

Heuristic Algorithms for Combinatorial Optimization problems

Ph.D. course in Computer Science

Roberto Cordone
DI - Università degli Studi di Milano



E-mail: roberto.cordone@unimi.it

Web page: <https://homes.di.unimi.it/cordone/courses/2023-haco/2023-haco.html>

Exchange algorithms

In Combinatorial Optimization every solution x is a subset of B

An **exchange heuristic** updates a current subset $x^{(t)}$ step by step

- 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, i.e. add subsets A external to $x^{(t)}$ and delete subsets D internal 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 criterium** $\varphi(x, A, D)$ 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

Neighbourhood

An exchange heuristic is defined by:

- 1 the pairs of exchangeable subsets (A, D) in every solution x ,
i.e. the solutions generated by a single exchange starting from x
- 2 the selection criterium $\varphi(x, A, D)$

Neighbourhood $N : X \rightarrow 2^X$ is a function which associates to each feasible solution $x \in X$ a subset of feasible solutions $N(x) \subseteq X$

The situation can be formally described with a search graph in which

- the nodes represent the feasible solutions $x \in X$
- the arcs connect each solution x to those of its neighbourhood $N(x)$,
moving elements into and out of x (they are often denoted as moves)

Every run of the algorithm corresponds to a path in its search graph

How does one define a neighbourhood and select a move?

Neighbourhoods based on distance

Every solution $x \in X$ can be represented by its **incidence vector**

$$\xi_i(x) = \begin{cases} 1 & \text{if } i \in x \\ 0 & \text{if } i \in B \setminus x \end{cases}$$

Hamming distance between two solutions x and x'
is the **number of elements in which their incidence vectors differ**

$$d_H(x, x') = \sum_{i \in B} |\xi_i(x) - \xi_i(x')|$$

Referring to the subsets, $d_H(x, x') = |x \setminus x'| + |x' \setminus x|$

A typical definition of neighbourhood, with an integer parameter k , is the **set of all solutions with a Hamming distance from x not larger than k**

$$N_{H_k}(x) = \{x' \in X : d_H(x, x') \leq k\}$$

Example: the KP

The KP instance with $B = \{1, 2, 3, 4\}$, $v = [5 \ 4 \ 3 \ 2]$ and $V = 10$, has 13 feasible solutions out of 16 subsets

(0111)	(1111)	(0001)	(0000)
(0011)	(1011)	(1010)	(0110)
(1110)	(1001)	(1000)	(0101)
(1100)	(1101)	(0010)	(0100)

since subsets $\{1, 2, 3, 4\}$, $\{1, 2, 3\}$ and $\{1, 2, 4\}$ are unfeasible

10 subsets (pink) have Hamming distance ≤ 2 from $x = \{1, 3, 4\}$ (blue)

The neighbourhood $N_{H_2}(x)$ consists of the 7 feasible subsets in pink

$N_{H_2}(x)$ excludes

- the 3 crossed subsets in pink because they are unfeasible
- the 5 subsets in black because their Hamming distance from x is > 2

Neighbourhoods based on operations

Another common definition of neighbourhood is obtained defining

- a family \mathcal{O} of operations on the solutions of the problem
- the set of all solutions generated applying to x the operations of \mathcal{O}

$$N_{\mathcal{O}}(x) = \{x' \in X : \exists o \in \mathcal{O} : o(x) = x'\}$$

Considering again the KP , \mathcal{O} can be defined as

- adding to x an element of $B \setminus x$
- deleting from x at most an element (*“at most” to impose $x \in N(x)$*)
- swapping one element of x with one of $B \setminus x$

The resulting neighbourhood $N_{\mathcal{O}}$ is related to those defined by the Hamming distance, but does not coincide with any of them

$$N_{H_1} \subset N_{\mathcal{O}} \subset N_{H_2}$$

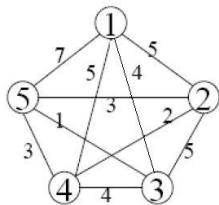
As the distance-based ones, these neighbourhoods can be parameterised considering sequences of k operations of \mathcal{O} instead of a single one

$$N_{\mathcal{O}_k}(x) = \{x' \in X : \exists o_1, \dots, o_k \in \mathcal{O} : o_k(o_{k-1}(\dots o_1(x))) = x'\}$$

Distance and operation-based neighbourhoods

In general, an operation-based neighbourhood includes solutions with different Hamming distances from x

For the *TSP* one can define a neighbourhood N_{S_1} including the solutions obtained swapping two nodes in their visit order



The neighbourhood of solution $x = (3, 1, 4, 5, 2)$ is:

$$N_{S_1}(x) = \{(1, 3, 4, 5, 2), (4, 1, 3, 5, 2), (5, 1, 4, 3, 2), (2, 1, 4, 5, 3), (3, 4, 1, 5, 2), (3, 5, 4, 1, 2), (3, 2, 4, 5, 1), (3, 1, 5, 4, 2), (3, 1, 2, 5, 4), (3, 1, 4, 2, 5)\}$$

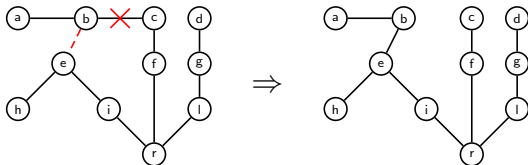
If the two nodes are adjacent, the modified arcs are $3 + 3$; otherwise, they are $4 + 4$

Different neighbourhoods for the same problem: the *CMST*

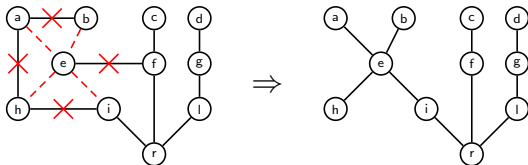
Different ground sets yield different neighbourhoods

In the *CMST* it is possible to set $B = E$ or $B = V \times T$

- exchange edges: delete (b, c) , add (b, e)



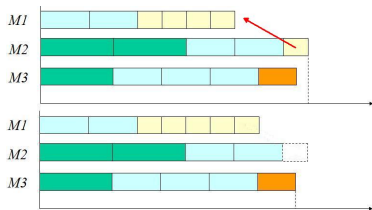
- exchange vertices: move e from subtree 2 to subtree 1, and recompute the two minimum spanning subtrees



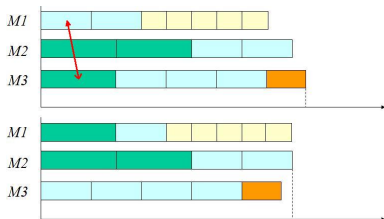
Different neighbourhoods for the same problem: the *PMSP*

For the *PMSP* it is possible to define

- the transfer neighbourhood $N_{\mathcal{T}_1}$, based on the set \mathcal{T}_1 of all transfers of a task on another machine



- the swap neighbourhood $N_{\mathcal{S}_1}$, based on the set \mathcal{S}_1 of the swaps of two tasks between two machines (one task for each machine)



Connectivity of the search graph

An exchange heuristic can return the optimum only if every feasible solution can reach at least one optimal solution, that is there is a path from x to X^* for every $x \in X$

Such a search graph is denoted as weakly connected to the optimum

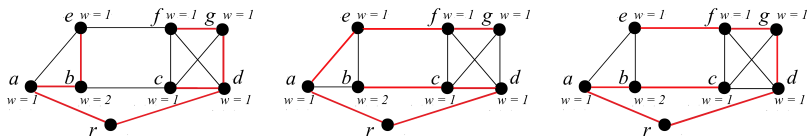
Since X^* is unknown, a stronger condition is often used: a search graph is strongly connected when it admits a path from x to y for every $x, y \in X$

A good neighbourhood should guarantee some connectivity conditions

- in the *MDP*, neighbourhood N_{S_1} connects any pair of solutions with at most k swaps
- in the *KP* and the *SCP*, no neighbourhood N_{S_k} gives that guarantee
(feasible solutions can have any cardinality)
- the search graph becomes connected also in the *KP* and the *SCP* if swaps are combined with both additions and deletions

Connectivity of the search graph

If feasibility is defined in a sophisticated way, exchanging, adding and deleting **single** elements can be insufficient to reach all solutions :
the unfeasible subsets can break all paths between some feasible solutions



If $V = 4$, only three solutions are feasible, all with two subtrees:

- $x = \{(r, a), (a, b), (b, e), (r, d), (c, d), (d, g), (f, g)\}$
- $x' = \{(r, a), (a, e), (e, f), (r, d), (c, d), (b, c), (f, g)\}$
- $x'' = \{(r, a), (a, b), (b, c), (r, d), (d, g), (e, f), (f, g)\}$

The three solutions are mutually reachable only exchanging at least two edges at a time; exchanging only one yields unfeasible subsets

Steepest descent (hill-climbing) heuristics

The simplest selection criterium $\varphi(x, A, D)$ is the objective function

It is used in nearly all exchange heuristics

When $\varphi(x, A, D) = f(x \cup A \setminus D)$, the heuristic moves from $x^{(t)}$ to the best solution in $N(x^{(t)})$

To avoid cyclic behaviour, only strictly improving solutions are accepted
Consequently, the best known solution is the last visited one

Algorithm SteepestDescent($I, x^{(0)}$)

$x := x^{(0)}$;

Stop := *false*;

While Stop = *false* *do*

$\tilde{x} := \arg \min_{x' \in N(x)} f(x')$;

If $f(\tilde{x}) \geq f(x)$ *then* Stop := *true*; *else* $x := \tilde{x}$;

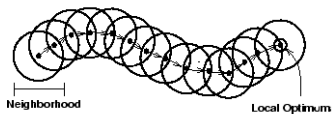
EndWhile;

Return ($x, f(x)$);

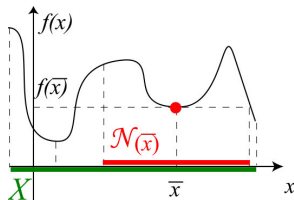
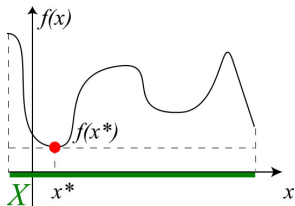
Local and global optimality

A *steepest descent* heuristic terminates when it finds a **locally optimal solution**, that is a solution $\bar{x} \in X$ such that

$$f(\bar{x}) \leq f(x) \text{ for each } x \in N(x)$$



A globally optimal solution is always also locally optimal, but the opposite is not true in general: $X^* \subseteq \bar{X}_N \subseteq X$



Exact neighbourhood

Exact neighbourhood is a neighbourhood function $N : X \rightarrow 2^X$ such that each local optimum is also a global optimum

$$\bar{X}_N = X^*$$

Trivial case: the neighbourhood of each solution coincides with the whole feasible region ($N(x) = X$ for each $x \in X$)

It is a useless neighbourhood: too wide to explore

The exact neighbourhoods are extremely rare

- exchange between edges for the **Minimum Spanning Tree problem**
- exchange between basic and nonbasic variables used by the **simplex algorithm for Linear Programming**

In general, the *steepest descent* heuristic does not find a global optimum
Its effectiveness depends on the properties of search graph and of objective

Properties of the search graph (1)

Some relevant properties for the effectiveness of an algorithm are

- the **size of the search space** $|X|$
- the **connectivity of the search graph** (as discussed above)
- the **diameter of the search graph**, that is the **number of arcs of the minimum path between the two farthest solutions**:
larger neighbourhoods produce graphs of smaller diameter
(*but other factors exist: see the “smallworld” effect*)

Consider neighbourhood N_{S_1} for the symmetric *TSP* on complete graphs

- the search space includes $|X| = (n - 1)!$ solutions
- N_{S_1} (swap of two nodes) includes $\binom{n}{2} = \frac{n(n-1)}{2}$ solutions
- the search graph is strongly connected and has diameter $\leq n - 2$:
every solution turns into another after at most $n - 2$ swaps

For example, $x = (1, 5, 4, 2, 3)$ becomes $x' = (1, 2, 3, 4, 5)$ in 3 steps

$$x = (1, 5, 4, 2, 3) \rightarrow (1, 2, 4, 5, 3) \rightarrow (1, 2, 3, 5, 4) \rightarrow (1, 2, 3, 4, 5) = x'$$

(*the first node is always 1, the last one is automatically in place*)

Properties of the search graph (2)

Other relevant properties

- the **density of global optima** ($\frac{|X^*|}{|X|}$) and **local optima** ($\frac{|\bar{X}_N|}{|X|}$):
if the local optima are numerous, it is hard to find the global ones
- the **distribution of the quality** $\delta(\bar{x})$ of **local optima** (SQD diagram):
if local optima are good, it is less important to find a global one
- the **distribution of the locally optimal solutions** in the search space:
if local optima are close to each other, it is not necessary to explore the whole space

These indices would require an exhaustive exploration of the search graph

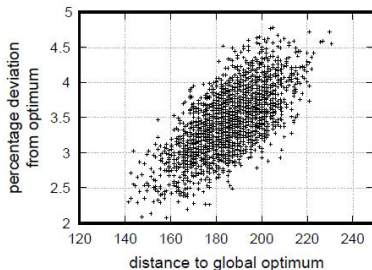
In practice, one performs a **sampling** and these analyses

- require **very long times**
- can be **misleading**, especially if the global optima are unknown

Example: the *TSP*

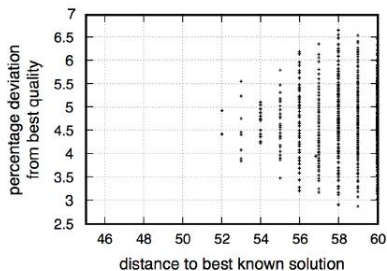
For the *TSP* on a complete symmetric graph with Euclidean costs

- the Hamming distance between two local optima is on average $\ll n$: the local optima concentrate in a small region of X
- the Hamming distance between local optima on average exceeds that between local and global optima: the global optima tend to concentrate in the middle of local optima
- the *FDC diagram* (*Fitness-Distance Correlation*) reports the quality $\delta(\bar{x})$ versus the distance from global optima $d_H(\bar{x}, X^*)$: if they are correlated, better local optima are closer to the global ones



Fitness-Distance Correlation

For the *Quadratic Assignment Problem (QAP)*, the situation is different



If quality and closeness to the global optima are strongly correlated

- it is profitable to build good starting solutions, because they drive the search near a good local optimum
- it is better to intensify than to diversify

If the correlation is weak

- a good initialization is less important
- it is better to diversify than to intensify

Landscape

The **landscape** is the triplet (X, N, f) , where

- X is the **search space**, or the set of feasible solutions
- $N : X \rightarrow 2^X$ is the **neighbourhood function**
- $f : X \rightarrow \mathbb{N}$ is the **objective function**

It is the **search graph with node weights given by the objective**

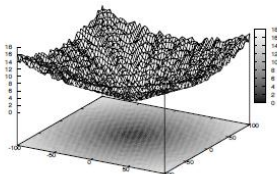
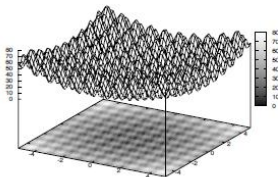
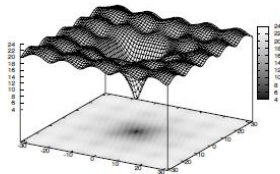
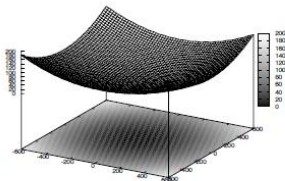


The effectiveness of steepest descent depends on the landscape

- smooth landscapes yield few local optima, possibly of good quality, hence to good results
- rugged landscapes yield several local optima of widespread quality, hence to bad results

Different kinds of landscape

There is a great variety of landscapes, very different from one another



Autocorrelation coefficient (1)

The complexity of a landscape can be empirically estimated

- ① performing a *random walk* in the search graph
- ② determining the sequence of values of the objective $f^{(1)}, \dots, f^{(t_{\max})}$

- ③ computing the **sample mean** $\bar{f} = \frac{\sum_{t=1}^{t_{\max}} f^{(t)}}{t_{\max}}$

- ④ computing the **empirical autocorrelation coefficient**

$$r(i) = \frac{\sum_{t=1}^{t_{\max}-i} (f^{(t)} - \bar{f})(f^{(t+i)} - \bar{f})}{\frac{\sum_{t=1}^{t_{\max}-i} (f^{(t)} - \bar{f})^2}{t_{\max}}}$$

that relates the difference of the objective values in the solutions visited with the distance between these solutions along the walk

Autocorrelation coefficient (2)

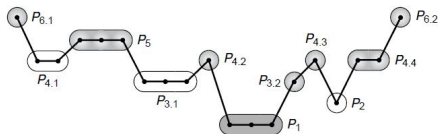
$$r(i) = \frac{\sum_{t=1}^{t_{\max}-i} (f^{(t)} - \bar{f})(f^{(t+i)} - \bar{f})}{\frac{\sum_{t=1}^{t_{\max}} (f^{(t)} - \bar{f})^2}{t_{\max}}}$$

- $r(0) = 1$ (*perfect correlation at 0 distance*)
- in general $r(i)$ decreases as the distance i increases
- 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 *steepest descent* heuristic is ineffective

Plateau

The search graph can be partitioned according to the objective value

- **plateau of value f** is each subset of solutions of value f that are adjacent in the search graph



Large plateaus complicate the choice of the solution: most neighbours are equivalent, and the choice ends up depending on the visit order

An extremely uniform landscape is not an advantage!

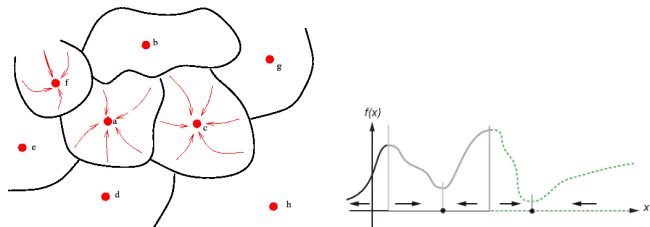
Example: all transfers and swaps between machines 1 and 3 leave the objective value unchanged (*most other moves worsen it*)



Attraction basins

Alternatively, the search graph can be partitioned into:

- **attraction basins** of the locally optimal solutions \bar{x} , that are the subsets of solutions $x^{(0)} \in X$ starting from which the *steepest descent* heuristic terminates in \bar{x}



The *steepest descent* heuristic is

- **effective** if the attraction basins are few and large (especially if the global optima have larger basins)
- **ineffective** if the attraction basins are many and small (especially if the global optima have smaller basins)

Algorithm SteepestDescent ($I, x^{(0)}$)

$x := x^{(0)}$;

Stop := *false*;

While Stop = *false* do { t_{\max} iterations }

$\tilde{x} := \arg \min_{x' \in N(x)} f(x)$;

If $f(\tilde{x}) \geq f(x)$ then Stop := *true*; else $x := \tilde{x}$;

EndWhile;

Return ($x, f(x)$);

The complexity of the *steepest descent* heuristic depends on

- 1 the number of iterations t_{\max} from $x^{(0)}$ to the local optimum found, which depends on the structure of the search graph (width of the attraction basins) and is hard to estimate *a priori*
- 2 the search for the best solution in the neighbourhood (\tilde{x}), which depends on how the search itself is performed, but usually only requires a standard complexity estimation

The exploration of the neighbourhood

Two strategies to explore the neighbourhood are possible

- 1 **exhaustive search**: evaluate all the neighbour solutions;
the complexity of a single step is the product of
 - the number of neighbour solutions ($|N(x)|$)
 - the evaluation of the cost of each solution ($\gamma_f(|B|, x)$)

If it is not possible to generate only feasible solution:

- visit a superset of the neighbourhood ($\tilde{N}(x) \supset N(x)$)
 - for each element x , evaluate the feasibility ($\gamma_x(|B|, x)$)
 - for the feasible ones, evaluate the cost ($\gamma_f(|B|, x)$)
- 2 **efficient exploration of the neighbourhood** without a complete visit:
find the best neighbour solution solving an auxiliary problem

Only some special neighbourhoods allow that

Exhaustive visit of the neighbourhood

Algorithm SteepestDescent($I, x^{(0)}$)

$x := x^{(0)}$;

Stop := false;

While Stop = false do

$\tilde{x} := x$;

$$\{ \tilde{x} := \arg \min_{x' \in N(x)} f(x') \}$$

For each $x' \in \tilde{N}(x)$ do

If $x' \in N(x)$ then

If $f(x') < f(\tilde{x})$ then $\tilde{x} := x'$;

EndIf;

EndFor;

If $f(\tilde{x}) \geq f(x)$ then Stop := true; else $x := \tilde{x}$;

EndWhile;

Return ($x, f(x)$);

The complexity of the neighbourhood exploration combines three terms

- 1 $|\tilde{N}(x)|$: the number of subsets visited
- 2 γ_X : the time to evaluate their feasibility
- 3 γ_f : the time to evaluate the objective for a feasible solution

Evaluating or updating the objective: the additive case

The first way to accelerate an exchange algorithm is to **minimize the time to evaluate the objective**: in particular, it is faster to **update $f(x)$ rather than to recompute it**

The update of an additive objective $f(x) = \sum_{j \in x} \phi_j$ requires to

- sum ϕ_i for each element $i \in A$, added to x
- subtract ϕ_j for each element $j \in D$, deleted from x

$$\delta f(x, A, D) = f(x \cup A \setminus D) - f(x) = \sum_{i \in A} \phi_i - \sum_{j \in D} \phi_j$$

Examples: swap of objects (*KP*), columns (*SCP*), edges (*CMSTP*), ...

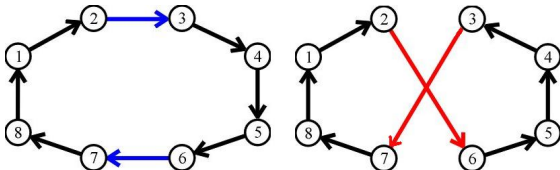
This update has two fundamental properties:

- it takes **constant time for a constant number of elements $|A| + |D|$**
- **$\delta f(x, A, D)$ does not depend on x** (we will talk about it later)

Example: the symmetric TSP

To generate neighbourhood $N_{\mathcal{R}_2}$ for the TSP we

- delete two nonconsecutive arcs (s_i, s_{i+1}) and (s_j, s_{j+1})
- add the two arcs (s_i, s_j) and (s_{i+1}, s_{j+1})
- revert the path (s_{i+1}, \dots, s_j) (modifying $O(n)$ arcs!)



If the graph and the cost function are symmetric, the variation of $f(x)$ is

$$\delta f(x, A, D) = c_{s_i, s_j} + c_{s_{i+1}, s_{j+1}} - c_{s_i, s_{i+1}} - c_{s_j, s_{j+1}}$$

but this is not true for the asymmetric TSP

What if the objective function is not additive?

Evaluating or updating the objective: the quadratic case

The *MDP* has a quadratic objective function: computing it costs $\Theta(n^2)$

Moving from x to $x' = x \setminus \{i\} \cup \{j\}$ (neighbourhood N_{S_1}), the update is

$$\delta f(x, i, j) = f(x \setminus \{i\} \cup \{j\}) - f(x) = \sum_{h, k \in x \setminus \{i\} \cup \{j\}} d_{hk} - \sum_{h, k \in x} d_{hk}$$

which depends on $O(n)$ distance terms, related to points i and j

There is a general trick for the symmetric quadratic functions with $d_{ii} = 0$

$$\begin{aligned} \delta f(x, i, j) &= \sum_{h \in x \setminus \{i\} \cup \{j\}} \sum_{k \in x \setminus \{i\} \cup \{j\}} d_{hk} - \sum_{h \in x} \sum_{k \in x} d_{hk} \Rightarrow \\ \Rightarrow \delta f(x, i, j) &= 2 \sum_{k \in x} d_{jk} - 2 \sum_{k \in x} d_{ik} - 2d_{ij} = 2(D_j(x) - D_i(x) - d_{ij}) \end{aligned}$$

If $D_\ell(x) = \sum_{k \in x} d_{\ell k}$ is known for each $\ell \in B$, the computation takes $O(1)$

Example: the MDP

Let us consider $f(x)/2$

Evaluate the exchange

$$x \rightarrow x' = x \setminus \{i\} \cup \{j\}$$

with $i \in x$ and $j \in B \setminus x$


$$f(x') = f(x) - D_i + D_j - d_{ij}$$

- the pairs including i are lost
- the pairs including j are acquired
- but the pair (i, j) is in excess

The cost is computed in $O(1)$ time for each solution

x $B \setminus x$



i   j



Example: the MDP

Let us consider $f(x)/2$

Evaluate the exchange

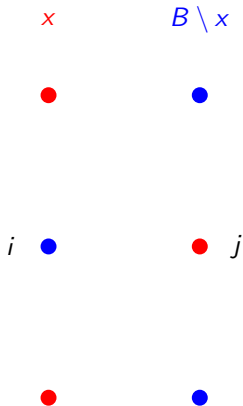
$$x \rightarrow x' = x \setminus \{i\} \cup \{j\}$$

with $i \in x$ and $j \in B \setminus x$

$$f(x') = f(x) - D_i + D_j - d_{ij}$$

- the pairs including i are lost
- the pairs including j are acquired
- but the pair (i, j) is in excess

The cost is computed in $O(1)$ time for each solution



Example: the MDP

Let us consider $f(x)/2$

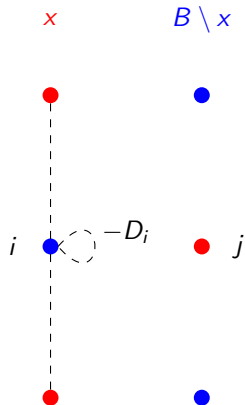
Evaluate the exchange

$$x \rightarrow x' = x \setminus \{i\} \cup \{j\}$$

with $i \in x$ and $j \in B \setminus x$

$$f(x') = f(x) - D_i + D_j - d_{ij}$$

- the pairs including i are lost
- the pairs including j are acquired
- but the pair (i, j) is in excess



The cost is computed in $O(1)$ time for each solution

Example: the MDP

Let us consider $f(x)/2$

Evaluate the exchange

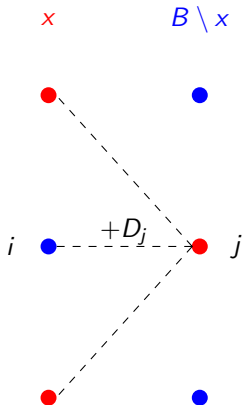
$$x \rightarrow x' = x \setminus \{i\} \cup \{j\}$$

with $i \in x$ and $j \in B \setminus x$

$$f(x') = f(x) - D_i + D_j - d_{ij}$$

- the pairs including i are lost
- the pairs including j are acquired
- but the pair (i, j) is in excess

The cost is computed in $O(1)$ time for each solution



Example: the MDP

Let us consider $f(x)/2$

Evaluate the exchange

$$x \rightarrow x' = x \setminus \{i\} \cup \{j\}$$

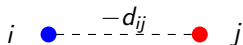
with $i \in x$ and $j \in B \setminus x$

$$f(x') = f(x) - D_i + D_j - d_{ij}$$

- the pairs including i are lost
- the pairs including j are acquired
- **but the pair (i, j) is in excess**

The cost is computed in $O(1)$ time for each solution

x $B \setminus x$



Example: the MDP

x $B \setminus x$

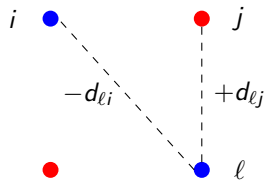


Update of the data structures:

- $D_\ell = D_\ell - d_{\ell i} + d_{\ell j}$, $\ell \in B$

For each element $\ell \in B$

- $d_{\ell i}$ disappears
- $d_{\ell j}$ appears



The auxiliary data structure is updated in $O(n)$ time for each iteration

Example: the MDP

x $B \setminus x$

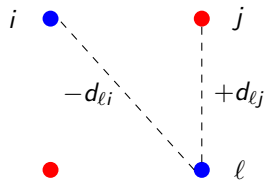


Update of the data structures:

- $D_\ell = D_\ell - d_{\ell i} + d_{\ell j}, \ell \in B$

For each element $\ell \in B$

- $d_{\ell i}$ disappears
- $d_{\ell j}$ appears



The auxiliary data structure is updated in $O(n)$ time for each iteration

Example: the *MDP*

x $B \setminus x$

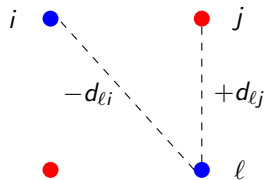


Update of the data structures:

- $D_\ell = D_\ell - d_{\ell i} + d_{\ell j}$, $\ell \in B$

For each element $\ell \in B$

- $d_{\ell i}$ disappears
- $d_{\ell j}$ appears



The auxiliary data structure is updated in $O(n)$ time for each iteration

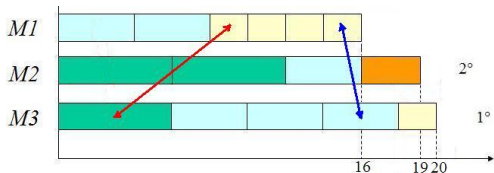
Updating the objective function: nonlinear examples

Many nonlinear functions can be updated with similar tricks

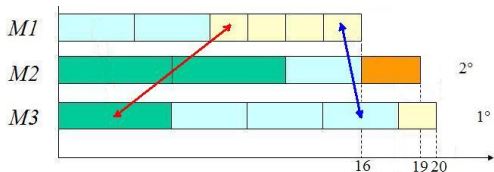
- save aggregated information on the current solution $x^{(t)}$
- use it to compute $f(x')$ efficiently for each $x' \in N(x^{(t)})$
- update it when moving to the following solution $x^{(t+1)}$

Using the transfer (N_{T_1}) and swap (N_{S_1}) neighbourhoods for the *PMSP*, the objective can be updated in constant time by managing

- 1 the completion time for each machine
- 2 the indices of the machines with the first and second maximum time



Example: the *PMSP*



Consider the exchange $\sigma = (i, j)$ of tasks i and j
(i on machine M_i , j on machine M_j)

- compute in constant time the new completion times:
one increases, the other decreases (or both remain constant)
- test in constant time whether either exceeds the maximum
- if the maximum time decreases, test in constant time whether the other time or the second maximum time becomes the maximum

Once the neighbourhood is visited and the exchange selected, update

- the two modified completion times (each one in constant time)
- their positions in a max-heap (each one in time $O(\log |M|)$)

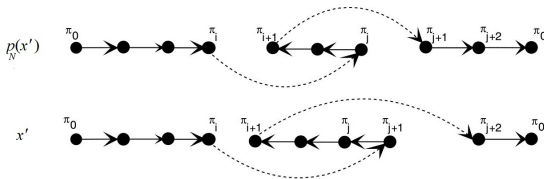
Use of local auxiliary information

The auxiliary information used to compute $f(x')$ can be

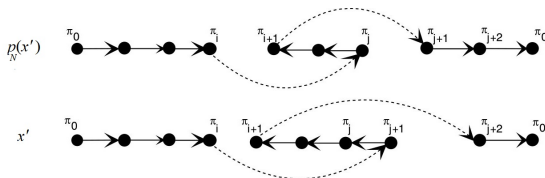
- global, that is referring to the current solution x
- local, that is referring to the solution $p_N(x')$ visited before x' in neighbourhood $N(x)$ according to a suitable order

Consider the neighbourhood $N_{\mathcal{R}_2}$ for the asymmetric TSP:

- the neighbour solutions differ from x for $O(n)$ arcs
- general neighbour solutions differ from each other for $O(n)$ arcs
- if the pairs of arcs (s_i, s_{i+1}) and (s_j, s_{j+1}) follow the lexicographic order, the reverted path changes only by one arc



Example: the asymmetric TSP



The variation of $f(x)$ between two generic neighbour solutions is

$$\delta f(x, i, j) = c_{s_i, s_j} + c_{s_{i+1}, s_{j+1}} - c_{s_i, s_{i+1}} - c_{s_j, s_{j+1}} + c_{s_j \dots s_{i+1}} - c_{s_{i+1} \dots s_j}$$

but moving from exchange (s_i, s_j) to exchange (s_i, s_{j+1})

- the first four terms change, but they can be checked in constant time
- the last two terms can be updated in constant time

$$\begin{cases} c_{s_j \dots s_{i+1}} = c_{s_j \dots s_{i+1}} + c_{s_{j+1}, s_j} \\ c_{s_{i+1} \dots s_j} = c_{s_{i+1} \dots s_j} + c_{s_j, s_{j+1}} \end{cases}$$

Is it acceptable to explore the neighbourhood in a predefined order ?

What about feasibility?

Defining neighbourhoods with the Hamming distance or with operations can generate also unfeasible subsets, that must be removed

$$\tilde{N}_{H_k}(x) = \{x' \subseteq B : d(x', x) \leq k\} \supseteq N_{H_k}(x) = \tilde{N}_{H_k}(x) \cap X$$

$$\tilde{N}_{\mathcal{O}}(x) = \{x' \subseteq B : \exists o \in \mathcal{O} : o(x) = x'\} \supseteq N_{\mathcal{O}}(x) = \tilde{N}_{\mathcal{O}}(x) \cap X$$

(Examples: KP, BPP, SCP, CMSTP...)

If it is not possible to avoid *a priori* the unfeasible subsets, one must

- test the **feasibility** of each element of $\tilde{N}(x)$ to obtain $N(x)$
- for the feasible elements, evaluate the **cost**

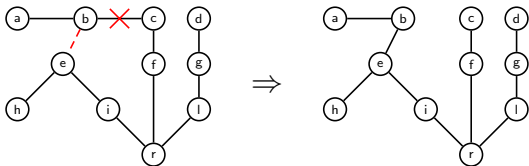
The feasibility test can be made efficient with techniques similar to the ones used for the objective evaluation

Example: update in constant time the total volume of a subset in the KP

Example: the *CMSTP*

Consider the swap neighbourhood N_{S_1} (add one edge, delete another)

- if the two edges are in the same branch, the solution remains feasible
- if they are in different branches, one loses weight, the other acquires it: **the variation is equal to the weight of the subtree transferred**



If each vertex saves the weight of its appended subtree, to test feasibility compare this weight with the residual capacity of the receiving branch (the weight appended to *b* with the residual capacity of the left branch)

Once the best exchange is performed, the information must be updated in time $O(n)$ visiting the ancestors of the receiving vertex *e*

A general scheme of sophisticated exploration

The use of auxiliary information requires

- 1 the **inicialization** of suitable data structures
 - partly **local**, i. e., related to neighbour solutions
 - partly **global**, i. e., related to the current solution
- 2 their **update** between subsequent solutions or iterations

Algorithm SteepestDescent($I, x^{(0)}$)

$x := x^{(0)}$; **GI** := InitializeGI(); Stop := false;

While Stop = false *do*

$\tilde{x} := 0$; $\tilde{\delta} := 0$; **LI** := InitializeLI()

For each $x' \in N(x)$ *do*

If $f(x') < f(\tilde{x})$ *then* $\tilde{x} := x'$;

LI := UpdateLI(LI, x')

EndFor;

If $f(\tilde{x}) \geq f(x)$

then Stop := true;

else $x := \tilde{x}$; **GI** := UpdateGI(GI, \tilde{x})

EndIf

EndWhile;

Return ($x, f(x)$);

Partial saving of the neighbourhood (1)

When performing an operation $o \in \mathcal{O}$ on a solution $x \in X$ the variation

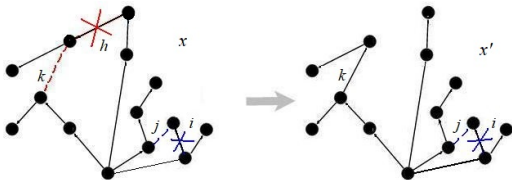
$$\delta f_o(x) = f(o(x)) - f(x)$$

sometimes depends only on a part of x (possibly, very small)

For example, consider the swap neighbourhood N_{S_1} for the *CMST*:

- add an edge $k \in B \setminus x$
- delete an edge $h \in x$

Two branches are involved: one acquires a subtree, the other loses it



The effect of swap (i, j) depends only on the branches including i and j : it is the same in x and x' and is not affected by swap (h, k)

$$\delta f_{i,j}(x) = \delta f_{i,j}(x')$$

Partial saving of the neighbourhood (2)

There exists a subset of operations $\tilde{\mathcal{O}} \subset \mathcal{O}$ such that

$$\delta f_o(x') = \delta f_o(x) \text{ for each } o \in \tilde{\mathcal{O}} \text{ and for each } x' = o(x)$$

It is then advantageous to

- 1 **compute and save $\delta f_o(x)$ for every $o \in \mathcal{O}$** , that is keep the set of feasible exchanges and their associated values δf
- 2 perform the best operation o^* , and generate a new solution x'
- 3 **recompute and save $\delta f_o(x')$ only for $o \in \mathcal{O} \setminus \tilde{\mathcal{O}}$** , that is remove the exchanges on modified branches, recompute their values, and **retrieve $\delta f_o(x')$ for all $o \in \tilde{\mathcal{O}}$** (*their values are still correct*)
- 4 go back to point 2

If the branches are numerous, $|\mathcal{O} \setminus \tilde{\mathcal{O}}| \ll |\mathcal{O}|$ and the saving is very strong

It is typical of problems whose solution is a partition

Trade-off between efficiency and effectiveness

The complexity of an exchange heuristic depends on three factors

- ① number of iterations
- ② cardinality of the visited neighbourhood
- ③ computation of the feasibility and cost for the single neighbour

The first two factors are clearly conflicting:

- a small neighbourhood is fast to explore, but requires several steps to reach a local optimum
- a large neighbourhood requires few steps, but is slow to explore

The optimal trade-off is somewhere in the middle: a neighbourhood

- large enough to include good solutions
- small enough to be explored quickly

but it is hard to identify, because

- efficiency quickly worsens as size increases
- the resulting solution also changes with the neighbourhood (large neighbourhoods have better local optima)

Fine tuning of the neighbourhoods

It is also possible to define a neighbourhood N and tune its size

- explore only a promising subneighbourhood $N' \subset N$

For example, if the objective function is additive, one can

- add only elements $j \in B \setminus x$ of low cost ϕ_j
 - delete only elements $i \in x$ of high cost ϕ_i
- terminate the visit after finding a promising solution
For example, the **first-best strategy** stops the exploration at the first solution better than the current one

If $f(\tilde{x}) < f(x)$ then $x := \tilde{x}$; Stop := true;

The effectiveness depends on the objective

- if the cost of some elements influences very much the objective, it is worth taking it into account, fixing or forbidding them

and on the structure of the neighbourhood

- if the landscape is smooth, the first improving solution approximates well the best solution of the neighbourhood: it is better to stop
- if the landscape is rugged, the best solution of the neighbourhood could be much better: it is better to go on

Very Large Scale Neighbourhood Search

Larger neighbourhoods yield in general larger attraction basins, so that

- the *steepest descent* heuristic becomes very effective
- but the exploration time is longer

The *Very Large Scale Neighbourhood (VLSN) Search* approaches have

- neighbourhoods exponential in $|B|$ (or high-order polynomial)
- explored in low-order polynomial time

Two strategies allow to limit the computational time

- 1 select a neighbourhood in which the objective can be optimized without an exhaustive exploration
- 2 explore the neighbourhood heuristically and return a promising neighbour solution, instead of the best one

Efficient visit of exponential neighbourhoods

The neighbourhoods based on operations can be easily parameterised

$$N_{\mathcal{O}_k}(x) = \{x' \in X : x' = o_k(o_{k-1}(\dots o_1(x))) \text{ with } o_1, \dots, o_k \in \mathcal{O}\}$$

and it would be nice to tune the number of operations k

- increasing k when necessary to improve the current solution x
- decreasing k when sufficient to improve the current solution x

The idea is to define a **composite move** as a **set of elementary moves**
(*that is a combinatorial optimisation problem!*)

Finding the optimal solution in such neighbourhoods requires to solve an auxiliary problem, typically on a matrix or graph

- **set packing**: *Dynasearch*
- **negative cost circuit**: cyclic exchanges
- **shortest path**: *ejection chains, order-and-split*

Such auxiliary tools are usually defined **improvement matrices** or **graphs**

Combining elementary moves into composite ones

An operation $o \in \mathcal{O}$ usually modifies only some components of solution x

Often **only the modified components of x determine**

- the **feasibility** of the new subset $o(x)$
- the **variation of the objective function** $\delta f_o(x) = f(o(x)) - f(x)$

Then, **two operations $o, o' \in \mathcal{O}$ that modify different components of x**

- are **compatible and commutable**

$$o'(o(x)) = o(o'(x)) \in X$$

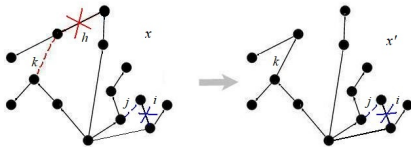
- **have an overall effect independent from the order of application and easy to compute: for additive functions it is usually the sum**

$$\delta f_{oo'}(x) = \delta f_{o'o}(x) = \delta f_o(x) + \delta f_{o'}(x)$$

The idea is to **perform a whole set of moves combining their effects**

Examples of move combinations

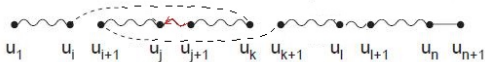
- *CMSTP*: transfer or swap vertices between different subtrees
(moves on overlapping subtrees could be infeasible)



- *VRP*: transfer or swap nodes between different circuits
(moves on overlapping circuits could be infeasible)
- *TSP*: 2-opt exchanges operating on disjoint segments of the circuit
(arcs that define an exchange are removed/reversed in the others)
 - moves (i, j) and (k, l) are compatible and can be applied in any order



- moves (i, k) and (j, l) are incompatible, as (i, k) reverses $(u_j, u_{j+1})!$



Let a **composite move** be a **set of elementary moves** with mutually independent effects on feasibility and the objective

The situation can be modelled with an **improvement matrix A** in which

- the rows represent the components of the solution (e.g., branches in the *CMSTP*, circuits in the *VRP*, circuit segments in the *TSP*)
- the columns represent the elementary moves $o \in \mathcal{O}$: each one has a value equal to the objective improvement $-\delta f_o(x)$
- if move o impacts on component i , $a_{io} = 1$; otherwise $a_{io} = 0$

Determine an **optimum packing of the columns**, that is a **subset of nonconflicting columns of maximum value**

The *Set Packing Problem* is in general \mathcal{NP} -hard, but

- on special matrices it is polynomial (as in the matrix from the *TSP*)
- if each move modifies at most two components
 - the rows can be seen as vertices
 - the columns can be seen as edges
 - each packing of columns becomes a matching

and the maximum matching problem is polynomial

Cyclic exchanges

In many problems

- a feasible solution is a partition of objects into components $S^{(\ell)}$, that is an assignment of objects to components (i, S_i) (*vertices or edges into branches for the CMSTP, nodes or arcs into circuits for the VRP, objects into containers in the BPP, etc. . .*)
- the feasibility is associated to the single components
- the objective function is additive with respect to the components

$$f(x) = \sum_{\ell=1}^r f(S^{(\ell)})$$

In these problems, it is natural to define the set of operations \mathcal{T}_k which includes the **transfers of k elements from their component to another** and to derive from \mathcal{T}_k the neighbourhood $N_{\mathcal{T}_k}$

- often the feasibility constraints forbid the simple transfers
- but the number of multiple transfers quickly grows with k

We want to find a **subset of $N_{\mathcal{T}_k}$ large, but efficient to explore**

The improvement graph

The **improvement graph** allows to describe sequences of transfers

- a **node** i corresponds to an **element** i of the current solution x
- an **arc** (i, j) corresponds to
 - the **transfer** of element i from its current component S_i to the current component S_j of element j
 - the **deletion** of element j from component S_j
- the **cost of arc** c_{ij} corresponds to the (positive or negative) **variation of the contribution of S_j to the objective**

$$c_{ij} = f(S_j \cup \{i\} \setminus \{j\}) - f(S_j)$$

with $c_{ij} = +\infty$ if it is **unfeasible to replace j with i in S_j**

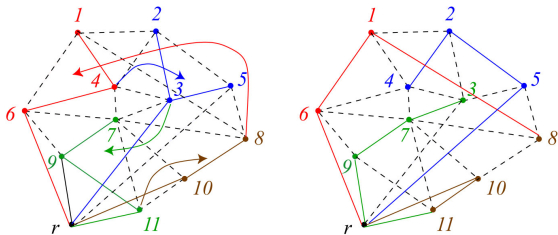
A circuit in such a graph corresponds to a closed sequence of transfers

The cost of the circuit corresponds to the cost of the sequence

- **but only if each node belongs to a different component**

Find the minimum cost circuit satisfying this condition

Example: the *CMSTP*



(4, *Rosso*, *Blu*) (3, *Blu*, *Verde*) (11, *Verde*, *Marrone*) (8, *Marrone*, *Rosso*)

Consider the composite move (4, 3), (3, 11), (11, 8), (8, 4):

- vertex 4 moves into the blue branch to replace vertex 3
- vertex 3 moves into the green branch to replace vertex 11
- vertex 11 moves into the brown branch to replace vertex 8
- vertex 8 moves into the red branch to replace vertex 4

The cost variation for subtree S_j yields the cost of arc c_{ij}

The weight of branch S_j varies by $w_i - w_j$: if unfeasible, forbid the arc

Search for the minimum cost circuit (1)

The problem is actually \mathcal{NP} -hard, but

- the constraint of visiting only once each component allows a rather efficient **dynamic programming algorithm** that grows partial paths
(if the components are r , the circuit has at most r arcs)
- **all partial paths of cost ≥ 0 can be neglected** because
 - the total variation of the objective sums the effect of the single moves

$$\delta f_{o_1, \dots, o_k}(x) = \sum_{\ell=1}^k \delta f_{o_\ell}(x)$$

- every sequence of numbers with negative sum admits a cyclic permutation whose partial sums are all negative
E. g., $(+1, -2, +4, -10, +2)$ admits $(-10, +2, +1, -2, +4)$
- therefore, **there is a cyclic permutation of the moves** o_1, \dots, o_k

$$\delta f_{o_1, \dots, o_k}(x) < 0 \Rightarrow \exists h : \delta f_{o_{(1+h) \pmod k}, \dots, o_{(k+h) \pmod k}}(x) < 0$$

that is, **improving at each step**

Search for the minimum cost circuit (2)

Moreover,

- there are heuristic polynomial algorithms for the problem
- there are polynomial algorithms to solve relaxations of the problem that neglect the constraint on the components, finding
 - a nonminimum negative circuit (Floyd-Warshall), if any exists
 - a circuit of minimum average cost (total cost / number of arcs)

If the cost of such relaxed solutions is

- positive, then no negative circuit exists
- negative, then the relaxed solution can be
 - optimal (if luckily they are feasible)
 - a starting point to generate a feasible heuristic solution

Noncyclic exchange chains

It is also possible to create **noncyclic transfer chains**, so that the cardinality of the components can vary

It is enough to **add to the improvement graph**

- a source node
- a node for each component
- arcs from the source node to the nodes associated to the elements
- arcs from the nodes associated to the elements to the nodes associated to the components

Then, find the **minimum cost path** that

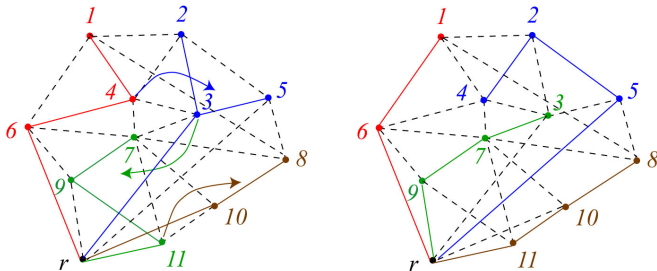
- starts from the source node
- ends in a component node (guaranteed by the topology)
- never visits two nodes associated to the same component

These paths correspond to open transfer chains in which

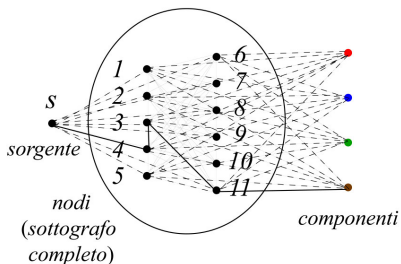
- a component loses an element
- zero or more components lose an element and acquire another one
- a component acquires an element

Example: the *CMSTP*

Noncyclic exchange $(s, 4)$, $(4, 3)$, $(3, 11)$, $(11, S_4)$



$(4, \text{Rosso}, \text{Blu})$ $(3, \text{Blu}, \text{Verde})$ $(11, \text{Verde}, \text{Marrone})$



Order-first split-second

The *Order-first split-second* method for partition problems

- builds a starting permutation of the elements
- partitions the elements into components in an optimum way under the additional constraint that elements of the same component be consecutive in the starting permutation

Of course, the solution depends on the starting permutation:
it is reasonable to repeat the resolution for different permutations
creating a two-level method

- 1 the upper level selects a permutation
- 2 the lower level computes the optimal partition for the permutation

*Problem: different permutations yield the same solution
(the permutations are more numerous than the solutions)*

The auxiliary graph

Once again, we exploit an auxiliary graph

Given the permutation (s_1, \dots, s_n) of the elements

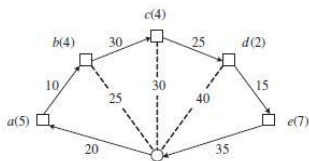
- each node s_i corresponds to a element s_i
plus a fictitious node s_0
- each arc (s_i, s_j) with $i < j$ corresponds to a potential component $S_\ell = (s_{i+1}, \dots, s_j)$ formed by the elements of the permutation
 - from s_i excluded
 - to s_j included
- the cost c_{s_i, s_j} corresponds to the cost of the component $f(S_\ell)$
- the arc does not exist if the component is unfeasible

Consequently

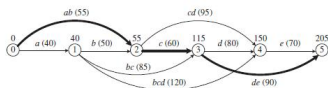
- each path from s_0 a s_n represents a solution (partition of elements)
- the cost of the path coincides with the cost of the partition
- the graph is acyclic: finding the optimum path costs $O(m)$ where $m \leq n(n-1)/2$ is the number of arcs

Example: the VRP

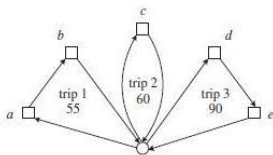
Given an instance of *VRP* with 5 nodes and capacity $W = 10$



the arcs corresponding to unfeasible paths (weight $> W$) do not exist, the costs of the arcs are the costs of the *TSP* solutions for $\{d, s_{i+1}, \dots, s_j\}$



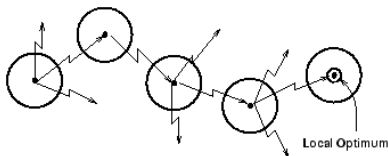
The optimal path corresponds to three circuits: (d, s_1, s_2, d) , (d, s_3, d) and (d, s_4, s_5, d)



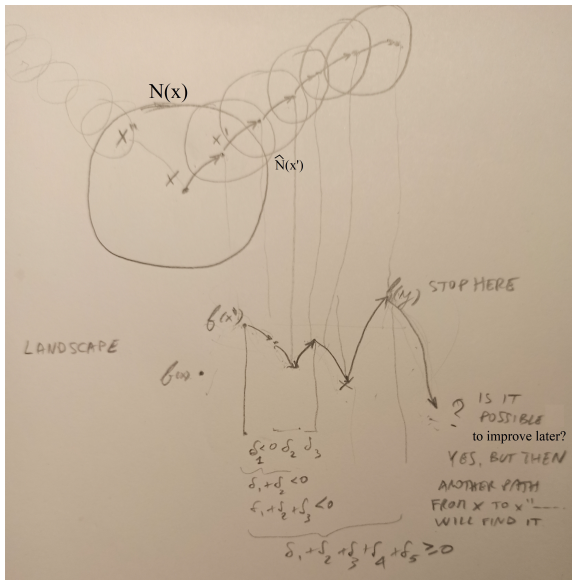
Variable Depth Search (VDS)

In the VDS a **composite move** as **sequence of elementary moves**

- consider each solution x' in the basic neighbourhood $N_{O_1}(x)$
- from it, make a sequence of moves **optimizing each elementary step**, but **allowing worsening moves** and **forbidding backward moves**
- **terminate** when the current solution y becomes worse than x' or all moves are forbidden (**the length k of the sequence is variable**)
- **return the best solution y^* found along the sequence**



Variable Depth Search



Scheme of the Variable Depth Search

Given $x^{(t)}$, for each $x' \in N(x^{(t)})$, instead of evaluating only $f(x')$

- 1 find a promising solution \tilde{y} in a neighbourhood $\hat{N}(x') \subseteq N(x')$
- 2 as long as \tilde{y} improves $x^{(t)}$, replace x' with \tilde{y} and go to 1
- 3 return the best solution y^* found during the whole process

For each $x' \in N(x)$

{ Steepest descent }

Compute $f(x')$

{ Variable Depth Search }

$y := x'$; $y^* := x'$; Stop := false;

While Stop = false do

$\tilde{y} := \arg \min_{y' \in \hat{N}(y)} f(y')$;

If $f(\tilde{y}) \geq f(x')$ then Stop := true; else $y := \tilde{y}$;

If $f(\tilde{y}) < f(y^*)$ then $y^* := \tilde{y}$;

EndWhile;

Compute $f(y^*)$;

It is a sort of roll-out mechanism for exchange algorithms

Differences with respect to *steepest descent*

With respect to *steepest descent* exploration

- *VDS* finds a local optimum for each solution of the neighbourhood performing a sort of one-step *look-ahead*
- *VDS* admits worsenings along the sequence of elementary moves (but never with respect to the starting solution)
- *VDS* makes moves that increase the distance from the starting point to avoid cyclic behaviours (*gradually restricting the neighbourhood*)

In order to limit the computational effort

- the elementary moves use a reduced neighbourhood $\hat{N} \subseteq N$
- \hat{N} (elementary step) is explored with the *first-best strategy*
- N (basic neighbourhood) is explored with the *first-best strategy*

Lin-Kernighan's algorithm for the symmetric TSP

Neighbourhood $N_{\mathcal{R}_k}(x)$ includes the solutions obtained

- deleting k arcs of x
- adding other k arcs that recreate a Hamiltonian circuit
- possibly inverting parts of the circuit *(leaving the cost unchanged)*

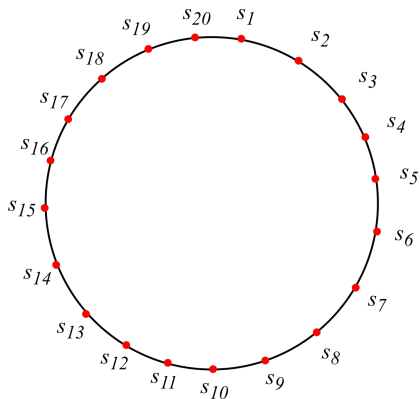
Lin-Kernighan's algorithm is a VDS with sequences of 2-opt exchanges:
a k -opt exchange is equivalent to a sequence of $(k - 1)$ 2-opt exchanges,
where each deletes one of the two arcs added by the previous exchange

Then for each solution $x' \in N_{\mathcal{R}_2}(x)$ obtained by exchange (i, j)

- evaluate the 2-opt exchanges that delete the added arc (s_i, s_{j+1}) and each arc of $x \cap x'$ to find the best exchange (i', j')
- if this improves upon x , perform exchange (i', j') , obtaining x''
- evaluate the exchanges that delete $(s_{i'}, s_{j'+1})$ and each arc of $x \cap x'' \dots$
- ...
- if the best solution among x', x'', \dots is better than x , accept it

Example: Lin-Kernighan's algorithm

Explore all the solutions $x' \in N_{\mathcal{R}_2}(x)$, obtained with exchanges (i, j)

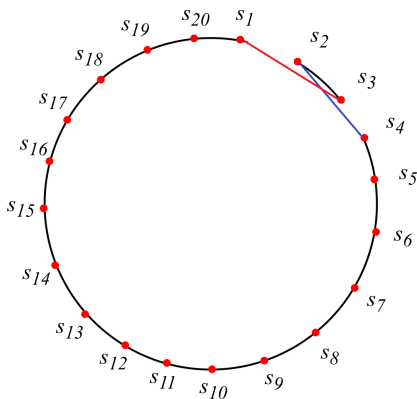


$$x = (s_1 \overline{s_2 s_3} s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20})$$

Let us focus on the exchange $(1, 3)$, that reverts (s_2, \dots, s_3)

Example: Lin-Kernighan's algorithm

The exchange (1, 3) replaces (s_1, s_2) and (s_3, s_4) with (s_1, s_3) and (s_2, s_4)

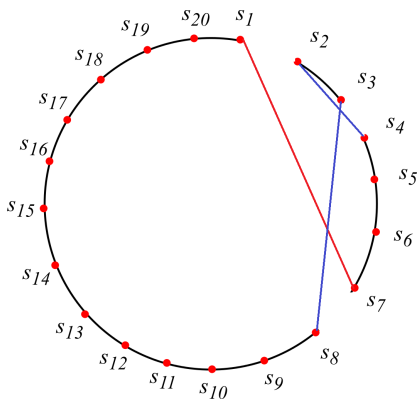


$$x' = (s_1 \overline{s_3 s_2 s_4 s_5 s_6 s_7} s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20})$$

Search for the best exchange that removes (s_1, s_3) and an arc of $x \cap x'$
Let us suppose that it is (1, 7), which reverts (s_3, \dots, s_7)

Example: Lin-Kernighan's algorithm

The exchange (1, 7) replaces (s_1, s_3) and (s_7, s_8) with (s_1, s_7) and (s_3, s_8)

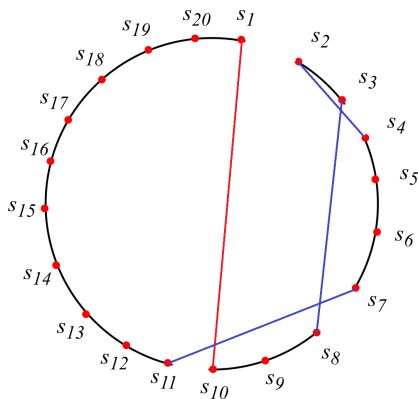


$$x'' = (s_1 \overline{s_7 s_6 s_5 s_4 s_2 s_3 s_8 s_9 s_{10}} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20})$$

Search for the best exchange that removes (s_1, s_7) and an arc of $x \cap x''$
Let us suppose that it is (1, 10), which reverts (s_7, \dots, s_{10})

Example: Lin-Kernighan's algorithm

The exchange (1, 10) replaces (s_1, s_7) and (s_{10}, s_{11}) con (s_1, s_{10}) and (s_7, s_{11})

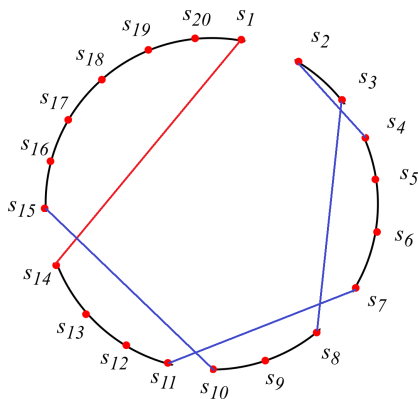


$$x''' = (s_1 \overline{s_{10} s_9 s_8 s_3 s_2 s_4 s_5 s_6 s_7 s_{11} s_{12} s_{13} s_{14}} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20})$$

Search for the best exchange that removes (s_1, s_{10}) and an arc of $x \cap x'''$
Let us suppose that it is (1, 14), which reverts (s_{10}, \dots, s_{14})

Example: Lin-Kernighan's algorithm

The exchange (1, 14) replaces (s_1, s_{10}) and (s_{14}, s_{15}) con (s_1, s_{14}) and (s_{10}, s_{15})

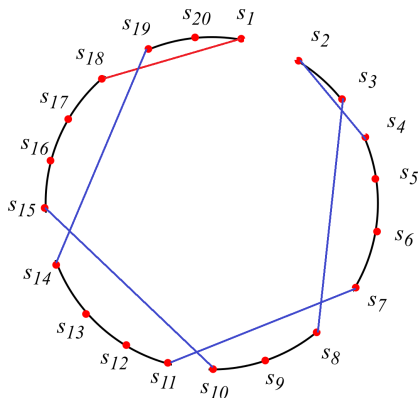


$$x^{iv} = (s_1 \overline{s_{14} s_{13} s_{12} s_{11} s_7 s_6 s_5 s_4 s_2 s_3 s_8 s_9 s_{10} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20}})$$

Search for the best exchange that removes (s_1, s_{14}) and an arc of $x \cap x^{iv}$
Let us suppose that it is (1, 18), which reverts (s_{14}, \dots, s_{18})

Example: Lin-Kernighan's algorithm

The exchange (1, 18) replaces (s_1, s_{14}) and (s_{18}, s_{19}) con (s_1, s_{18}) and (s_{14}, s_{19})



$$x^V = (s_1 s_{18} s_{17} s_{16} s_{15} s_{10} s_9 s_8 s_3 s_2 s_4 s_5 s_6 s_7 s_{11} s_{12} s_{13} s_{14} s_{19} s_{20})$$

Search for the best exchange that removes (s_1, s_{18}) and an arc of $x \cap x^V$
Let us suppose that all exchanges yield solutions worst than x :
terminate, returning the best solution found

Implementation details

- each step deletes an arc of the starting solution to avoid going back and one of the arcs added in the previous step to reduce complexity
- this imposes an upper bound on the length of the sequence
- stopping the sequence as soon as the solution is no longer better than the starting solution does not impair the result
 - the total variation of the objective sums the effect of the single moves

$$\delta f_{o_1, \dots, o_k}(x) = \sum_{\ell=1}^k \delta f_{o_\ell}(x)$$

- every sequence of numbers with negative sum admits a cyclic permutation whose partial sums are all negative
- therefore, there is a cyclic permutation of the moves o_1, \dots, o_k

$$\delta f_{o_1, \dots, o_k}(x) < 0 \Rightarrow \exists h : \delta f_{o_{(1+h) \pmod k}, \dots, o_{(k+h) \pmod k}}(x) < 0$$

that is, improving at each step

Iterated greedy methods (*destroy-and-repair*)

Every exchange can be seen as a combination of addition and deletion

$$x' = x \cup A \setminus D$$

with $A = x' \setminus x$ and $D = x \setminus x'$

However

- single swaps $x' = x \cup \{j\} \setminus \{i\}$ can give bad or unfeasible results
- larger neighbourhoods can be inefficient
- in many problems the right cardinalities of A and D are unknown, because the solutions have nonuniform cardinality (e.g., *KP*, *SCP*...)

A possible idea is to

- 1 delete from x a subset $D \subset x$ of cardinality $\leq k$ (destroy heuristic)
- 2 complete it with a constructive heuristic (repair heuristic)

or, of course, the opposite

- 1 add to x a set $A \subset B \setminus x$ of cardinality $\leq k$
- 2 reduce it with a destructive heuristic

Selection of A and D

Most of the time both subsets are chosen heuristically, not exhaustively

- tuning their size $|A|$ and $|D|$ with some parameter
- selecting promising elements based on their cost/value
- applying the first-best strategy
(immediately accept any improving solution)

Usually both subsets are chosen in a randomised way

In this case, they are metaheuristics