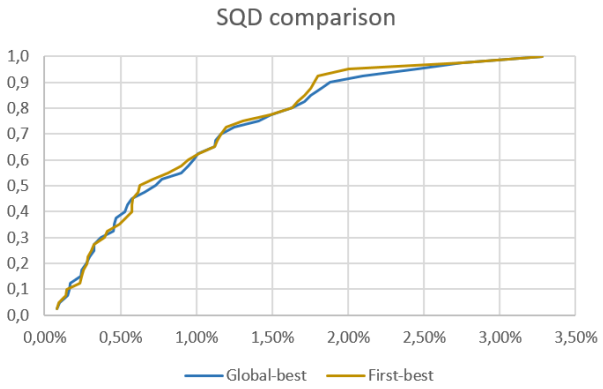


Figure D.7: Comparison of the solution quality of the exchange heuristic with the global-best and first-best neighbourhood exploration strategies

Figure D.8, however, shows that the first-best strategy is clearly faster. This could be enough to suggest to adopt it instead of the classical global-best strategy.

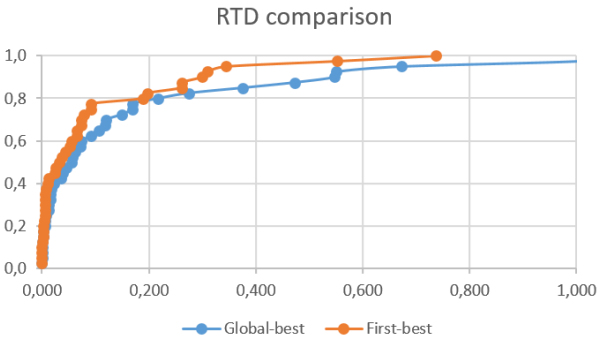


Figure D.8: Comparison of the computational time of the exchange heuristic with the global-best and first-best neighbourhood exploration strategies

In order to further support this choice, we can compare the two sets of results with Wilcoxon's test. The results concerning the solution quality are:

```
W+ = 168, W- = 297, N = 30, p <= 0.188
```

and suggest that, while there is a slight predominance of the first-best strategy with respect to the global-best, it would not be unlikely that such a predominance be due only to a random extraction (the p-value is 18.8%, that is quite large).

On the other hand, the results for the computational time are:

```
W+ = 677.50, W- = 25.50, N = 37, p <= 9.094e-007
```

and indeed suggest that the global-best strategy takes a longer time, even if the first-best strategy requires more neighbourhood explorations (the p-value is approximately 10^{-6}).

APPENDIX E

Laboratory on exchange metaheuristics

This chapter discusses the application of exchange metaheuristics to the *Maximum Diversity Problem* (*MDP*). Exchange metaheuristic extend the basic scheme of exchange heuristics modifying its elements (the starting solution, the neighbourhood, the objective function or the selection rule) in order to proceed with the search after reaching a locally optimal solution. In the following, we will consider two such extensions, both based on the basic *steepest ascent* heuristic described in the previous chapter. The former is a *Variable Neighbourhood Search* (*VNS*) heuristic, that restarts the search from a new solution generated with a *shaking* procedure on the best known solution. The latter is a *Tabu Search* (*TS*) heuristic, that prolongs the search beyond local optima by looking for the minimum cost neighbour solution that respects suitable *tabu* conditions, designed to avoid a cyclic behaviour.

For the sake of simplicity, we will build the starting solution with the farthest-point constructive heuristic introduced in Chapter 3 and apply the steepest ascent heuristic with the *first-best* strategy that proved to be equally effective and more efficient in Chapter 4. The command line arguments will allow the user to choose which of the two metaheuristics to apply and the associated parameters:

- for the *VNS* metaheuristic, option `-vns`, followed by five parameters: the total number of neighbourhood explorations $t_{\max}$, the minimum neighbourhood index $s_{\min}$, the index variation δs, the maximum neighbourhood index $s_{\max}$ and the seed of the pseudorandom number generator;
- for the *TS* metaheuristic, option `-ts`, followed by three parameters: the total number of neighbourhood explorations $t_{\max}$, the tabu tenure for the reinsertion of deleted elements L^{in}, and the tabu tenure for the removal of added elements L^{out}.

```
parse_command_line(argc,argv,data_file,exchange_algo,&niter,&seed,
                   &s_min,&delta_s,&s_max,&l_in,&l_out);

load_data(data_file,&I);

create_solution(I.n,&x);
greedy_farthest(&I,&x);

start = clock();
if (strcmp(exchange_algo,"-vns") == 0)
  variable_neighborhood_search(&I,&x,"-fb",niter,s_min,delta_s,s_max,&seed);
```

```
else if (strcmp(exchange_algo,"-ts") == 0)
  tabu_search(&I,&x,"-fb",niter,l_in,l_out);
end = clock();
tempo = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6f ",tempo);
printf("%8d ",niter);
print_sorted_solution(&x,I.n);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

E.1 Variable neighbourhood search

The *Variable Neighbourhood Search* metaheuristic applies a basic steepest ascent heuristic, and restarts it every time this terminates in a locally optimal solution. The restart is performed with a *shaking* procedure that modifies the current best known solution generating it at random in a suitable neighbourhood, whose size is the main parameter of the method[1]. The scheme is the following:

Algorithm VariableNeighbourhoodSearch$(I, x^{(0)}, \ell, s_{\min}, \delta s, s_{\max})$
$x :=$ SteepestAscent$(x^{(0)})$; $x^* := x$;
$s := s_{\min}$;
For $l := 1$ *to* ℓ *do*
  $x' :=$ Shaking(x^*, s);
  $x' :=$ SteepestAscent(x');
  If $f(x') > f(x^*)$
    then $x^* := x'$; $s := s_{\min}$;
    else $s := s + \delta s$;
  If $s > s_{\max}$ *then* $s := s_{\min}$;
EndWhile;
Return $(x^*, f(x^*))$;

This scheme is general enough to not require any specific adaptation for the *MDP*, except for the replacement of steepest descent with steepest ascent, as usual because it is a maximisation problem. However, we will also modify the termination condition, replacing the number of restarts ℓ with the total number of neighbourhood explorations $t_{\max}$ in order to get a better control of the computational time (of course, we could directly fix the total computational time, and that would be even more precise). To this purpose, we exploit the information on the number of neighbourhood explorations performed that is already provided by the steepest ascent procedure, but we also need to terminate if before reaching a locally optimal solution whenever

[1] This parameter is generally denoted as k (as in the slides of the theoretical lessons), but here we will denote it as s to distinguish it from the required number of points in the feasible solutions of the *MDP*.

the total number of available explorations has been consumed. For the sake of simplicity, we will adopt the steepest ascent heuristic discussed in the previous chapter, based on the single-swap neighbourhood N_{S_1} with the first-best exploration strategy. It is therefore straightforward to implement the `VNS` procedure as follows.

```
  truncated_steepest_ascent(pI,px,visit_strategy,niter,&iter);
  tot_iter = iter;

  create_solution(pI->n,&x_star);
  copy_solution (px,&x_star);

  s = s_min;
  while (tot_iter < niter)
  {
    shaking(pI,px,s,pseed);
    truncated_steepest_ascent(pI,px,visit_strategy,niter-tot_iter,&iter);
    tot_iter += iter;

    if (px->f > x_star.f)
    {
      copy_solution(px,&x_star);
      s = s_min;
    }
    else
    {
      s += delta_s;
      if (s > s_max) s = s_min;
    }
```

The variable `tot_iter` saves the cumulative number of neighbourhood explorations performed, in order to compare it with the maximum imposed value `niter`.

Moreover, the `truncated_steepest_ascent` procedure coincides with the already implemented `steepest_ascent` procedure, with the additional termination condition of stopping as soon as the remaining `niter-tot_iter` explorations have been performed. The modified procedure is already provided in a modified `localsearch.c` library.

```
*pniter = 0;
if (max_iter <= 0) return;
do
{
  explore_neighbourhood(px,pI,visit_strategy,&p_in,&p_out,&delta_f);
  if (delta_f > 0)
  {
    swap_points(p_in,p_out,px,pI);
    (*pniter)++;
  }

} while ( (delta_f > 0) && (*pniter < max_iter) );
```

The only part of the `variable_neighborhood_search` procedure that remains unimplemented is the `shaking` procedure, that receives the reference solution (that is the best known one, as we are applying the basic version of the *VNS*) the current value of parameter s and the seed of the pseudorandom number generator, and returns the perturbed solution that will be used as a starting point for the following application of the steepest ascent procedure. In order to implement it, we need to define a hierarchy of neighbourhoods from which to extract a random solution. In general, these neighbourhoods should be progressively increasing, in order to allow a controllable amount of intensification (using the first neighbourhoods) or diversification (using the last ones). In the specific case of the *MDP*, swaps are the most natural operation to generate neighbourhoods, due to the cardinality constraint, that guarantees the feasibility of every solution they generate, while proving the unfeasibility of the subsets generated by any other kind of operation. Therefore, we will adopt the hierarchy formed by the swap neighbourhoods N_{S_s}. Every solution in N_{S_s} is obtained performing s single swaps of a point in the current solution with a point out of it. A technical detail to be defined is whether we consider:

- s possibly overlapping swaps, in which a point deleted by one of the swaps can be added again by a following swap;
- s disjoint swaps, that is we swap s points from the starting solution with s points out of the starting solution.

The former definition allows to obtain also solutions that could be obtained with any number $s' \leq s$ of swaps (because swapping points i and j, followed by j and k is equivalent to swapping directly i and k). Therefore, it is a more general definition and it guarantees that each neighbourhood in the hierarchy include the previous ones. The latter definition restricts the neighbourhood only to disjoint swaps and yields disjoint neighbourhoods. However, it has the advantage that all neighbour solutions have a Hamming distance exactly equal to $2s$ from the reference solution x^*. This property seems particularly desirable from the point of view of controlling the size of the perturbation introduced (in the former case, a perturbation with a very large s might generate a solution very close to x^*, possibly even coincident with it). We therefore adopt the second definition. Generating a random s-swap is then straightforward: it requires to save in a vector the elements of the solution and extract s uniformly without repetitions. The same must be done for the elements out of the solution. Then, we can match the two sets of points in s pairs and swap the points to obtain the perturbed solution. There are several ways to do that, whose time and space complexity can be discussed in detail. Choosing one arbitrarily, we will write in a single vector of n elements the indices of all points: those in the solution in its first positions and the other ones in the last positions (using the incidence vector to distinguish them). Then, we will randomly extract s indices from the first and s from the second subvector, moving them to the ends of the vector to avoid generating duplicate terms and finally swap the corresponding points with the `swap_points` procedure. It can be argued that an *ad hoc* procedure could be more efficient (in particular for large values of s), but the shaking procedure is applied rarely enough to assume that it will take a negligible part of the overall computational time of the algorithm[2]

[2]This assumption should be verified, of course.

```
/* Build a vector with the indices of the internal points
   in the first k positions and the indices of the external
   point in the last n-k positions */
Indices = int_alloc(pI->n+1);
i_in = 1; i_out = pI->n;
for (i = 1; i <= pI->n; i++)
  if (px->in_x[i] == true)
    Indices[i_in++] = i;
  else
    Indices[i_out--] = i;

/* Select s internal points and move their indices to the
   first s positions of Indices */
for (i_in = 1; i_in <= s; i_in++)
{
  i = rand_int(i_in,pI->k,pseed);
  temp = Indices[i_in];
  Indices[i_in] = Indices[i];
  Indices[i] = temp;
}

/* Select s external points and move their indices to the
   last s positions of Indices */
for (i_out = pI->n; i_out >= pI->n-s+1; i_out--)
{
  i = rand_int(pI->k+1,i_out,pseed);
  temp = Indices[i_out];
  Indices[i_out] = Indices[i];
  Indices[i] = temp;
}

/* Perform s exchanges between the first and the last points of Indices */
for (i_in = 1, i_out = pI->n; i_in <= s; i_in++, i_out--)
    swap_points(get_point(Indices[i_in],pI),
                get_point(Indices[i_out],pI),px,pI);

free(Indices);
```

E.1.1 Time complexity estimation

The computational complexity of the *VNS* metaheuristic combines that of the steepest ascent heuristic with that of the shaking procedure. The former has already been estimated as $O(t_{\max}(n-k)k)$, with the modification that $t_{\max}$ is now the total number of neighbourhood explorations, fixed by the user. Applying the first-best exploration strategy means that the number of explored solutions should be $\leq (n-k)k$, though usually smaller (probably much smaller in the first iterations, when it is easier to improve the current solution, and nearly equal later when approaching the local optimum). The latter term is given by the initialisation in $O(n)$ time of the index vector, followed by the generation of s pseudorandom number (constant time, though

probably not a very small constant) and s swaps, that take (as discussed in the previous chapter) $O(n)$ time each. Overall, this is $O(\ell ns)$ time, where $s \leq \min(k, n-k)$ (as discussed in the following) and ℓ is the number of restarts (unknown *a priori*). However, s and ℓ tend to be inversely correlated, since smaller perturbations probably imply shorter paths to the local optimum and more restarts, whereas larger perturbations imply longer paths to the local optimum and less restarts. A rough guess is that each shaking application should take a time comparable to a neighbourhood exploration, so that overall its contribution should be negligible with respect to the steepest ascent heuristic, but this needs to be verified experimentally.

E.1.2 Empirical evaluation

We can now evaluate the performance of the *VNS* metaheuristic. We have decided to use the total number of neighbourhood explorations as a termination condition. This value should be set so as to allow the heuristic to experiment with every possible working condition. However, we also want to get results in a reasonably short time. In the previous chapter, we have estimated that the average number of iterations performed from a starting solution to the corresponding local optimum ranges between 15 and 30 in our benchmark instances. It will probably be smaller for starting solutions generated with small perturations from the best known solution. Since $s \leq k$, it should take at most $30 \cdot 400$ neighbourhood explorations to reach the strongest perturbations on the largest instances. Therefore, $t_{\max} = 10\,000$ could be a good choice, but we will adopt $t_{\max} = 1\,000$ to fit the experiments in the space of a lesson.

In the first experiments, we will let parameter s autotune, by fixing $s_{\min}$, δs and $s_{\max}$ to their simplest values: $s_{\min} = 1$, $\delta s = 1$ and $s_{\max} = \min(k, n-k)$.

E.1.2.1 Computational time analysis

The detailed results show that the overall computational time (including the initialisation constructive procedure, the shaking procedure and the steepest ascent procedure) ranges from fractions of a second to less than four seconds. This is even shorter than the exchange heuristic initialised by the try-all heuristic, which is a quite promising fact, should the results prove at least as good. Let us remind that the steepest ascent procedure applied to the farthest-point procedure could not enhance it enough to overcome the advantage provided by the better try-all initialisation procedure, even if its computational time was smaller. We can investigate whether the use of *VNS* allows exploit the shorter computational time to get better results.

Figure E.1 reports the logarithmic *scaling diagram* for the computational time of the *VNS* heuristic on the whole benchmark. The diagram shows a very regular polynomial increase with size, also thanks to the fixed number of iterations $t_{\max}$. The $O((n-k)k)$ theoretical estimate, with $k \propto n$, suggests a quadratic complexity, which is in extremely good accordance with the linear interpolation:

$$T_A = \beta n^{\alpha} \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

since $\alpha \approx 2.008$ and $\beta \approx 2.6 \cdot 10^{-6}$.

E.1.2.2 Solution quality analysis

Figure E.2 reports the *SQD* diagram of the *VNS* metaheuristic, compared with that of the steepest ascent heuristic applied to the result of the farthest-point and of the try-all heuristics: the former allows the comparison with the result obtained stopping at the first local optimum; the

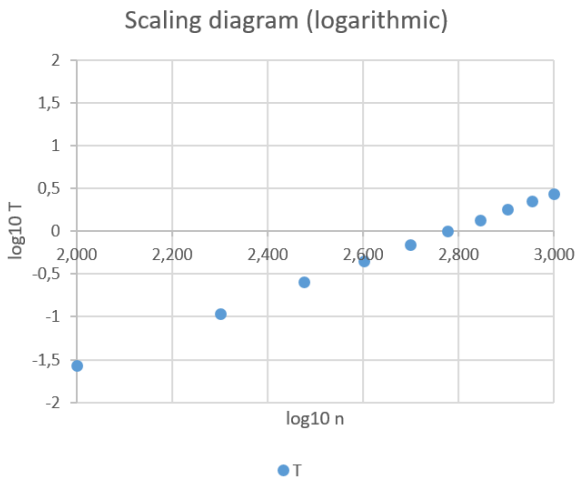


Figure E.1: Scaling diagram in logarithmic scales for the *VNS* algorithm on the benchmark

latter allows a comparison with a heuristic taking a similar computational time. The improvement with respect to the former is clear (the average gap decreases from 0.93% to 0.33%): it is a strict dominance, given that they visit the same solutions, and the *VNS* proceeds when the competitor just terminates. Also with respect to the try-all initialisation the average gap is better (0.33% versus 0.35%) and the *SQD* diagram shows a much larger fraction of very good results, but also a larger fraction of bad results. Notice that the computational time is not the same: a true comparison should be made in perfectly equal conditions.

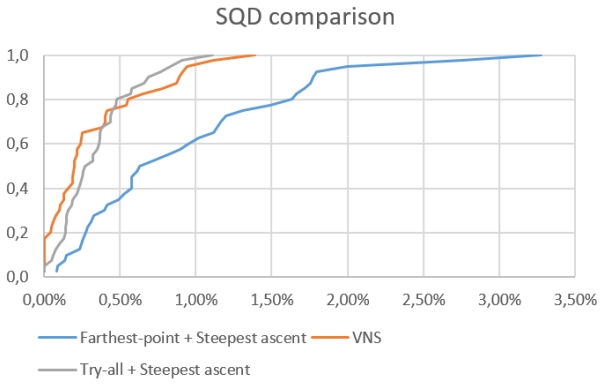


Figure E.2: Solution Quality Distribution diagram for the *VNS* metaheuristic compared with the *steepest ascent* heuristic initialised with the *farthest point* and the *try-all* procedures

The boxplots reported in Figure E.3 provide the same information: *VNS* provides both better and worse results with respect to the *steepest ascent* initialised by the try-all heuristic.

In order to have a rough idea of the corresponding computational times, Figure E.4 provides the *RTD* diagram, from which it is apparent that *VNS* has some margin of further improvement for the slower runs, that correspond to the largest instances. On the contrary, it is already slower for the faster runs, that is the smallest instances. Therefore an equal-time comparison is clearly necessary to be fair. Performing one, however, is complicated when some of the competing algorithms have an intrinsic termination condition: we should build a time-limited version of the *VNS* heuristic, run the steepest ascent heuristic saving its computational time, and feed it to

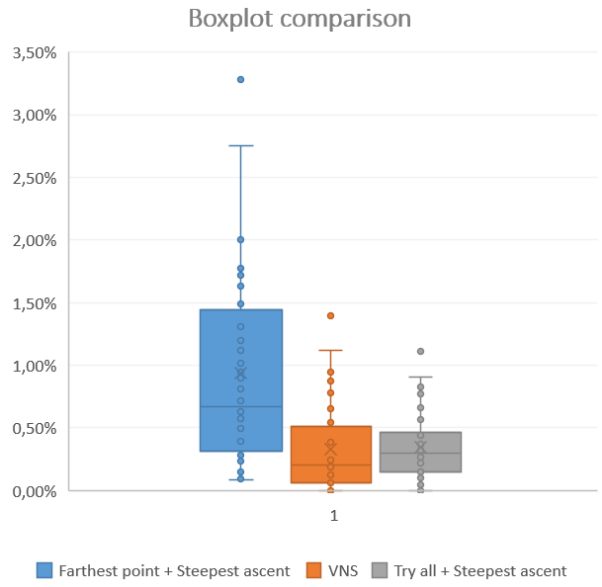


Figure E.3: Boxplots for the *VNS* metaheuristic compared with the *steepest ascent* heuristic initialised with the *farthest point* and the *try-all* procedures

the *VNS* heuristic. This is beyond the scope of the course.

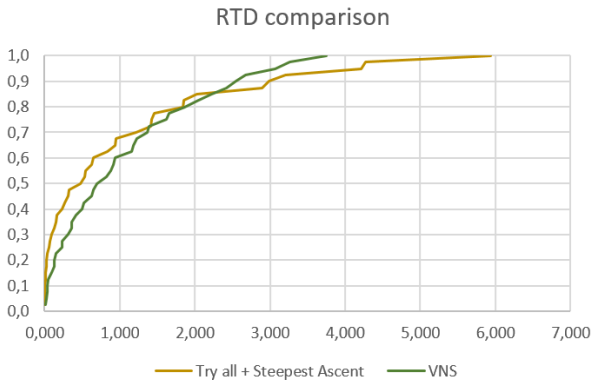


Figure E.4: Solution Quality Distribution diagrams for the *VNS* metaheuristic and the *steepest ascent* heuristic initialised with the *try-all* procedure

E.1.3 Parameter tuning

It is often possible to improve the performance of an algorithm by tuning the values of its parameters. In the case of the *VNS*, this corresponds to increasing the smallest neighbourhood used to restart the search, or decreasing the largest one, or skipping some intermediate neighbourhoods. The last possibility is useful when the number of neighbourhoods is huge. In this case, s ranges from 1 to 400, that is quite large. As we have fixed a rather small total number of neighbourhood explorations $t_{\max}$, it is possible (and it can be easily verified printing the value of s at each shaking operation) that on the larger benchmark instances only small shakings are performed. In order to test this aspect, we compare some alternative parameter configurations.

The test is very limited, considering the following configurations, chosen so as to cover rather extreme cases:

- $s_{\min} = 1$ and $s_{\max} = k$: the trivial configuration (probably not fully exploited, for the insufficient number of iterations);
- $s_{\min} = 1$ and $s_{\max} = 10$: an intensifying configuration, imposing rather small perturbations;
- $s_{\min} = 1$ and $s_{\max} = k/2$: a configuration avoiding only very large perturbations (this could coincide with the trivial fone if the insufficient number of neighbourhood explorations forbids to increase s beyond the upper bound);
- $s_{\min} = k/2$ and $s_{\max} = k$: a configuration producing strong perturbations since the beginning to diversify the search and try to avoid falling back in the reference solution;
- $s_{\min} = k$ and $s_{\max} = k$: a degenerate configuration in which the maximum possible perturbation is applied (all points must in the best known result are replaced by new random points);

The number of configurations and the number of iterations for each configuration are too limited to provide really significant results, but they are a good occasion for some intuitions and further discussion. A good way to formulate reasonable configurations is to print the values of s that improve the best known result. Increasing $s_{\min}$ and decreasing $s_{\max}$ to approach such values will modify the overall behaviour of the algorithm, because the same sequence of pseudorandom numbers will imply completely different choices, but it is probably a good idea: if no improvement is found out of a certain range of values of s, this probably means that the radius of the basins of attraction falls in a similar range, and therefore a good perturbation should be in the same range.

A quick glance at the computational times suggests that they are nearly independent from the parameters: the variations are usually below 10%, that in a period of few seconds is probably just due to random fluctuations. The quality of the results is roughly indicated by Table E.1 which reports the average gaps with respect to the best known result. The diversifying configuration ($[k/2,k]$) is the best one, followed by the relaxed configuration ($[1,k]$) and the slightly intensifying one ($[1,k/2]$), that are very similar, as expected. The strongly intensifying and the strongly diversifying configuration appear to be the worst. It can be interesting to notice that the strongly diversifying configuration ($[k,k]$) actually proved the best in one of my personal contributions to the literature, which combined rather short runs of a *TS* metaheuristic with a *VNS* restart mechanism. This is not unreasonable, given that the *TS* procedure probably guarantees a good exploration of the region surrounding the current best known result (better than the steepest ascent procedure adopted here), so that stronger perturbations in the restart make sense.

Figures E.5 and E.6 show the *SQD* and boxplot diagrams of the five configurations, that clearly confirm the dominances suggested by the average gaps. Indeed, checking the results of the steepest ascent heuristic initialised with the try-all procedure, the diversifying configuration seems to be comparable even in the upper part of the diagram.

E.1.3.1 Statistical tests

We can also compare the performance of the five configurations with statistical tests. Using Wilcoxon's test poses a methodological problem: the test is designed to compare two empirical

$s_{\min}$	$s_{\max}$	Average gap
1	k	0.33%
1	10	0.45%
1	$k/2$	0.34%
$k/2$	k	0.22%
k	k	0.44%

Table E.1: Average gaps with respect to the best known result of the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

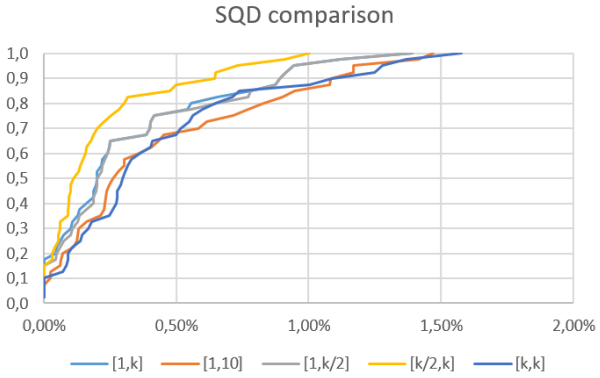


Figure E.5: Solution Quality Distribution diagrams for the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

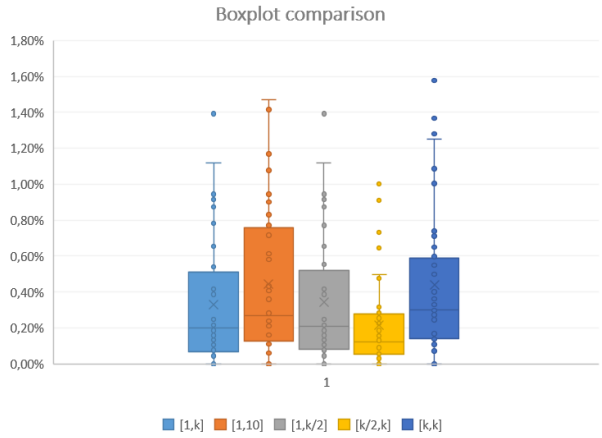


Figure E.6: Boxplot diagrams for the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

populations. It is possible to use it on more than two algorithms, applying it to all pairs or choosing a reference heuristic and comparing the other ones to it. This, however, requires to handle with care the interpretation of the results. The p-value obtained, in fact, estimates the probability of observing the empirical results under the assumption that the two samples compared derive from the same population, that is, that the two configurations compared are equally effective. When p is small, this interpretation of the results can be rather safely rejected, but it must not be considered as straightforwardly false. If the test is applied n_t times, the probability that at least one of the interpretations drawn from the test is false becomes larger and larger. This means that our conclusions should be based on stricter requirements. The literature offer several methods to correct the estimates provided by a pairwise test. The simplest one is the *Bonferroni correction*, that is based on Boole's inequality for the *familywise error rate* (*FWER*):

$$FWER = P\left[\bigcup_{i=1}^{n_t}(p_i \leq \alpha)\right] \leq \sum_{i=1}^{n_t} P\left[(p_i \leq \alpha)\right]$$

In other words, the sum of the p-values obtained gives an overestimate of the p-value for the overall observation. A simple way to impose it every given threshold on the significancy of the results (for example, the classical 5%) consists in dividing such a threshold by the number n_t of the tests.

For example, let us check the hypothesis that the diversifying configuration is better than the other four. This corresponds to the four following different pairwise tests:

1. $[k/2,k]$ versus $[1,k]$

```
W+ = 470, W- = 91, N = 33, p <= 0.0007329
```

2. $[k/2,k]$ versus $[1,10]$

```
W+ = 630, W- = 73, N = 37, p <= 2.743e-005
```

3. $[k/2,k]$ versus $[1,k/2]$

```
W+ = 499, W- = 96, N = 34, p <= 0.0005896
```

4. $[k/2,k]$ versus $[k,k]$

```
W+ = 612, W- = 18, N = 35, p <= 1.197e-006
```

The hypothesis is consistent with the observations even after applying Bonferroni's correction, since the sum of all p-values is 0.2% and each one is smaller than $5\%/n_t = 1.25\%$ (even a threshold much tighter than 5% would be respected).

Of course, other systems of assumptions, such as a full ordering among the configurations, could be checked, but they would probably be less interesting: we are mainly interested in finding the best performing configuration on the given benchmark.

E.2 Tabu search

The *Tabu Search* metaheuristic applies a basic steepest ascent heuristic, but modifies it introducing a limitation of the neighbourhood that forbids to accept as incumbent a solution already visited, but also (in the standard attribute-based version) a solution similar to the ones already visited. In order to adapt the *TS* metaheuristic to the *MDP*, we will initialise the search with the farthest-point heuristic and adopt the basic single-swap neighbourhood N_{S_1} as done for the *VNS*. This should make the comparison between the two approaches fairer, allowing to understand whether the problem is better attacked by restarting or prolonging the search after hitting a locally optimal solution. The general *TS* scheme can be easily adapted to the *MDP* with the usual replacements due to the maximising nature of the problem.

Algorithm TabuSearch$\left(I, x^{(0)}, t_{\max}, L\right)$
$x := x^{(0)}$; $x^* := x^{(0)}$;
$\bar{A} := \emptyset$;
For $t := 1$ *to* $t_{\max}$ *do*
  $x' := \emptyset$;
  For each $y \in N(x)$ *do*
    If $f(y) > f(x')$ *then*
      If Tabu$\left(y, \bar{A}\right)$ = *false or* $f(y) > f(x^*)$ *then* $x' := y$;
    EndIf
  EndFor
  $x := x'$;
  $\bar{A} :=$ Update$\left(\bar{A}, x', L\right)$;
  If $f(x') > f(x^*)$ *then* $x^* := x'$;
EndFor
Return $(x^*, f(x^*))$;

First, we must decide whether to apply the basic version of *TS* or the most common attribute-based one. In the *MDP* literature, the best performing algorithm proposed so far to solve the problem actually uses the basic version. However, this algorithm introduces several refinements, and we are interested in the *MDP* mainly as an example for the application of *TS* to general Combinatorial Optimization problems. We will therefore implement an attribute-based *TS* metaheuristic. It is rather natural to define a pair of complementary attribute sets, that are the presence and the absence of given points in the current solution. Setting $A = x$ and $A' = P \setminus x$ means that every time a point is deleted from the solution, it becomes tabu for a given number L^{in} of neighbourhood explorations to add it back; conversely, every time a point is added to the solution, it becomes tabu for a given number L^{out} of neighbourhood explorations to delete it.

Procedure *tabu_search* implements both tabu lists on a single integer vector `T`, each of whose components reports the last iteration at which the corresponding point has changed its status: if i currently belongs to x, `T[i]` is the iteration at which i has entered the solution; if i currently does not belong to x, `T[i]` is the iteration at which i has gone out of the solution. At the beginning, `T[i]` is set to `-MAX_INT` for all points, so that the check on the tabu status of any point states that the point is not tabu, and therefore free for exchanges. Procedure `explore_neighbourhood_with_tabu` differs from the `explore_neighbourhood` proce-

dure implemented in the previous chapter for the steepest ascent heuristic in that it takes into account also the tabu status (in fact, it requires the current iteration index `iter`, vector `T`, the tabu tenures `Lin` and `Lout`, and the value of the best known solution `w_star.f`, to apply the aspiration criterium). The update of the tabu list simply amounts to saving the current iteration index in the two positions of vector `T` associated to the points swapped. We should also take into account the possibility that all moves in the neighbourhood are tabu. In that case, the neighbourhood exploration procedure should return the solution with the oldest tabu status. The current implementation simply returns no point and performs no move, remaining idle until some tabu expires. This is an inefficient implementation, that shall be corrected in future versions of the algorithm.

```
create_solution(pI->n,&x_star);
copy_solution(px,&x_star);

T = int_alloc(pI->n+1);
for (i = 1; i <= pI->n; i++)
  T[i] = INT_MIN;

for (iter = 1; iter <= max_iter; iter++)
{
  explore_neighbourhood_with_tabu(px,pI,visit_strategy,
                                  iter,T,l_in,l_out,x_star.f,
                                  &p_in,&p_out,&delta_f);
  if (p_in != NO_POINT)
  {
    swap_points(p_in,p_out,px,pI);
    T[get_index(p_in,pI)] = T[get_index(p_out,pI)] = iter;
    if (px->f > x_star.f) copy_solution(px,&x_star);
  }
}

free(T);

copy_solution(&x_star,px);
```

The exploration of the neighbourhood is performed exactly as in the steepest ascent heuristic, with the addition of a further check. If the currently explored swap is tabu (and this is checked by function `is_tabu`, then the move is performed only if it improves upon the best known one, that is if the improvement δf applied to the current solution value yields a value strictly better than the best known one (*aspiration criterium*). In this special case, in fact, the new solution is only apparently violating a tabu, and is on the contrary providing a precious overall improvement to the search process. We still apply the first-best exploration strategy, returning the first improving solution in the neighbourhood, but only if it is nontabu, or it satisfies the aspiration criterium.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px);
```

```
      p_in = next_point(p_in,px))
  for (p_out = first_point_out(px); !end_point_list(p_out,px);
       p_out = next_point(p_out,px))
  {
    delta_f = evaluate_exchange(p_in,p_out,px,pI);
    if ( (delta_f > *pdelta_f) &&
         (!is_tabu(p_in,p_out,pI,iter,T,l_in,l_out,px->f+delta_f,fstar)) )
    {
      *pdelta_f = delta_f;
      *pp_in = p_in;
      *pp_out = p_out;
      if ( (delta_f > 0) && (strcmp(visit_strategy,"-fb") == 0) ) break;
    }
  }
```

Finally, the check of the tabu status simply consists in determining whether the current iteration index has reached or not the value at which the tabu expires. This must be checked both for the point i that is leaving the solution (and in that case the tabu tenure is `l_out`) and for the point j that is entering the solution (in that case the tabu tenure is `l_in`) .

```
if (f > f_star) /* aspiration criterium */
  return false;
else
  return ((iter <= T[get_index(p_in,pI)] + l_out) ||
          (iter <= T[get_index(p_out,pI)] + l_in));
```

E.2.1 Time complexity estimation

It is rather obvious that the computational complexity of the *TS* metaheuristic coincides with that of the basic steepest ascent heuristic, as all additional operations require constant time in their respective locations:

- the evaluation of the tabu status adds a constant number of operations to the evaluation of the cost of each explored solution;
- the update of the tabu list adds a constant number of operations to the execution of the move, that is the exploration of a neighbourhood;

The allocation, initialization and deallocation of vector `T` add $O(n)$ time to the overall algorithm. Therefore, the overall complexity remains $O(t_{\max}(n-k)k)$, where the number of neighbourhood explorations $t_{\max}$ is a parameter provided by the user as the termination condition.

E.2.2 Empirical evaluation

We can now evaluate the performance of the *TS* metaheuristic. We set the total number of neighbourhood explorations to $t_{\max} = 1\,000$, as for the *VNS* metaheuristic, in order to allow a meaningful comparison between them, even if a completely fair comparison would require to give them the same computational time.

E.2.2.1 Cyclic or erratic behaviours

Contrary to the *VNS*, where it is usually very easy to determine a default configuration for the parameters $s_{\min}$, δs and $s_{\max}$, the most influential parameters of the *TS*, that is the *tabu tenures* need to be tuned with a certain care since the beginning. Two basic complementary risks must be avoided:

- *cyclic behaviours*: if the tabu tenure is too short, the search can get stuck in a cyclic sequence of solutions, because the search is attracted by locally optimal solutions that have already been visited and the tabu expires before the cycle starts again;
- *erratic behaviours* or empty neighbourhoods: if the tabu tenure is too long, the search can wander in the solution space avoiding the more promising regions because these are too close to solutions that have already been visited; the neighbourhood can even become fully tabu.

Since the number of points out of the solution varies from $n-k=60$ to $n-k=900$, and the number of points in the solution varies from $k=10$ to $k=400$, the tenures should keep below these values, but above a few units, which is still a rather large range. The tenures could also possibly depend on the size of the instance. It is also likely that the tenure for reinsertion L^{in} should be larger than the tenure for redeletions L^{out}, because the candidate elements for insertion (the $n-k$ external ones) are more numerous than the candidate elements for deletion (the k internal ones). A simple way to verify the occurrence of the cyclic behaviours and (less evidently) of erratic ones, is to plot the profile of the objective function along the search.

In order to give a quick idea of how reasonable values are introduced, we will fix focus on the smallest instance (`n0100k010.txt`), and investigate different possible values for L^{in} while trivially setting $L^{\text{out}}=1$, so that the tabu mechanism is mainly based on the reinsertion of deleted elements. Figure E.7 reports the profile of the objective function for the first 100 neighbourhood explorations. The configuration with $L^{\text{in}}=5$ exhibits a clearly periodic profile, suggesting that the tenure is too short. The one with $L^{\text{in}}=6$ appears much better, but long cycles (with a period of 106 iterations actually arise after a while). The configuration with $L^{\text{in}}=8$ hits the best known result ($f^*=3\,561$) several times during the whole run. The configuration with $L^{\text{in}}=30$ starts moving erratically in regions of worse quality. Finally, the configuration with $L^{\text{in}}=100$ seems to avoid good solutions and also shows sequences of iterations in which the value of the objective does not change because the whole neighbourhood is tabu ($L^{\text{in}}=100>n-k=90$) and our simple implementation trivially waits for the tabu to expire.

Figure E.8 reports the profile of the objective function for the whole run on the largest instance (`n1000k400.txt`). The configurations with the smallest values of L^{in} actually prove better, without exhibiting cyclic behaviours. This (absolutely nonobvious) behaviour is possibly due to the much larger size of the neighbourhood, that allows the algorithm to choose the incumbent in a larger set, and thus reduces the risk of repeating the same sequence of moves.

E.2.2.2 Computational time analysis

The overall computational time, including the initialisation procedure is very similar to that of the *VNS* metaheuristic: it ranges from fractions of a second to about four seconds. It is therefore another interesting candidate to provide an efficient solving approach. Figure E.9 reports the logarithmic *scaling diagram* on the whole benchmark. The diagram shows a polynomial

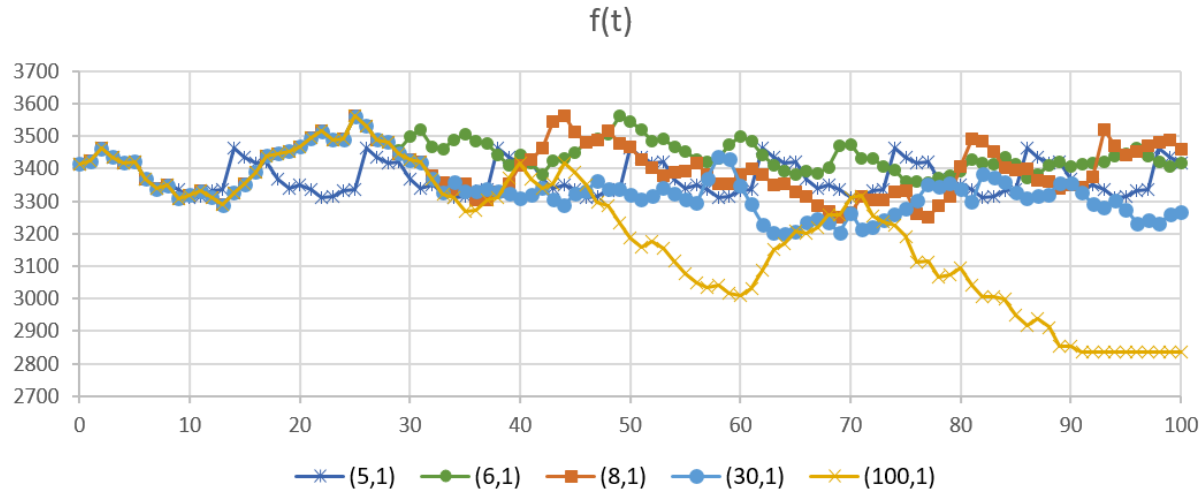


Figure E.7: Profile of the objective function for the *TS* metaheuristic on instance `n0100k010.txt` with different values of tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

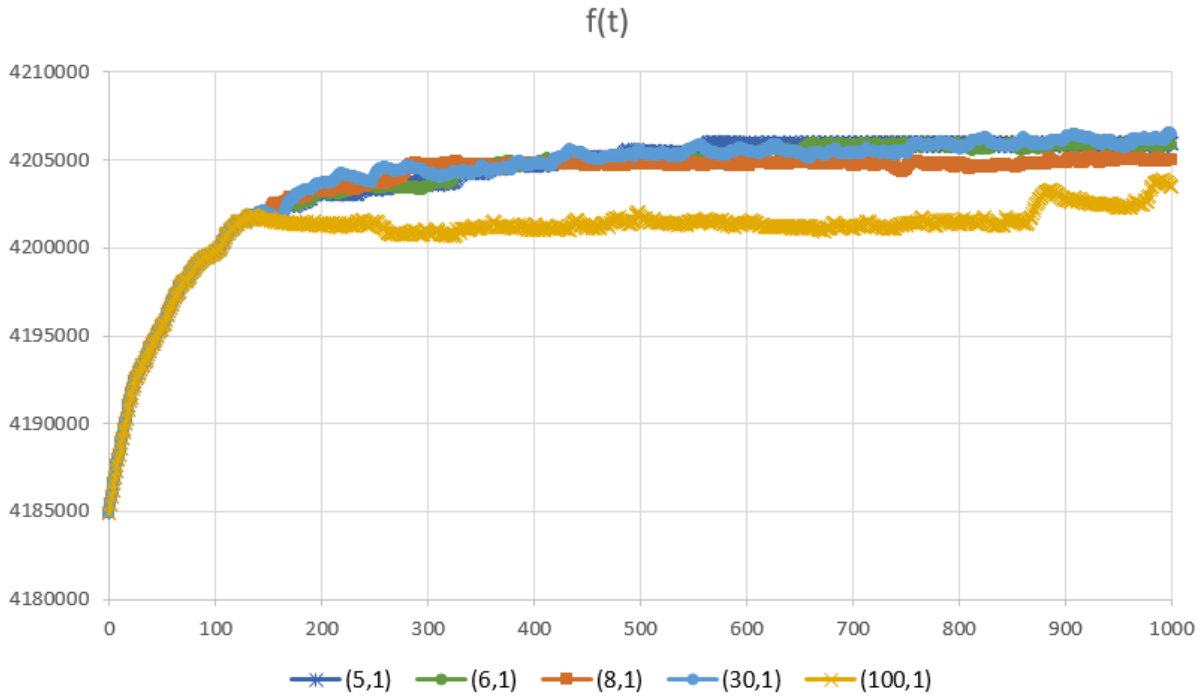


Figure E.8: Profile of the objective function for the *TS* metaheuristic on instance `n1000k400.txt` with different values of tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

increase with size, in good accordance with the $O((n-k)k)$ theoretical estimate, which corresponds to a quadratic complexity when $k \propto n$. In fact, the linear interpolation is:

$$T_A = \beta n^{\alpha} \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

since $\alpha \approx 2.012$ and $\beta \approx 2.7 \cdot 10^{-6}$.

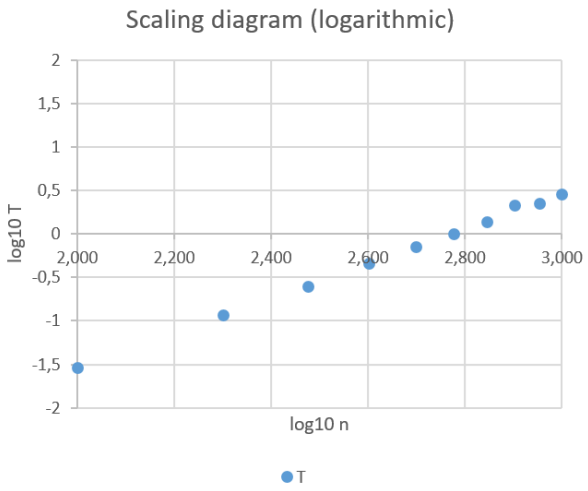


Figure E.9: Scaling diagram in logarithmic scales for the *TS* algorithm on the benchmark

E.2.3 Parameter tuning

We now compare a small number of configurations in which both tenures L^{in} and L^{out} assume different values. For the sake of simplicity, we apply the same values to all instances of the benchmark, also based on the results of the previous experiments, which showed that rather short tenures are enough to avoid cyclic behaviours, and at the same time to allow the search to focus on good quality solutions, both on the smallest and the largest instance. We consider the six configurations obtained setting $L^{\text{in}} = 5$, 6 or 8 and $L^{\text{out}} = 1$ or 2. This is, of course, only a very simple illustrative investigation. Table **??** reports the average gaps over the whole benchmark. They are quite promising: most of them are smaller than the corresponding values obtained by the *VNS* metaheuristic; the larger tenures, in particular, provide the best gap.

L^{in}	L^{out}	Average gap
5	1	0.31%
6	1	0.26%
8	1	0.19%
5	2	0.17%
6	2	0.18%
8	2	0.14%

Table E.2: Average gaps with respect to the best known result of the *TS* metaheuristic with different tunings of the tabu tenures L^{in} and L^{out}

Figure E.10 reports the *SQD* diagram of the six configurations, confirming the better performance of the configuration with the larger tenures.

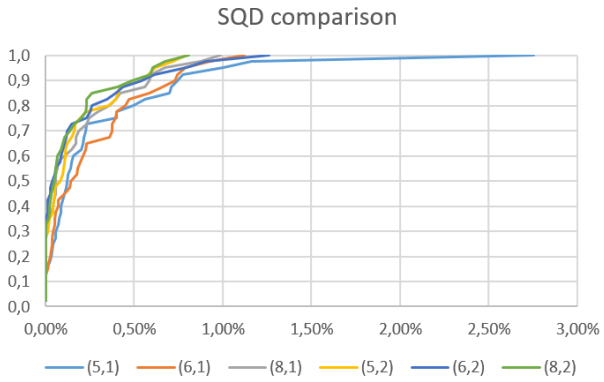


Figure E.10: Solution Quality Distribution diagrams for the *TS* metaheuristic with different tunings of the tabu tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

The corresponding boxplots are reported in Figure E.11: they seem to contradict the *SQD* diagram ($L^{\text{in}} = 1$ and $L^{\text{out}} = 8$ look better than $L^{\text{in}} = 2$ and $L^{\text{out}} = 8$), but that mainly depends on the automatic definition of outliers by Excel[3] It seems anyway justified to consider larger tenures as better. Indeed, the experiments should now include larger values, to determine whether it is possible to further improve the results, but we stop here.

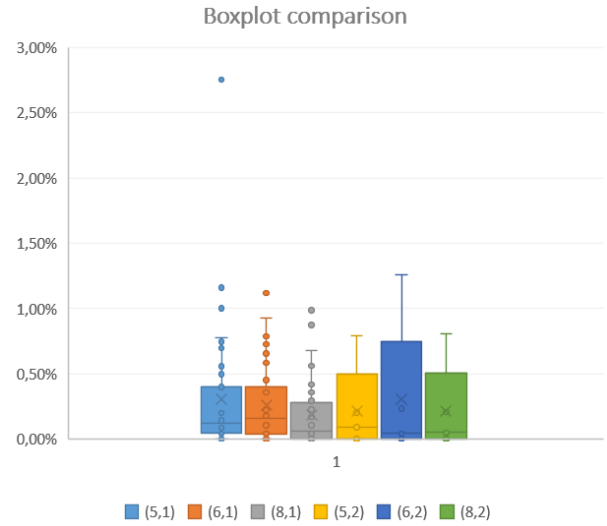


Figure E.11: Boxplot diagrams for the *TS* metaheuristic with different tunings of the tabu tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

The application of Wilcoxon's test to compare $\left(L^{\text{in}}, L^{\text{out}}\right)$ with the other configurations gives the following results:

1. $(8,2)$ versus $(5,1)$

```
W+ = 432, W- = 64, N = 31, p <= 0.0003233
```

[3]There seems to be a formula to label a value as an outlier. I do not know whether this is a statistic standard or just an Excel convention.

2. $(8,2)$ versus $(6,1)$

```
W+ = 494, W- = 101, N = 34, p <= 0.0008056
```

3. $(8,2)$ versus $(8,1)$

```
W+ = 261.50, W- = 173.50, N = 29, p <= 0.3469
```

4. $(8,2)$ versus $(5,2)$

```
W+ = 313, W- = 152, N = 30, p <= 0.09987
```

5. $(8,2)$ versus $(6,2)$

```
W+ = 248, W- = 187, N = 29, p <= 0.5165
```

which suggests that only the first two comparisons, with configurations having both tenures shorter, are significant (even applying the Bonferroni correction), whereas the other comparisons could easily be the result of a random sampling.

E.3 Comparison between *VNS* and *TS*

To conclude our experiments, we compare the *VNS* and the *TS* metaheuristic, with their best performing configurations, both with respect to the quality of the solutions and the computational time. The corresponding *SQD* and *RTD* diagrams are reported in Figures E.12 and E.13, and show that the *TS* metaheuristic is more effective while taking the same time as the *VNS* algorithm. The result is confirmed by the following response of Wilcoxon's test:

```
W+ = 131, W- = 464, N = 34, p <= 0.00454
```

It is important not to overestimate the range of these conclusions. They refer to a not very large benchmark of a specific nature, to a computation consisting of $t_{\max} = 1\,000$ iterations, to a pair of configurations that have been obtained with a very short investigation. They are however a preliminary result of a certain soundness, obtained with an experimental methodology based on the formulation and verification of hypotheses.

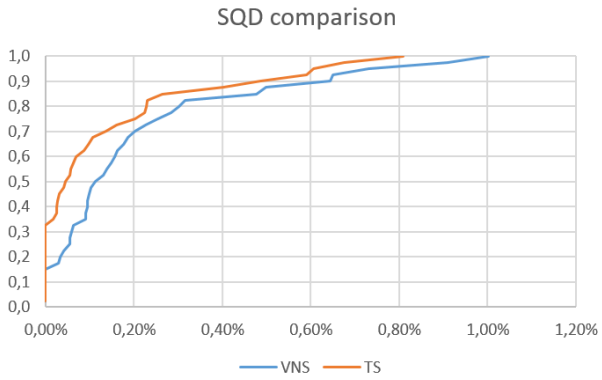


Figure E.12: Solution Quality Distribution diagrams for the *VNS* metaheuristic and the *TS* metaheuristic

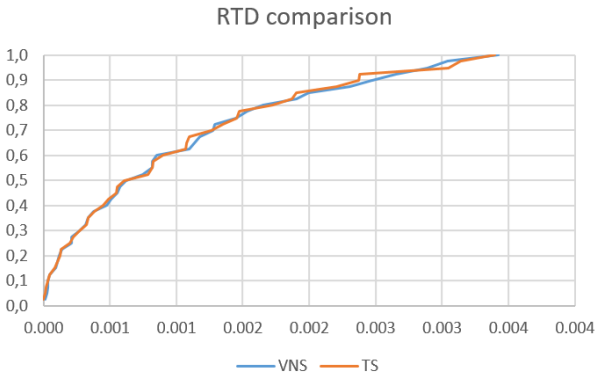


Figure E.13: Run Time Distribution diagrams for the *VNS* metaheuristic and the *TS* metaheuristic

APPENDIX F

Laboratory on recombination metaheuristics

In this chapter we consider a couple of recombination metaheuristics for the *Maximum Diversity Problem* (*MDP*), namely a *Path Relinking* (*PR*) and a *Scatter Search* (*SS*) approach. Recombination-based approaches do not have a common basic scheme, and they very frequently use randomization or memory. Therefore, the distinction between heuristics and metaheuristics is much less easy to draw than for constructive and exchange approaches. In general, however, they manipulate a reference set of solutions extracting from it suitable subsets and combining the elements of such subsets to generate new solutions. The aim is to retrieve promising portions from different solutions and integrate them in a better way, instead of labouriously modifying the bad portions of a single solution while keeping the good ones.

Since *PR* and *SS* algorithms are mainly deterministic methods based on the idea of exploiting as much as possible the information provided by the data, the initial population usually consists of locally optimal solutions generated by a previous exchange heuristic or metaheuristic. In order to make both methods virtually unlimited, we need a mechanism to generate new solutions even when the recombination mechanism fails to do so. For the sake of simplicity, we generate random solutions and improve each of them to local optimality, as we have done in Chapter 4, to restart the steepest ascent heuristic. In order to understand whether the performance of the two approaches depends on the recombination mechanism or not, we will compare them with the simple steepest ascent heuristic with random restart without applying any recombination procedure. In order to allow a fair comparison among these very different approaches, we will impose a limit on the total computational time as a termination condition.

The neighbourhood used in all three approaches will be the usual single-swap neighbourhood N_{S_1}, explored with the first-best strategy. We will discuss later in detail how the mechanism to generate new solutions is integrated in the schemes of *PR* and *SS*. These will be, therefore, more complicated than the ones reported in the theoretical lessons, which assumed the approaches to terminate as soon as the recombination operations fail to update the reference set. We will consider first *PR* and then *SS*, contrary to the theoretical lessons, because the former is simpler and this will allow to introduce more gradually different concepts and algorithmic components.

The command line arguments of the `main` procedure allow the user to choose which of the three metaheuristics to apply and the associated parameters:

- for the *PR* metaheuristic, option `-pr`, followed by three parameters: the total time $\tau_{\max}$, the cardinality of the reference set $|R|$, and the seed of the pseudorandom number generator;

- for the *SS* metaheuristic, option `-ss`, followed by four parameters: the total time $\tau_{\max}$, the cardinality of the best set $|B|$ and of the diverse set $|D|$, and the seed of the pseudorandom number generator;
- for the *random restart* metaheuristic, option `-rr`, followed by two parameters: the total time $\tau_{\max}$ and the seed of the pseudorandom number generator.

The structure of the `main` function is therefore the usual one, with the exception that no starting solution is generated, because the population is initialized inside the two recombination procedures.

```
parse_command_line(argc,argv,data_file,exchange_algo,&time,&seed,&nb,&nd);

load_data(data_file,&I);

create_solution(I.n,&x);

start = clock();
if (strcmp(exchange_algo,"-ss") == 0)
  scatter_search(&I,&x,"-fb",niter,nb,nd,&seed);
else if (strcmp(exchange_algo,"-pr") == 0)
  path_relinking(&I,&x,"-fb",niter,nb,&seed);
else if (strcmp(algo,"-rr") == 0)
  random_restart(&I,&x,"-fb",tauMax,&seed);
end = clock();
tempo = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6f ",tempo);
printf("%8d ",niter);
print_sorted_solution(&x,I.n);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

F.1 Path relinking

The *Path Relinking* metaheuristic manages a *reference set*, that is composed of the best known solutions. We will denote it as B (instead of R) to stress the similarities with the design of the *Scatter Search* approach. The basic scheme of *PR* extracts pair of solutions from B and applies an auxiliary exchange procedure to move from the first to the second solution of each pair. Then, it identifies the best solution along each path thus determined and improves it with an exchange procedure. Usually, for the sake of simplicity, the exchange procedure used for both purposes is the same used to generate the solutions of the reference set. The only difference is that, when drawing the relinking path, the objective is to minimize the Hamming distance from the final solution, and only secondarily to optimize the objective of the problem. The locally optimal solution found for each pair is checked to decide whether it is worth inserting in the

reference set. The process terminates when all solutions generated in this way are rejected, and the reference set is not updated.

Therefore, this scheme has an intrinsic termination condition. In order to prolong it indefinitely, if the candidate solutions generated by the relinking paths are less than a given number, new ones are generated at random and improved by steepest ascent. The same process is applied (with no candidate obtained by recombination) to generate the starting reference set. Then, these candidate solutions are tested for insertion in the reference set. This is not the only possible approach: many others are used in the literature, but this one has the advantage of being simple. An important distinction is between the *static update*, which collects the new solutions in a pool and tests them for insertion in the reference set at the end of the recombination phase, and the *dynamic update*, which immediately tests the new solutions. The former approach has the advantage of a simpler implementation and of exploiting all the generated solutions, whereas the latter can remove a solution from the reference set before using it for recombination. On the other hand, the latter approach is more aggressive, as it immediately exploits the new solutions, possibly leading to good results earlier. We will adopt the static update for the sake of simplicity.

The resulting scheme is the following:

Algorithm PathRelinking$(I, \tau_{\max}, n_B)$
$B := \emptyset; P := \emptyset;$
While Time() $\leq \tau_{\max}$ *do*
  { Build or integrate the candidate population }
  While $|P| < n_B$ *do*
    $x :=$ RandomSolution(I);
    $x :=$ SteepestAscent(x);
    If $x \notin P$ *then* $P := P \cup \{x\}$;
  EndWhile
  { Test the candidate solutions for insertion in the reference set }
  For each $x \in P$ *do*
    If $x \notin B$ *then* $B :=$ UpdateBestSet(B, x);
  EndFor
  { Recombine the solutions in the reference set into candidate ones }
  $P := \emptyset$;
  For each $(x, y) \in B \times B$ *do*
    $\Gamma_{x,y} :=$ FindRelinkingPath(x, y, I);
    $z := \arg \max_{w \in \Gamma_{xy} \setminus \{x,y\}} f(w)$;
    $z :=$ SteepestAscent(z);
    If $z \notin P$ *then* $P := P \cup \{z\}$;
  EndFor
EndWhile
$x^* := \arg \max_{x \in B} f(x)$;
Return $(x^*, f(x^*))$;

Function *Time()* returns the time elapsed from the beginning of the algorithm, in order to enforce the termination condition. In C, this is done with function `clock()`, that returns the number of

time units[1] elapsed since the beginning of the execution. For the sake of simplicity, we will call this function only in the outer loop. If performing a single iteration is slow, due to the size and features of the instance, to the size of the reference set or any other factor, the time limit will be actually violated. This should be avoided checking the time also inside the inner loops. Of course, too frequent checks negatively affect the overall computational time.

The first inner loop builds a population of candidate solutions by generating random solutions and improving them with steepest ascent. For the sake of simplicity, the number of candidates is set large enough to fill the reference set (n_B), though in general it could be larger. Duplicate candidates are not accepted, because they would not provide any advantage in the following operations. There is therefore a risk that this loop could not terminate, if many identical locally optimal solutions are obtained. This is rather unlikely, however, with a sufficiently diversifying generation mechanism, such as a random extraction. Notice that in the following iterations of the outer loop, the random candidates will be used only to integrate the solutions generated by recombination when the latter are too few, and that the candidates derived from recombination could be more numerous than n_B, in which case this loop will simply be skipped. Additional parameters could be introduced to tune all these aspects.

The second inner loop tests each candidate solution to check whether it deserves to be included in the reference set B. This is done with a suitable procedure that, in the positive case, adds the solution to update the subset. At the end, the population is cleared, because all its useful elements have been added to the reference set.

The third inner loop considers each pair of solutions (x, y) from B, draws a relinking path from x to y and finds the best solution z on this path (excluding the two extremes). This is then improved by steepest ascent and saved in the population P. The exchange procedure will, as usual, exploit the single-swap neighbourhood, but it will use the global-best strategy when drawing the relinking path, and the first-best strategy when improving the solution. The reason is that the neighourhood used to move between candidate solutions is severely limited by the requirement to strictly reduce the Hamming distance from the final solution, so that a more careful exploration of the neighbourhood seems preferable. When all pairs of reference solutions have been considered, the algorithm starts a new iteration.

The procedures required by this phase of the implementation are in part available in the library `solutionpool`, that manages the allocation and deallocation of solution pools (`create_solutionpool` and `destroy_solutionpool`), the search for a given solution in a pool (`is_in_solutionpool`), the append of a new solution at the end of a pool with residual room (`add_solution_to_pool`), the insertion of a new solution in a pool of solutions sorted by increasing objective values (`update_best_set`), and the removal of all solutions from a pool (`clean_solutionpool`). The generation of a random solution, that was already implemented in Chapter 3, is provided in library `randomsolution`.

The practical implementation creates at the beginning and destroys in the end a current solution `x` and two pools of solutions `B` and `P`, representing the reference set and the population of candidate solutions. Two pools are required by the static update, that first builds all candidates and then tests them for insertion in the reference set. Pool `B` has maximum size `nb`; it is sorted by nondecreasing values of the objective, and solutions are added to it in the second loop by procedure `update_best_set`. Pool `P` is unsorted, solutions are added at the end of the available space in the first and the third loop; its size is `nb*(nb-1)`, because the first loop stops when it

[1] Typically, they are milliseconds or microseconds, depending on the machine. QUESTO VA ANTICIPATO NEL COMMENTO AL MAIN DEL PRIMO CAPITOLO, DATO CHE SI USA ANCHE LI', LASCIANDO QUI SOLO UN RICHIAMO.

reaches `nb` solutions and the third loop considers all pairs of solutions of `B`, though the actual number will probably be much smaller. At the end of the second loop, pool `P` is cleaned: that is simpler and more efficient than deallocating and reallocating it, exactly as it is preferable to clean the auxiliary solution `x` after its use[2]. The only procedure that is used and declared, but not yet implemented, is `find_relinking_solution`, that determines the relinking path and returns its best solution (excluding the two extremes x and y). This is then improved by steepest

[2] An interesting point to discuss on the management of solutions pools is whether to add copies of the new solutions, as it is done at present, or move them physically into the pool. I have not pondered this point enough to take a clear decision.

ascent and included among the candidate ones, if it is not a duplicate.

```
start = clock();
create_solution(pI->n,&x);
create_solutionpool(nb,&B);
create_solutionpool(nb*(nb-1),&P);

while (((double)clock() - start) / CLOCKS_PER_SEC < tauMax)
{
  /* Build or integrate the candidate population */
  while (P.card < nb)
  {
    clean_solution(&x,pI->n);
    generate_random_solution(pI,&x,pseed);
    steepest_ascent(pI,&x,visit_strategy,&iter);
    if (!is_in_solutionpool(&x,pI->n,&P)) add_solution_to_pool(&x,pI->n,&P);
  }

  /* Test the candidate solutions for insertion in the reference set */
  for (s = 1; s <= P.card; s++)
    update_best_set(P.S[s],pI->n,&B);
  clean_solutionpool(&P);

  /* Recombine the solutions in the reference set into candidate ones */
  for (s = 1; s <= B.card; s++)
    for (s2 = 1; s2 <= B.card; s2++)
      if (s != s2)
      {
        clean_solution(&x,pI->n);
        find_relinking_solution(B.S[s],B.S[s2],pI,"-gb",&x);
        steepest_ascent(pI,&x,visit_strategy,&iter);
        if (!is_in_solutionpool(&x,pI->n,&P)) add_solution_to_pool(&x,pI->n,&P);
      }
}
if (B.card > 0) copy_solution(B.S[1],px);

destroy_solution(&x);
destroy_solutionpool(&B);
destroy_solutionpool(&P);
```

The implementation of function `find_relinking_solution` builds upon the steepest ascent procedure. There are two main differences:

1. the procedure minimizes first the Hamming distance from the final solution y, then the objective function (in case of ties);
2. the procedure terminates when solution y is reached.

The special features of the *MDP*, in particular its only constraint fixing the cardinality of the solution, implies that:

1. the only way to reduce the Hamming distance of the current solution z from y (always exactly by 2) is to swap a point in $z \setminus y$ with a point in $y \setminus z$;
2. it is possible to focus on the moves that reduce the Hamming distance and avoid all other moves of N_{S_1};
3. there are $|z \setminus y| \cdot |y \setminus z|$ such moves, that is a positive number, as long as $z \neq y$: the best of these moves will be performed.

Since the size of this neighbourhood and the number of moves is probably much smaller than for a typical single-swap, we will adopt the global-best visit strategy for the relinking path identification. An experimental comparison with the first-best strategy should be performed, but we will not do it for lack of time.

```
create_solution(pI->n,&z);
copy_solution(px,&z);

clean_solution(pz,pI->n);
do
{
  explore_neighbourhood_for_relinking
      (&z,py,pI,visit_strategy,&p_in,&p_out,&delta_f);
  if (p_in != NO_POINT)
  {
    swap_points(p_in,p_out,&z,pI);
    if (z.f > pz->f) copy_solution(&z,pz);
  }

} while (p_in != NO_POINT);
```

The neighbourhood is explored in procedure `explore_neighbourhood_for_relinking` exactly as in the steepest ascent heuristic, with two limitations to guarantee that the solutions explored are closer to the final one than the current solution: the deleted point `p_in` must not belong to the final solution `y`, and the added point `p_out` must belong to the final solution `y`. This is similar to what happens in *Tabu Search*, but much simpler, as it is just a straightforward reduction of the neighbourhood. Notice that, if $k < n/2$, it would be more efficient to scan `p_out` in `py` and check that it is not in `px`.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px);
        p_in = next_point(p_in,px))
  if (!py->in_x[get_index(p_in,pI)])
    for (p_out = first_point_out(px); !end_point_list(p_out,px);
          p_out = next_point(p_out,px))
      if (py->in_x[get_index(p_out,pI)])
      {
        delta_f = evaluate_exchange(p_in,p_out,px,pI);
        if (delta_f > *pdelta_f)
```

```
    {
      *pdelta_f = delta_f;
      *pp_in = p_in;
      *pp_out = p_out;
    }
  }
```

F.1.1 Time complexity estimation

An *a priori* estimation of time complexity of the *PR* metaheuristic is quite complex, as it consists of procedures of a very different nature, that include conditions whose occurrence is usually unpredictable. Moreover, in our experiments the computational time is fixed by the user. However, the analysis can suggest the relative weight of the different components of the algorithm, and consequently on which procedures and parameters to focus in order to improve it.

The first inner loop, in which random candidates are generated and improved, is particularly hard to characterize: we know from Chapter 3 that random solutions are usually reduced to a locally optimal one in a number of neighbourhood explorations $t_{\max}$ that increases more than linearly with size. Of course, each exploration takes $O(k(n-k))$ time. The main problem is that this loop ends when the population P includes at least n_B solutions, which can take zero time (if the relinking paths are enough to fill it) or a potentially infinite time (if the random initializations repeat over and over again the same locally optimal solutions). A very rough estimate can be $O(n_B t_{\max} k(n-k))$ time.

The second inner loop tests all candidate solutions for insertion in B. In the worst case, the candidates could be $n_B(n_B-1)$; each one could require to scan all the reference solutions point by point (if all candidates and all reference solutions have the same value, an extremely unlikely case); finally, each candidate could be copied into the pool: overall, that would be $O\left(n_B^3 k + n_B^2 n\right)$ time. In practice, the candidate solutions are usually n_B and most of them are tested against some reference solutions using only the objective value (a single reference solution will be enough to reject bad candidates), so the time could be as low as $O(n_B)$.

The third inner loop draws a relinking path for $n_B(n_B-1)$ pairs of reference solutions. Each path takes at most k neighbourhood explorations and each exploration takes $O(k(n-k))$ time, as usual (probably with smaller multiplying coefficients, due to the limitation of the neighbourhood). Then, the best solution along the path is improved by steepest ascent. Overall, this should be $O\left(n_B^2\left(k^2(n-k)+t_{\max}k(n-k)\right)\right)$ time, where $t_{\max}$ is probably smaller than for a random initialization. This is probably the most expensive component of the algorithm, though the first one could also be relevant. A profiler could confirm or disprove it empirically, but its use exceeds the scope of the present discussion[3].

F.1.2 Empirical evaluation

We can now evaluate the performance of the *PR* metaheuristic. We will experiment with typical values for the number of reference solutions n_B, ranging from 5 to 20. Smaller values would not be enough to generate enough new solution by recombination, whereas larger values would take too much time to systematically test all pairs and would also probably generate many duplicate or bad quality locally optimal solutions. Another possible experiment could be to compare the effect of using the global-best or the first-best strategy in the two exchange procedures that,

[3]Next year, may be.

respectively, improve the recombined and the random solutions or build the relinking paths. For the sake of briefness, we will not perform this analysis.

The only other parameter is the total time. From the previous chapters we know that for the largest instances of the benchmark 1 000 neighbourhood explorations correspond to about 4 seconds and that a randomly generated solution requires around 30 neighbourhood explorations. Each generation of the *PR* framework requires to improve about n_B^2 solutions, some of which are fully random, whereas most derive from the relinking paths. A very rough computation suggests that a generation could take about $20^2 \cdot 30/1\,000 \cdot 4 = 48$ seconds. This is quite long with respect to the time used in the previous chapters, though it is probably an overestimate, because the solutions derived from the relinking paths are probably quicker to optimize. As a compromise, we will set the time to $t_{\max} = 30$ seconds.

Notice that if the time limit is tested only at the beginning of each generation, it could expire before the generation has ended, but the algorithm would terminate only later, violating the limit. This happens also for the random restart approach, because the steepest ascent procedure runs to completion before testing the time limit. To avoid these violations, the condition should be tested more often; possibly, a truncated version of steepest ascent should be designed. However, for the sake of simplicity we will skip this step. Our experiments, in fact, show times ranging from 30 to 30.9 seconds: a more precise termination would be desirable, but is not strictly required for our rough analysis.

F.1.3 Parameter tuning

We now compare the effect on the quality of the solution of four different sizes of the reference set, namely $n_B = 5$, 10, 15 and 20. Table F.1 reports the resulting average gaps over the whole benchmark. For comparison purposes, the table also reports the average gap obtained by the random restart approach. The best performing configurations set $n_B = 15$ and $n_B = 20$.

	PR				*RR*
n_B	5	10	15	20	
Gap	0.26%	0.20%	0.12%	0.12%	0.10%

Table F.1: Average gaps with respect to the best known result of the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (*RR*)

Figure F.1 reports the *SQD* diagram of the four configurations, confirming the relations suggested by the average gap (possibly with a prevalence of $n_B = 15$ on $n_B = 20$, since the latter looks more or less dominated by random restart, whereas the former does not).

The corresponding boxplots are reported in Figure F.2, and approximately suggest the same conclusions.

Finally, applying Wilcoxon's test to compare the *PR* algorithm with $n_B = 15$ with the other tunings we obtain the following results:

1. $n_B = 15$ versus $n_B = 5$

```
W+ = 569, W- = 134, N = 37, p <= 0.001062
```

2. $n_B = 15$ versus $n_B = 10$

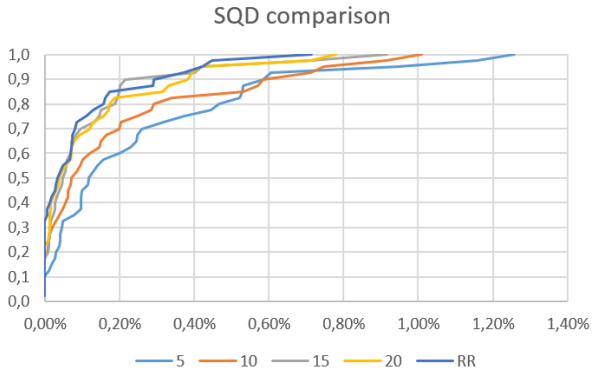


Figure F.1: Solution Quality Distribution diagrams for the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (*RR*)

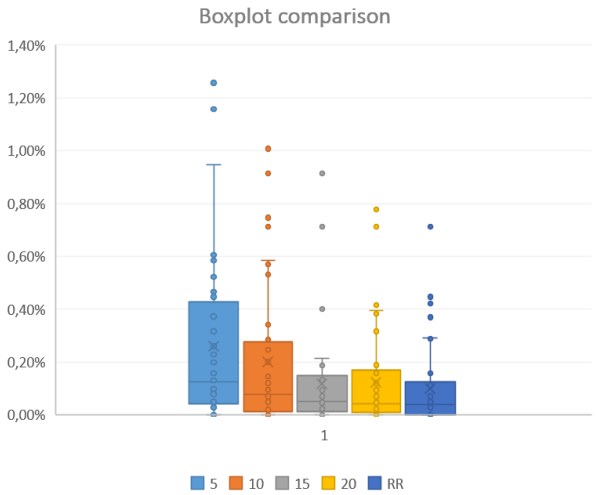


Figure F.2: Boxplot diagrams for the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (*RR*)

```
W+ = 206, W- = 119, N = 25, p <= 0.2473
```

3. $n_B = 15$ versus $n_B = 20$

```
W+ = 144, W- = 109, N = 22, p <= 0.581
```

which suggests that only the first comparison is significant, whereas the other ones could easily be the result of a random sampling. So, the results are inconclusive.

F.1.3.1 Comparison with random restart

The comparison between the *PR* metaheuristic and the steepest ascent with purely random restart allows to estimate the contribution of recombination to the performance. In fact, the *PR* metaheuristic adopts the same method to initialize the population and to integrate it when recombination is not enough to fill the required minimum number of solutions. From the tables and pictures presented above, all configurations of the *PR* algorithm appear to be worse than random restart: even the best one has an average gap of 0.12% versus 0.10%, and a *SQD* diagram that is dominated in most of the range, though not for all values.
Wilcoxon's test however yields the following results:

1. *RR* versus $n_B = 5$

```
W+ = 411, W- = 184, N = 34, p <= 0.05337
```

2. *RR* versus $n_B = 10$

```
W+ = 301, W- = 294, N = 34, p <= 0.9591
```

3. *RR* versus $n_B = 15$

```
W+ = 247, W- = 314, N = 33, p <= 0.5554
```

4. *RR* versus $n_B = 20$

```
W+ = 185, W- = 221, N = 28, p <= 0.6903
```

While we could expect some tests to be inconclusive, it is rather surprising to find out that all of them are. Analyzing the detailed results, one can remark that random restart seems to perform better on smaller instances and worse on larger ones. Now, our application of Wilcoxon's test considers the absolute differences, whereas the *SQD* considers the relative differences, or percent gaps. The former favours the algorithm that is better on large instances (*PR*), whereas the latter favours the algorithm that is better on small instances (*RR*). This could perhaps explain the results. Anyway, the most correct interpretation is probably to suspend any judgment.

Another interesting insight can be obtained by plotting the values of the solutions met along the relinking paths: we can see that typically the objective is good (that is, large) in the two extreme solutions and decreases in the intermediate ones. Therefore, the best solution is usually the first or the last along the path, that is a neighbour of the given locally optimal solutions[4]: the subsequent exchange heuristic is unlikely to find better solutions. This means that the improvement is mainly due to the random generation step, and therefore increasing the time dedicated to this step by removing the recombination mechanism is actually profitable. In the largest instances, however, this is not always the case: the recombination mechanism starts yielding new solutions, whose quality is better than that of the random ones. These are however just speculations, working hypothesis for further studies, that should be verified checking the actual behaviour of the algorithm on some instances.

F.2 Scatter search

The *Scatter Search* metaheuristic also manages a *reference set R*, that is however composed of two subsets: B includes the best known solutions, D includes the most diverse solutions. The basic scheme extracts pairs of solutions, one from B and one from $B \cup D$, and combines them with a suitable procedure. Then, it improves the new solution by an exchange procedure and checks whether the locally optimal solution thus obtained is worth inserting in B (first) or in D (then). The process terminates when all solutions generated in this way are rejected, and the reference set is not updated. This scheme has an intrinsic termination condition, at least if the recombination procedure is deterministic. In fact, applying it to the same set of reference solutions, it generates the same locally optimal solutions. If the recombination mechanism is randomized, different solutions could be obtained in different iterations, but the process is very intensifying, so that, even if it is profitable at first, in the long term it is likely to lead to repeated solutions and stagnation.

In order to implement a potentially unlimited algorithm with a user-defined time limit, we exploit the same mechanism used for *PR* in the previous section, that is also the same mechanism used to generate the starting population: new solutions are produced by generating random solutions and improving them with steepest ascent. Then, all solutions of the new population are tested for insertion in the reference set, first in B and then in D. For the sake of simplicity, we will adopt once again a static update mechanism.

The resulting scheme is the following.

[4] A possible idea could be to abandon the standard scheme of *PR* and improve an intermediate solution, instead of the best one along the relinking path.

Algorithm ScatterSearch($I, \tau_{\max}, n_B, n_D$)
$B := \emptyset; D := \emptyset; P := \emptyset;$
While Time() $\leq \tau_{\max}$ *do*
  { Build or integrate the candidate population }
  While $|P| < n_B + n_D$ *do*
    $x :=$ RandomSolution(I);
    $x :=$ SteepestAscent(x);
    If $x \notin P$ *then* $P := P \cup \{x\}$;
  EndWhile
  { Test the candidate solutions for insertion in the reference set }
  For each $x \in P$ *do*
    If $x \notin B$ *then* $B :=$ UpdateBestSet(B, x);
    If $x \notin B \cup D$ *then* $D :=$ UpdateDiverseSet(D, x, B);
  EndFor
  { Recombine the solutions in the reference set into candidate ones }
  For each $(x, y) \in B \times (B \cup D)$ *do*
    $P := P \cup$ Recombine(x, y, I);
  EndFor
EndWhile
$x^* := \arg\max_{x \in B} f(x)$;
Return $(x^*, f(x^*))$;

Notice the strong structural similarity with the scheme of *PR*. The first inner loop builds a population of candidate solutions by generating random solutions and improving them with steepest ascent. For the sake of simplicity, the number of candidates is set large enough to fill the reference set ($n_B + n_D$). As in *PR*, duplicate candidates are not accepted, because they would provide no advantage, and a (very limited) risk of not terminating the loop exists.

The second inner loop tests each candidate solution to check whether it deserves to be included in either of the two subsets B and D. This is done with suitable procedures that, in the positive case, also add the solution to update the subset. These procedures return `true` or `false` to indicate whether the candidate solution has been accepted or not. A solution added to B is not tested on D. Since the definition of "diverse" solution refers to both subsets, the update of the diverse set requires set B, or at least information on the Hamming distance of each current solution $y \in D$ from $B \cup D \setminus \{y\}$. At the end, the population P is cleaned, as in *PR*.

The third inner loop considers each pair of solutions (x, y) from $B \times (B \cup D)$ and recombines them to produce a solution z that is saved in pool P. Now the algorithm goes back to the random generation, in case the recombined solutions are not enough to provide the sufficient number of solutions to update the reference set.

The implementation can exploit many of the procedures already discussed: those provided in library `solutionpool` to manages solution pools, and the generation of random solutions. Of course, we need an additional pool `D`. As mentioned above, the update of the diverse set requires information on the Hamming distance: the two vectors `Hmin` and `Htot` provide, respectively, the minimum and total distance of each reference solution from the other ones. We use both the minimum and total distance to better discriminate between solutions: in fact, the Hamming

distance is an integer value between 1 and k, and therefore it could easily assume identical values for several solutions. The update procedure should take care to keep these vectors sorted in the same way as the solutions in D.

Procedure `recombine_solutions` simply recombines two solutions into a third one in one of the several ways discussed in the theoretical lessons. In the present phase, we simply declare it

without defining its content.

```
start = clock();
create_solutionpool(nb,&B);
create_solutionpool(nd,&D);
create_solutionpool(nb*(nb+nd),&P);

create_solution(pI->n,&x);

while (((double)clock() - start) / CLOCKS_PER_SEC < tauMax)
{
  /* Build or integrate the candidate population */
  while (P.card < nb + nd)
  {
    generate_random_solution(pI,&x,pseed);
    steepest_ascent(pI,&x,visit_strategy,&iter);
    if (!is_in_solutionpool(&x,pI->n,&P))
    {
     add_solution_to_pool(&x,pI->n,&P);
    }
    clean_solution(&x,pI->n);
  }

  /* Test the candidate solutions for insertion in the reference set */
  for (s = 1; s <= P.card; s++)
  {
    insert = update_best_set(P.S[s],pI->n,&B);
    if (!insert) insert = update_diverse_set(P.S[s],pI->n,&D);
  }

  /* Recombine the solutions in the reference set into candidate ones */
  for (s = 1; s <= B.card; s2++)
  {
    for (s2 = 1; s2 <= B.card; s2++)
      if (s != s2)
      {
        recombine_solutions(B.S[s],B.S[s2],pI,&x);
        add_solution_to_pool(&x,pI->n,&P);
      }
    clean_solutionpool(&P);

    for (s2 = 1; s2 <= D.card; s2++)
    {
      recombine_solutions(B.S[s],D.S[s2],pI,&x);
      add_solution_to_pool(&x,pI->n,&P);
    }
  }
}

if (nb > 0) copy_solution(B.S[1],px);
```



```
destroy_solutionpool(&B);
destroy_solutionpool(&D);
destroy_solutionpool(&P);
```

It is however more complicated because pool `D` is sorted with respect to the (minimum and total) Hamming distance from $B \cup D$. Therefore...

PROBLEM: THE DEFINITION OF POOL DOES NOT INCLUDE FIELDS FOR Htot AND Hmin. USING EXTERNAL VECTORS MAKES THE CODE MORE COMPLEX AND POSES THE QUESTION WHETHER THE FUNCTION SHOULD BE INCLUDED OR NOT IN THE LIBRARY, PLUS THE FOLLOWING ADDITIONAL PROBLEMS: WHEN ARE THE HAMMING DISTANCES OF THE RECOMBINED SOLUTION COMPUTED? WHERE ARE THEY COMPARED WITH THE HAMMING DISTANCES OF THE SOLUTIONS IN D?

```
TO BE COMPLETED
```

The core of the *SS* metaheuristic is the recombination procedure. Usually, its first step consists in initializing the new solution `x` with the intersection of the two parent solutions `x1` and `x2`. Then, the partial solution is augmented with elements drawn from the two parents. This can be performed in several ways, concerning two main aspects:

- whether the choice of the elements is random or greedy (or any semigreedy combination): the first approach allows a given pair of parent solutions to generate many different new solutions, the second usually generates better solutions;
- whether the elements are chosen alternatively from the two parents or freely: the first approach guarantees the largest possible distance from the two parent solutions, the latter allows a larger variety (in particular, if combined with choices at least partly random).

In the following, for the sake of simplicity, we will draw random elements alternatively from the two parent solutions, counting on the steepest ascent procedure as a tool to generate good solutions (rather than on a greedy choice) and on the recombination of good solutions as a tool to intensify the search (rather than on a biased extraction favouring one parent). It is in general possible that the constraints of the problem forbid to build a whole solution simply drawing random elements from the two parents. In the case of the *MDP*, however, this is possible, thanks to the very simple cardinality constraint. We have therefore simply to create an empty solution, add to it the points that belong to both parents (for example, by scanning one and checking which of its elements also belong to the other), put the other points of both solutions in suitable vectors from which they can be extracted at random, and perform the extraction as already done in several occasions in the previous chapters.

UPDATE: FIRST, SCAN A SOLUTION AND PUT IN THE DESTINATION THE POINTS THAT ARE ALSO IN THE OTHER SOLUTION. THEN, BUILD VECTORS FOR THE REMAINING CANDIDATE POINTS FROM THE TWO SOURCES. FINALLY, ALTERNATIVELY EXTRACT ONE POINT FROM EACH OF THE TWO SOURCES, AND ADD IT TO THE RECOMBINED SOLUTION. THERE IS MUCH IN COMMON WITH OTHER RANDOM EXTRACTIONS USED IN THE PREVIOUS CHAPTERS.

```
TO BE COMPLETED
```

F.2.1 Time complexity estimation

A time complexity estimation of the *SS* metaheuristic is quite complex, as it consists of procedures of a very different nature, including tests whose outcome is usually impossible to predict *a*

priori. The generation of the reference set, for example, requires a random extraction in $O(kn)$ time, where coefficient n depends on the update of vector D. This apparently plays no role, and therefore seems damaging, but it still strongly accelerates the steepest ascent procedure, that probably gives a large contribution to the overall computational time: an assumption to be verified empirically. Then, it requires a steepest ascent procedure, whose complexity has already been characterized as $O(t_{\max}(n-k)k)$, where $t_{\max}$ is the number of neighbourhood exploration from a random solution to the local optimum in which steepest ascent terminates, that is an unknown function of n and k. The generation is repeated until B and D are both full, which requires at least $n_B + n_D$ iterations, but it could require many more, if the locally optimal solutions found are often the same. This depends strongly on chance and on the landscape of the problem (the *MDP* should have a sufficiently rugged landscape to generate many different local optima, but this is just a guess). Then, $n_B(n_B + n_D - 1)$ pairs of solutions are taken into account[5]. For each pair, the recombination procedure requires to find the intersection and the two set differences of the parent solutions (in time $O(k)$) and to add the points of the intersection and random points of the differences to build the new solution (in time $O(n)$ for each of the k points, once again for the update of vector D). Each solution must be checked for insertion in the best set B in time $O(k)$ (in the worst case, that becomes probably quite rare after a while, because it corresponds to improving most of the best solutions at every attempt). If this fails, it must be checked for insertion in the diverse set B in time WHO KNOWS? The main problem is that there is no way to predict how many times these operations will be repeated overall. Therefore, the theoretical analysis is mainly useful to determine the relative weight of the different contributions: the steepest ascent procedure appears to be the most computationally intensive, and this justifies the choice to keep vector D even if it slows down some of the other procedures.

F.2.2 Empirical evaluation

We can now evaluate the performance of the *SS* metaheuristic. We consider reference sets of the same size used for *PR*. In this case, however, we have two subset, B and D. For the sake of simplicity, we assume them to have the same size. Since the pairs of solution to be recombined are $n_B(n_B + n_D - 1)$, we will experiment with the following configurations: $n_B = n_D = 5$, $n_B = n_D = 10$ and $n_B = n_D = 15$. In this way, the number of pairs will not be very different from the one considered by *PR*.

OTHER POSSIBLE PARAMETERS: GREEDY CHOICE VERSUS RANDOM CHOICE; FREE CHOICE VERSUS ALTERNATE CHOICE

F.2.2.1 Computational time analysis

As for *PR* the computational time is fixed to 30 seconds overall.

TEST WHETHER THE TIME IS VIOLATED DUE TO THE CHECK MADE ONLY AT THE BEGINNING OF EACH ITERATION (PRESUMABLY NOT). IF THIS HAPPENS, ADD OTHER CHECKS.

[5]This means that every pair of best solutions is considered twice: once in each direction. The random choice of elements during the recombination suggests that such a repetition could be nonredundant, but this should be verified empirically.

F.2.3 Parameter tuning

AVERAGE RESULTS

Figure F.3 reports the *SQD* diagram of the *SS* metaheuristic, compared with that of the steepest ascent heuristic applied to purely random solutions. This allows to estimate the contribution of recombination to the overall performance. In fact, the *SS* metaheuristic adopts the same method to initialize the population and to integrate it when recombination is not enough to fill the required minimum number of solutions

Figure F.3: Solution Quality Distribution diagram for the *SS* metaheuristic compared with the *steepest ascent* heuristic initialised with random solutions with a time limit of 30 seconds

The boxplots reported in Figure F.4 provide the same information: SUMMARY OF THE PICTURE

Figure F.4: Boxplots for the *SS* metaheuristic compared with the *steepest ascent* heuristic initialised with random solutions with a time limit of 30 seconds

WILCOXON'S TEST

F.3 Comparison with *PR* and random restart