Università degli Studi di Milano

Heuristic algorithms (laboratory sessions)

Roberto Cordone

– Academic year 2019/20 –

Contents

Introduction

The laboratory sessions of the course on Heuristic algorithms have the 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 $\mathcal{NP}$-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.

Chapter 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. Chapter 2 describes the implementation and evaluation of some constructive and destructive heuristics.

Chapter 1

The Maximum Diversity Problem

1.1 Definition

The *Maximum Diversity Problem* (*MDP*) is defined by:

- a set P of *points* in an abstract space (with $n = |P|$;
- 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 $k \in \mathbb{N}$ con $0 < k < |P|$.

The problem consists in determining a subset $x \subset P$ such that

- the sum of the pairwise distances between the points of x be maximum.
- the cardinality of x be equal to k;

$$\begin{aligned} \max_{x \subseteq P} f &= \sum_{i \in x} \sum_{j \in x} d_{ij} \\ |x| &= k \end{aligned}$$

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. Finally, 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 $(d_{ij} + d_{ji})/2$, 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.

$$d_{ii} = 0 \text{ for all } i \in P$$

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.

$$d_{ij} \geq 0 \text{ per ogni } i, j \in P$$

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:

$$d_{ij} + d_{jk} \geq d_{ik} \text{ per ogni } i, j, k \in P$$

There are *MDP* instances which enjoy it and instances which do not.

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 ...
...
```

1.3 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.

1.4 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 $O\left(n^2\right)$ time if done scanning the incidence vector, $O\left(k^2\right)$ 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

        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:

[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.

```
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.

1.5 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.

1.6 The main function

The `main` function in all the algorithms presented in the following chapters 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.

Chapter 2

Constructive heuristics

2.1 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.

2.2 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.

2.2.1 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 ($f_A(x)$).

Computational time analysis

Figure 2.1 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 2.2) 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.

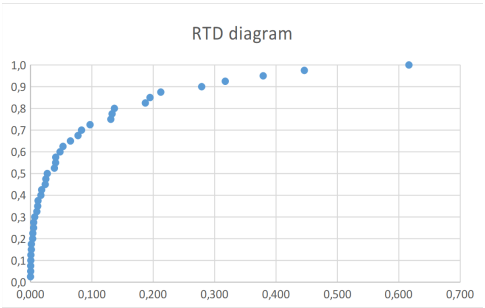

Figure 2.1: *RTD* diagram for the greedy algorithm on the benchmark

The correct tool to describe the dependence of the computational time T_A on the instance size P is the *scaling diagram* (see Figure 2.3). 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 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

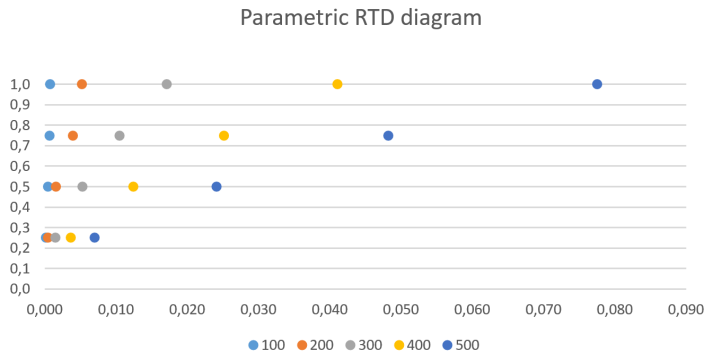

Figure 2.2: Parametric *RTD* diagrams for the greedy algorithm on the benchmark

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 $T_A \in O\left(n^3\right)$.

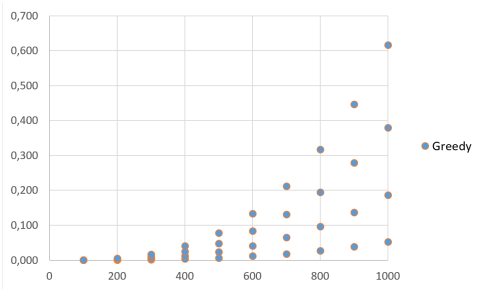

Figure 2.3: Scaling diagram for the greedy algorithm on the benchmark

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 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,

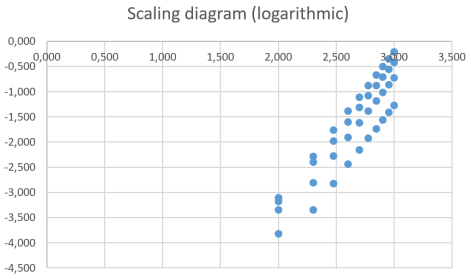

Figure 2.4: Scaling diagram in logarithmic scales for the greedy algorithm on the benchmark

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 2.1 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...
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

[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 2.1: Results of the basic greedy algorithm

necessarily zero. Something is clearly wrong in the selection criterium φ, at least at the first step.

2.3 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 2.5 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.

2.4 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.

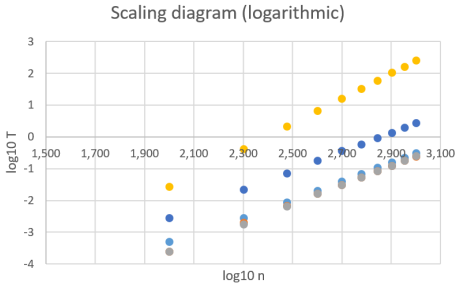

Figure 2.5: Scaling diagram for the four greedy algorithms variants (and the stingy one) on the benchmark

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 $O(n^3)$, that is of the same order as nk^2, but with larger multiplying factors. In fact, the scaling diagram of Figure 2.5 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.

2.5 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 2.6 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 chapter. 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.

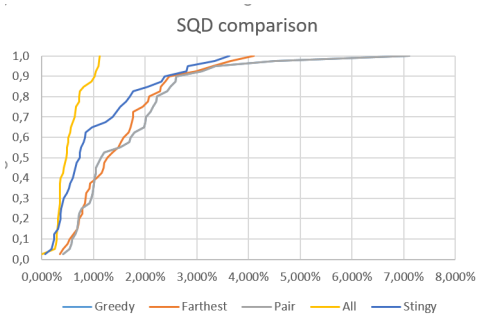

Figure 2.6: Solution Quality Distribution diagram for the four greedy algorithms variants (and the stingy one) on the benchmark

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 2.7.

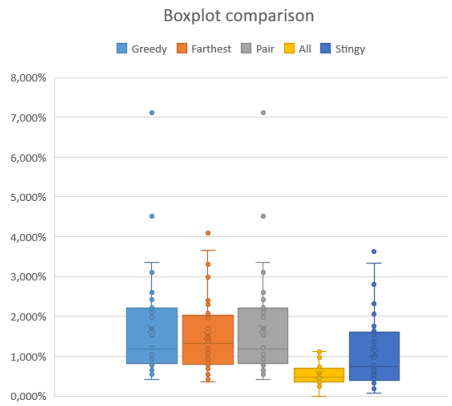

Figure 2.7: Boxplots for the four greedy algorithms variants (and the stingy one) on the benchmark

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 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 2.8)? Is it different for different algorithms?

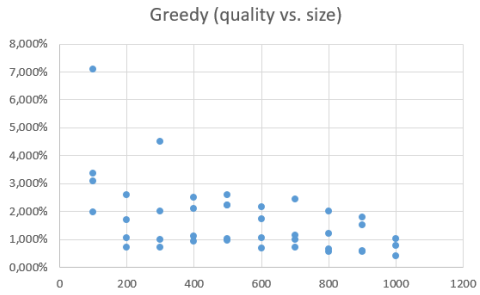

Figure 2.8: Scaling diagram for the quality of the greedy algorithm with respect to the size of the instances of the benchmark

2.6 Roll-out heuristic

TO BE DONE