# Heuristic Algorithms

## Master's Degree in Computer Science/Mathematics

Roberto Cordone
DI - Università degli Studi di Milano

| | |
|---|---|
| Schedule: | Thursday 14.30 - 16.30 in classroom 201 |
| | Friday 14.30 - 16.30 in classroom 100 |
| Office hours: | on appointment |
| E-mail: | roberto.cordone@unimi.it |
| Web page: | https://homes.di.unimi.it/cordone/courses/2023-ae/2023-ae.html |
| Ariel site: | https://rcordoneha.ariel.ctu.unimi.it |

# 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 optimised 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

Neighbourhoods 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

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

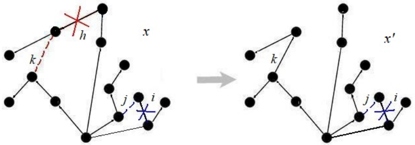
- have an overall effect independent from the order of application and easy to compute: for additive functions it is usually the sum

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

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 unfeasible*)




- *VRP*: transfer or swap nodes between different circuits (*moves on overlapping circuits could be unfeasible*)
- *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 $(u_j, u_{j+1})$!

# Dynasearch

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 $o \in \mathcal{O}$ and
  the value of a column equals the objective improvement $-\delta f_o(x)$
- $a_{io} = 1$ when move $o$ affects component $i$, $a_{io} = 0$ otherwise

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 of a graph
  - the columns can be seen as edges of a graph
  - 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

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

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

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

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

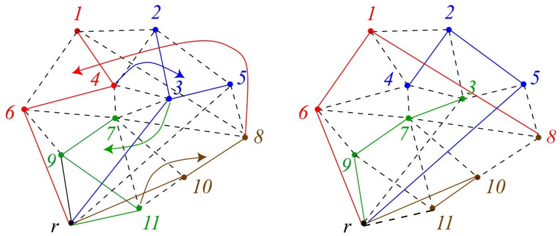
A circuit in such a graph corresponds to a closed sequence of transfers

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*




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 $\geq 0$ can be neglected because
  - the total variation of the objective sums the effect of the single moves

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

  - 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 $o_1, \ldots, o_k$

$$\delta f_{o_1,\ldots,o_k}(x) < 0 \Rightarrow \exists h : \delta f_{o_{(h+1) \bmod k},\ldots,o_{(h+\ell) \bmod k}}(x) < 0 \text{ for } \ell = 1, \ldots, k$$

    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

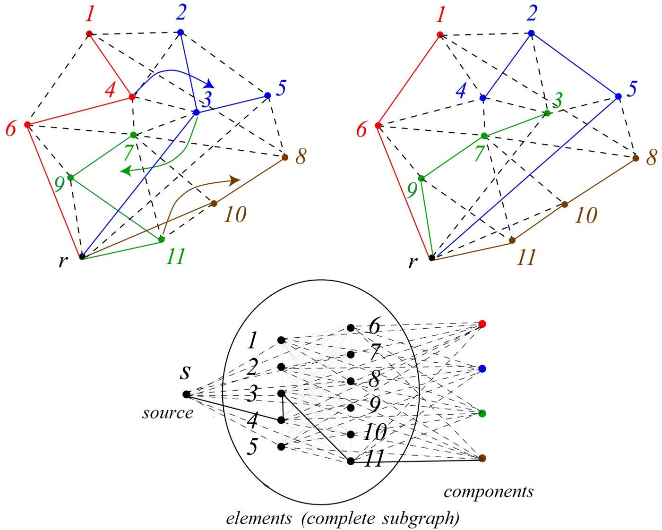
- starts from the source node
- ends in a component node
- visits at most one node for each component

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



# Order-first split-second

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

- builds a starting permutation of the elements to be partitioned
- 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

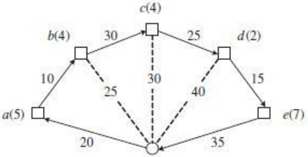
- each node $v_i$ corresponds to an element $s_i$
  plus a fictitious node $v_0$
- each arc $(v_i, v_j)$ with $i < j$ corresponds to a potential component $S_\ell$
  that assigns to the same subset the elements $(s_{i+1}, \dots, s_j)$
  - from $s_i$ excluded
  - to $s_j$ included
- the cost $c_{v_i,v_j}$ corresponds to the cost of the component $f(S_\ell)$
- the arc does not exist if the component is unfeasible

Consequently

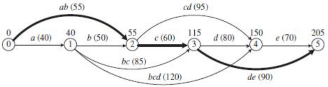
- each path from $v_0$ a $v_n$ represents a solution (partition of elements)
- the cost of the path coincides with the cost of the partition
- the graph is acyclic: finding the optimum path costs $O(m)$ where
  $m \leq n(n-1)/2$ is the number of arcs

# Example: the *VRP*

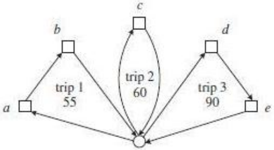
Given an instance of *VRP* with 5 nodes and capacity $W = 10$




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

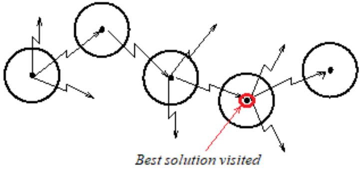



The optimal path corresponds to three circuits: $(d, v_1, v_2, d)$, $(d, v_3, d)$ and $(d, v_4, v_5, d)$

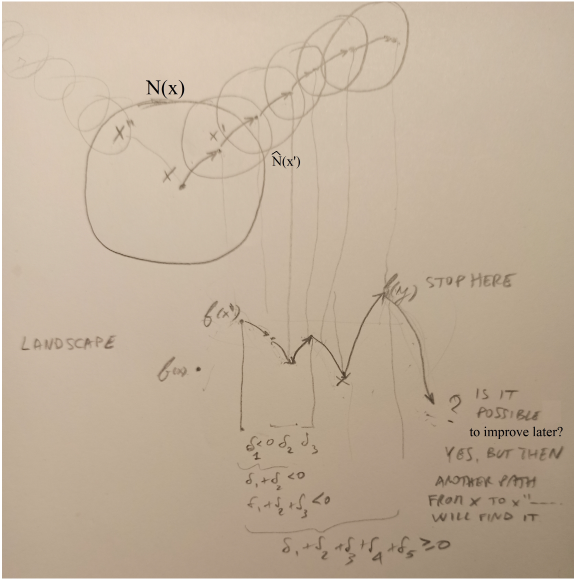


In the *VDS* a composite move is a sequence of elementary moves

- consider each solution $x'$ in the basic neighbourhood $N_{\mathcal{O}_1}(x)$
- from it, make a sequence of moves optimising 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 $x' \in N\left(x^{(t)}\right)$, 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* } | { *Variable Depth Search* } |
|---|---|
| Compute $f(x')$ | $y := x'$; $y^* := x'$; Stop $:=$ *false*; |
| | *While* Stop $=$ *false do* |
| | $\quad \tilde{y} := \arg \min_{y' \in \hat{N}(y)} f(y')$; |
| | $\quad$ *If* $f(\tilde{y}) \geq f(x')$ *then* Stop $:=$ *true*; *else* $y := \tilde{y}$; |
| | $\quad$ *If* $f(\tilde{y}) < f(y^*)$ *then* $y^* := \tilde{y}$; |
| | *EndWhile*; |
| | Return $f(y^*)$; |

*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*)

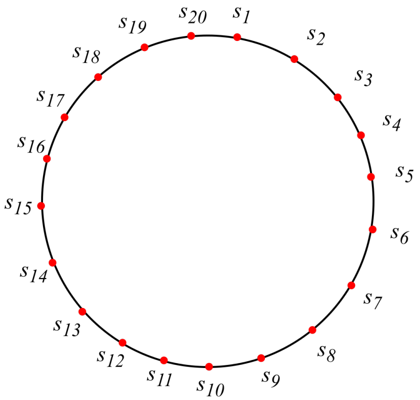
Lin-Kernighan's algorithm is a *VDS* with sequences of 2-opt exchanges:
a $k$-opt exchange is equivalent to a sequence of $(k-1)$ 2-opt exchanges, where each deletes one of the two arcs added by the previous exchange

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 $x \cap x''$...
- ...
- 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 $(i, j)$

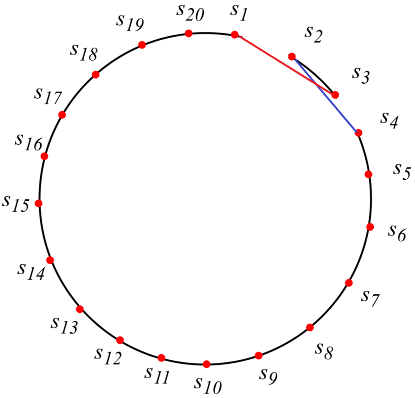



$$x = (s_1\,\overline{s_2\,s_3}\,s_4\,s_5\,s_6\,s_7\,s_8\,s_9\,s_{10}\,s_{11}\,s_{12}\,s_{13}\,s_{14}\,s_{15}\,s_{16}\,s_{17}\,s_{18}\,s_{19}\,s_{20})$$

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

# 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 $(s_2, s_4)$

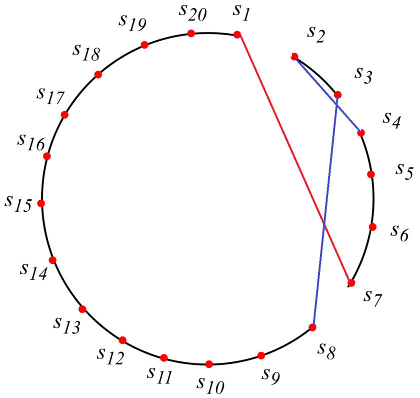



$$x' = (s_1\ \overline{s_3\ s_2\ s_4\ s_5\ s_6\ s_7}\ s_8\ s_9\ s_{10}\ s_{11}\ s_{12}\ s_{13}\ s_{14}\ s_{15}\ s_{16}\ s_{17}\ s_{18}\ s_{19}\ s_{20})$$

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 $(s_3, \ldots, s_7)$

# 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 $(s_3, s_8)$

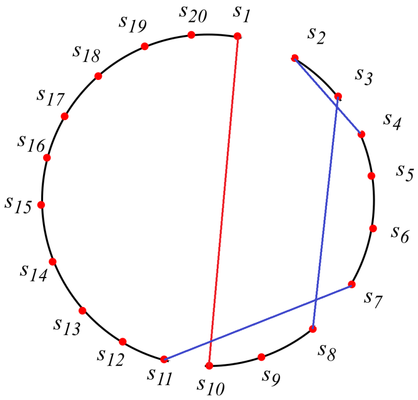



$$x'' = (s_1\ \overline{s_7\ s_6\ s_5\ s_4\ s_2\ s_3\ s_8\ s_9\ s_{10}}\ s_{11}\ s_{12}\ s_{13}\ s_{14}\ s_{15}\ s_{16}\ s_{17}\ s_{18}\ s_{19}\ s_{20})$$

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 $(s_7, \ldots, s_{10})$

# Example: Lin-Kernighan's algorithm

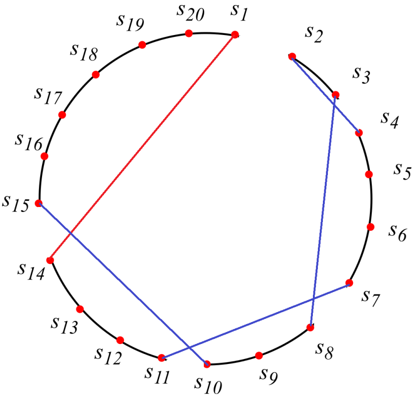
The exchange $(1, 10)$ replaces $(s_1, s_7)$ and $(s_{10}, s_{11})$ con $(s_1, s_{10})$ and $(s_7, s_{11})$




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

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 $(s_{10}, \ldots, s_{14})$

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

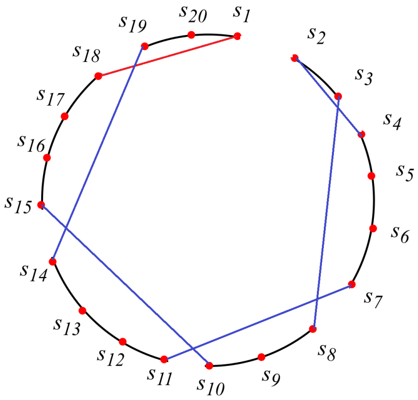



$$x^{iv} = (s_1 \; \overline{s_{14}\; s_{13}\; s_{12}\; s_{11}\; s_7\; s_6\; s_5\; s_4\; s_2\; s_3\; s_8\; s_9\; s_{10}\; s_{15}\; s_{16}\; s_{17}\; s_{18}} \; s_{19}\; s_{20})$$

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 $(s_{14}, \ldots, s_{18})$

# 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 worse 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

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

  - 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 $o_1, \ldots, o_k$

    $\delta f_{o_1,\ldots,o_k}(x) < 0 \Rightarrow \exists h : \delta f_{o_{(h+1) \bmod k},\ldots,o_{(h+\ell) \bmod k}}(x) < 0$ for $\ell = 1, \ldots, k$

    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*