Heuristic Algorithms

Lecture Notes

Prof. Roberto Cordone

Edoardo Marangoni

University of Milan
Department of Computer Science
January 16, 2021

Contents

Part I

Introduction to Heuristics and Problems

CHAPTER 1

Introduction

This course will be held by Professor Roberto Cordone, reachable at roberto.cordone@unimi.it. The lessons will be recorded in advance and posted on Ariel: the live lessons will focus on questions about the lessons and further insights.

1.1 Introduction

The introduction to this course will explain the goals of it and introduce a discussion on the word *heuristic*, which has several meanings, different for each field in which it's used. What follows is an overview of the problems addressed in this course and an introduction to the relative algorithms, as not *all* of the heuristic algorithms for *all* problems will be taught: in fact, this course will only treat **solution based heuristics** for combinatorial optimization problems.

The main goal of this course is convincing the student that heuristic algorithms are **not recipes for specific problems**! Therefore, this course is not just a show off of specific algorithms used for specific issues: the theme of this course can be summarized by saying that *any* heuristic can be used on *any* problem, obviously with a difference in performances and outcomes - the point is to consider general ideas appliable to a range of problems. So, first some common heuristic algorithms are going to be discussed and then the design of a heuristic tailored for a specific problem will be taught, considering each features of the problem and determining which algorithms can work on it and which can't, then implement it in some laboratories (see the Appendices) and evaluate quantitatively the performance of the algorithm.

1.1.1 Heuristics

The term **heuristics** comes from the greek *eurisko*, meaning *I find*. This word is derived from the very famous story of Archimedes, who had the problem of learning if the golden crown he had been given was actually golden or not - the greeks knew that gold is heavy, so if one knows the weight and the volume the ratio can be computed to tell if it's solid gold or plated gold. But, while measuring the weight is easy, the volume is harder to measure for an object shaped like a crown. Archimedes realized that if you put something in water, the displace in the water level is exactly the same as the volume of the object that has been put in the water. Eureka!

History

The word heuristic was coined in the 19th century, so no Greek ever used it. The greeks talked about something similar to heuristics: a mathematician called Pappus of Alexandria, living around the 4th century, published a book called "*analymenos*", meaning *treasure of analys*, in

which he discussed how to build a mathematical proof, starting from the data and evidence and coming to the thesis. In the 17th century, mathematicians like Descartes and Leibnitz discussed the *ars inveniendi* - which means *art of finding*; finding what, one may ask: Mathematical Truth of course. In the 19th century Bernard Bolzano discussed the *Erfindungskunst*, once again "the *art of finding*", which treated the most common strategies to build mathematical proofs. Shortly after, lots of philosophers, psychologists and economist began considering heuristic, but in a completely different context: heuristics were now pratical and simple rules that do not aim at an optimal result but at a *satisficing* one. Finally, in 1945, Gyorgy Polya published a book entitled "How to solve it" and the meaning of heuristic was brought back to its mathematical meaning.

What is a Heuristic Algorithm?

Considering the field of neuropsychology, philosopy, law and so on, the meaning of heuristics is the exact opposite of *algorithm*, as the latter is a formal procedure composed of deterministic steps leading to the solution of a problem, while the former is a *creative* solution; in other words, the algorithm has some correctness proof while the heuristic is drawn from common sense. So what does it mean to have a **heuristic algorithm**? A heuristic algorithm is a procedure run by a computer and composed by well defined steps and does not have a correctness proof, so its not guaranteed to be correct: this sound strange and useless, one may say. On the contrary, it can be quite useful provided that it *costs* much less in space and time than a possible exact alternative and that it *frequently* yields something *close* to the correct solution. The *closeness* is defined following a definition of *distance*, so it's necessary to have a **metric** in order to be able to express that a solution is satisficing or not and it's also necessary to have a **probabilistic distribution** that expresses how often the algorithm yields a satisfactory solution.

Going back to the idea of proofs, problems and so forth, we can observe that an exact algorithm is always followed by some proof of its correctness and, often, the proof embodies the algorithm itself building on some starting point and showing what the actions "do" in some context, obtaining a solution. The relation between the algorithm and its proof is very tight. Heuristics do not have this perfect correspondance, but they are used in order to find the right algorithm and its proof!

1.2 Different types of problems

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

A problem is generically a question on some mathematical system. These problems can be classified based on the nature of their solution: **decision problems** have a boolean solution, **search problems** have as solution objects found in some system, **optimization problems** have as solution a number being the value of a suitable function estimating how good or how bad subsets drawn from a subsystem are. An example of this last type is the problem of finding the minimum time required to go from home to the town center: the system is the map of the town; the subsystems are specific paths. Each of them has a length and the problem is to find the path with the shortest time of trasversal. Optimization problems can be combined with search problems, if not only a number is sought but also the solution itself: these are the problems that are going to be considered. Other families of problems are **counting problems** and **enumeration problems**.

1.2.1 Optimization-Search Problems

An Optimization-Search Problem can be represented as

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

Where x is a solution which describes each subsystem of the problem, the **feasible region** X is the set of subsystem which satisfy given conditions and $f : X \to \mathbb{R}$ is the **objective function** that quantitatively measures the quality of each subsystem. Usually, $\text{opt} \in \{\min, \max\}$.

While in the case of Optimization the problem consists in determing the **optimal value** f^* of the objective function

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

in search problems the problems consists in determing at least one **optimal solution**:

$$x^* \in X^* = \arg \text{ opt } \mathrm{f}(x) = \{x^* \in X : \mathrm{f}(x^*) = \text{opt } \mathrm{f}(x)\}$$

The arg function encloses the whole set that verify what follows. People are interested in Optimization-Search problems as several application fields require objects or structures characterized by very high or very low values of a suitable evaluation function. Exact optimization is nice but very costly from a computational point of view; therefore, often, evaluation functions are just approximation of what really goes on. In this course we're assuming the point of view of optimization, trying to optimize as good as possible the objective function.

Finding objects that satisfy some condition that's interesting in a search problem or even a decision problem can be transformed into an optimization problem by relaxing some of those conditions. Let's give an example of this: imagine trying to find a solution to the problem of coloring a map with exactly four colors; a solution problem would ask for a colored map. A decision problem would ask for the existance of a colored map. An optimization problem would ask for a minimum number of colors which a map can be colored with! Starting with five or six or whatever colors, one can try to minimize the violations to the condition of the threshold of four - once the arrangement is found and has zero violations both the search and decision problem are solved as well.

1.2.2 Combinatorial Optimization

A problem is a Combinatorial Optimization problem when the feasible region X is a **finite set**, so it has a finite number of solution. This is a restrictive assumption, but there are lots of problems which have infinitely many solution that can be somehow reduced to problems which have a finite number of solution. Some examples are Linear Programming, in which one has to find only the *basic* solution and not *all* of them. In general lots of ideas developed in Combinatorial Optimization problems can be extended to other problems with infinite solutions.

An alternative definition of Combinatorial Optimization problems is as follows. These problems are known as *combinatorial* because they're based on the idea that the solutions are *combinations* (a subset) of a ground set with irrelevant ordering. So, our definition of sees the feasible region as $X \subseteq 2^B$ for a given finite suitable ground set B. In other words, B is a given finite set and 2^B is the power set of B, which contains the empty set $\emptyset$, all the singletons and each combination of elements: X is a subset of this set, so each solution if a subset of B (an element of X).

1.3 Different types of heuristics

1.3.1 Solution Based Heuristics

Solution based heurisitcs are based on the operations that you can make of a subset of a ground set. If one wants to find a subset of a set, one can build the subset by taking elements and stopping. This is a form of **constructive heuristic**: starting from an empty solution a solution is constructed adding elements. The idea can be reverted by starting with the full set and removing elements from of it, and this is a form of **destructive heuristic**. One can exchange elements starting from a subset constructed in some way; this is different from the heuristics described before as the sets were strictly growing or decreasing: in **exchange heuristics** the elements of the subsets are just exchanged, no cardinality limitation is given. Finally, the third family of heuristics is that of **recombination heuristics**, in which multiple subsets are created contextually and elements are exchanged between them.

In general a designer of a heuristic algorithm would use one of this techniques or combine them.

1.3.2 Metaheuristics

There are two important things that intervene in the design of an algorithm, **randomization** and **memory**. Some algorithms use randomization and some don't, just as some use memory and some don't. These two elements are completely orthogonal to the classification seen before, so the full set of options of heuristic is $4 * 4 = 16$. Heuristic algorithms that use randomization and/or memory are called **metaheuristics**.

Randomization can increase the possibility of finding good solutions, even thought the randomization algorithm itself is deterministic and uses an entropy source (epoch time, keyboard activity, radioactive decay...). Memory-using algorithms generate a list of solution and use previous solutions to generate the following ones.

1.3.3 Caveats

It's necessary to be aware of some risks in order to avoid taking them.

- Avoid the choice of an algorithm if not for its intrinsic value. There are communities in which a certain algorithm is considered very good even if other algorithms may be better: one has to prove that an algorithm is good.
- Avoid the use algorithms based on analogies deriving from physical and natural phenomena.
- Sometimes heuristics are used in stance of exact algorithms: this is plain stupid.
- Avoid number crunching, performing complex computations on unreliable data and "SUV attitude" relying on hardware power.
- Avoid unjustified components and parameters.
- Avoid overfitting, that is adapting components and parameters of the algorithm to the specific dataset used in experiments.

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

CHAPTER 2

Combinatorial Optimization Problems

Let's shortly go over what a Combinatorial Pptimization problem is: the objective is to minimize or maximize an objective function $f(x)$ defined as $f : X \to \mathbb{R}$. This set X is a subset of the power set of B ($X \subseteq 2^B$), with B being a finite ground set; therefore, $x \subseteq 2^B$, or, x is some kind of *combination*.

This section will unroll a list of Combinatorial Optimization problems in order to give an idea of what they are, how to recognize one and to know how to deal with it. One of the basic points will be trying to understand *what B is*, as it's the fundament on which the algorithm is built on top of: constructive, destructive, exchange and recombinaton heuristics all operate on objects of the ground set. Easiest problems revolve around **set problems**, which are the first to be introduced. Then, **logic function problems**, **numerical matrix problems** and **graph problems** are considered.

Reviewing several problems instead of just giving general rules because some practical problems are involved and not **abstract contexts**: these abstract ideas must be applied to different algorithms for different problems in order to be well understood, as they can be good or bad depending on the *structure* of the problem. In particular, some ideas only work on some problems, and some problems apparently different can have some hidden structur that algorithms can exploit. The point is that we're trying to apply abstract to ideas to many problems trying to understand what's common in these problems and what's different between each other.

2.1 Weighted Set problems

2.1.1 The Knapsack problem

Having a knapstack of limited capacity, one wants to fill it with different objects, each of which has a "value". Not all of the objects fit inside, so a choice has to be made: only a subsect of **maximum value** needs to be put inside the knapsack. In this context, the elements of the problems are: the set E of elementary objects, the function $v : E \to \mathbb{N}$ describing the volume of each object, the number $V \in \mathbb{N}$ describing the capacity of the knapsack and, finally, the function $\phi : E \to \mathbb{N}$ describing the value of each object.

The ground set is trivially the set of objects, $B \equiv E$; the feasible region includes all of the subsets of object whose total volume does not exceed the capacity of the knapsack:

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

The objective is, clearly, to maximize the total value of the chosen objects, finding

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

Table 2.1 gives an example of possible values for this problem. In the case represented in the table and given $V = 8$, two possible solutions can be $x' = \{c, d, e\} \in X$, which has $f(x') = 13$ and $x'' = \{a, c, d\} \notin X$ which yields $f(x'') = 16$. In other word x'' is not a **feasible** solution as it is not in X, even though it was still technically defined as one.

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

Table 2.1: Knapsack problem values.

2.1.2 Maximum Diversity problem

The Maximum Diversity Problem (MDP) problem is defined on a metric space, so a space with a notion of *distance*. This problem is defined on a set of points P and on a function $d : P \times P \to \mathbb{N}$ which provides the distance between two points. Also, a number $k \in \{1, \cdots, |P|\}$ is given and represent the number of points to select.

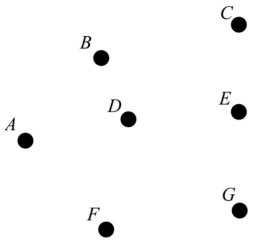


Figure 2.1: An instance of MDP.

What is to be found is a subset of exactly k points such that the sum of all the pairwise distances between the points in the subset is maximized. This is a combinatorial problem as the number of point pairs combination is limited and the ground set is $B \equiv P$, the set of all points. The feasible region includes all subsets of k points

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

And the objective function is

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

So, we want to maximize this sum. Solutions with $k = 3$ is represented in Figure 2.2.

2.1.3 Interlude I: the objective function

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

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

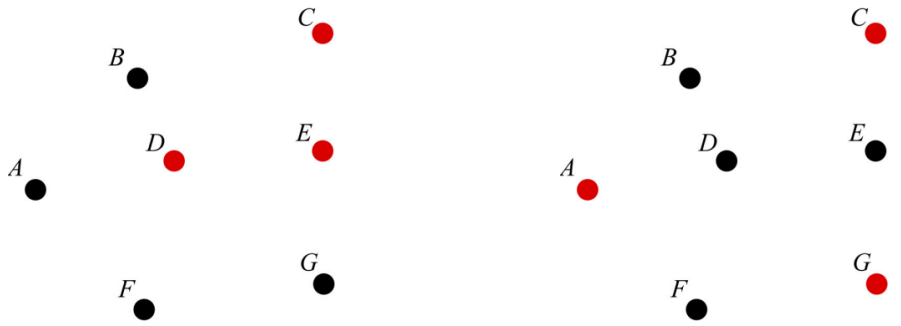


Figure 2.2: Two possible solutions for the previous MDP instance.

The computation of this function can be really complex, even exhaustive: this, of course, is not the case of the previous two problems. The KP has an **additive objective function** which sums values of an auxiliary function defined on the ground set:

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

This is interesting as it means that one needs to store only the values of the auxiliary function ϕ which are $|B|$ values and not $|2^B|$, as it would be for the values of the objective function which is defined on X. The same goes for the MDP and its auxiliary function d, even though the latter has a quadratic objective function, as given $n = |B|$ points in the worst case one needs to sum $(n * (n-1))/2$ numbers (the distance from each point to each other point, a complete graph) but the calculation of the objective function is still "just" a sum, hence having a polynomial complexity.

While for the KP once a value for a specific combination is given one can modify the elements in the combination and easily recalculate the value of the function (e.g: $f(\{c,d,e\}) = 13$, take out e and put f in, then $f(\{c,d,f\}) = 13 - \phi(e) + \phi(f) = 10$), the value of the objective function of the MDP has to be treated differently to be calculated in linear time, as we'll show later.

Another important remark is that the KP and the MDP problem are defined on the whole of 2^B and this is generally unuseful, as we're looking for a feasible solution but, in some occasion, it will be really useful.

To summarize, look at the problem and try to understand how the objective function goes: is it additive? Is it quadratic? Is it easy to compute from scratch? Is it easy to update? On what is it defined?

2.2 Partitioning Set problems

In these problems a set of objects is given, the goal is to divide it in subsets obtaining a partition with some pecularity.

2.2.1 Bin Packing problem

The Bin Packing Problem (BPP) is a problem in which a set of object with a volume is given and each of them needs to go in some container of fixed capacity. The objective is to use the least number of containers. Formally, a set E of object is given, a function $v : E \to \mathbb{N}$ gives the volume of each object, a set C of containers is given and a number $V \in \mathbb{N}$ represent the volume of the containers.

How to describe this problem as a Combinatorial Optimization problem? How can a solution be described as a subset of the ground set B? The ground set is defined as

$$B = E \times C$$

all the $\langle object, container \rangle$ pairs. This is because a solution is a subset of these pairs: take the first object and put it in the first container, take the second and put it in the first container and so on. Once a list of this pairs containg all the elements in E is built, a (potential) solution is achieved.

The feasible region includes all partitions of the objects among the containers not exceeding the capacity of any container:

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

with $B_e = \{(i,j) \in B : i = e\}$ and $B^c = \{(i,j) \in B : j = c\}$. In words, the first is a partition contstraint (every element appears exactly once in the solution) and the second is a volume constraint (the sum of the volumes of all elements in a container must not exceed the volume of the container, for all containers).

The objective is to minimize the number of used containers:

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

An example of instance and solution of BPP is in Figure 2.3.

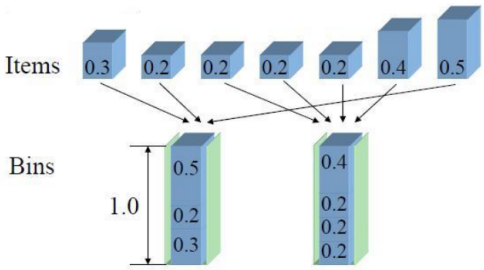


Figure 2.3: An instance of BPP.

2.2.2 Parallel Machine Scheduling problem

The Parallel Machine Scheduling Problem (PMSP) is a problem in which a set of tasks have to be divided among a set of machines so as the completion time is minimized. Each task has a duration d: the problem is to assign tasks to machines so that the maximum time a machine operates to solve its tasks is minimized. Formally, a set T of tasks is given and each task has defined a value d ($d : T \to \mathbb{N}$) to be distributed on a set of M machines.

As before, the ground set is $B = T \times M$, including all possible $\langle task, machine \rangle$ pairs. It's important to underline that the *sequence* in which tasks are executed by machine is not important: what matters is the completion time, which is simply the time at which the last task completes or the maximum time at which a machine completes the execution of its tasks.

The feasible region includes all partitions of tasks among machines

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

And the objective function is

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

in word, the subset x that minimizes the maximum time of execution for the tasks scheduling (x that minimizes the maximum execution time among all machines working with the scheduling x).

2.2.3 Interlude II: again, the objective function

It's necessary to get familiar with the fact that the ground set B isn't always one of the sets in the problem, but it may be some form of combination of them. Now, let's address the questions arised in the last interlude. This time, the objective functions aren't additive and aren't trivial to compute (but not that hard as well). There's a polynomial algorithm that calculates the objective function, but is not as easy as before! Is it possible to define objective function values on solutions that are not feasible? In general yes, as said before, and in this case it is as well.

In this case, the change in a solution has a variable impact on the objective: in some cases nothing happens (like scheduling a task on a machine and not changing the overall maximum), in others the objective function changes equally to the value of the change itself (like adding a task on the machine with the overall maximum) and, in some other cases, the change induces an intermediate change in the objective function itself, not exactly the value of the change; an example is moving a task from the machine with the maximum execution time to another: it's not necessarily true that the overall max time changed by the execution time of the swapped task.

It's important to consider if these values can be calculated without recomputing the whole function and this will be considered later. Another intresting point is that the objective function is **flat**, meaning that several solutions have the same value: this is a problem when comparing different modifications.

2.3 Logic Function problems

2.3.1 Max-SAT problem

This problem is very well known in Computer Science, and it's called Max-SAT, where SAT stands for *satisfiability*. The problem is formally composed of these elements: a set V of logical variables x_j with a boolean value, a literal function consisting of an affirmed or negated variable

$$l_j(x) \in \{x_j, \neg x_j\}$$

which are used to create disjunctions

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

called *logical formulae*, which in turn are used to build a CNF, or *conjunctive normal form*, logical product of logical formulae:

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

It's called conjunctive *normal* form as a theory proves that any logical function can be put in this form without modifying values it assumes for any truth assignment (i.e. it's semantically

equivalent). Statisfying a logical function means finding a truth assignment so that it assumes the value 1. It's possible to consider a *weighted* version of this problems: for example, let

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

be the set of variables,

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

be the set of literals,

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

be the CNF to satisty and

$$w_i = 1 \quad \forall C_i$$

the weight function. A solution $x = \{(x_1,0),(x_2,0),(x_3,1),(x_4,1)\}$ satisfies 5 formulae out of 7, so $f(x) = 5$, as The ground set is the truth assignment: $B = V \times \{1,0\}$. Out of this basic ground set, assignments are taken in order to build a solution. The feasible region includes all subsets of truth assignments that are **complete** (each variable has at least a literal) and **consistent** (each variable has at most one literal):

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

with $B_{x_j} = \{(x_j,0),(x_j,1)\}$. The objective is to maximize the total weight of the satisfied formulae

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

The complexity of the objective function is polynomial, and is not defined on unfeasible solutions. In the case of uniform-weighted there are only restricted values of the Max-SAT, ranging from 0 to n, the number of logical formulae, even though there are $2^{|V|}$ combinations that can be considered.

2.4 Numerical Matrix problems

2.4.1 Set Covering problem

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

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

it's said that a column j *covers* a row i when $a_{i,j} = 1$.

The SCP requires to cover all the rows using the columns with the minimum cost. An example is given in Table 6.1, and the solution $x' = \{c_1, c_3, c_5\} \in X$ has $f(x') = 19$. Formally, in this problem, the ground set $B \equiv C$, the set containing all the columns; the feasible region includes all subsets of columns that cover all rows

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

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

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

which is additive.

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

Table 2.2: Example of the SCP.

2.4.2 Interlude III: the feasibility test

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

In the case of the KP, the feasibility test required to compute from the solution and test a single number (the total weight) just as in the MDP, where the cardinality of the solution was under some restriction. Other problems, like Max-SAT and PMSP require to test the feasibility against a single set of numbers and, finally, problems like BPP require to test against several sets of numbers.

Some modification to the solutions can be forbidden *a priori* in order to avoid infeasibility. Suppose to have a feasible solution to the MDP: any change in which the number of points removed is not equal to the number of points added makes the solution unfeasible. Some modification are not guaranteed to add unfeasibility to the solution, requiring a *posteriori* test which may require starting from scratch.

2.4.3 Set Packing problem

The Set Packing Problem is very similar to the SCP, as it is still based on binary matrix with set of rows and columns with a function $\phi : C \to \mathbb{N}$ providing a value for each column. Contrary to the SCP, in the SPP two columns j' and j'' *conflict* if they have $a_{ij'} = a_{ij''} = 1$, in words they both "pack" the row i. The goal of the problem is choosing the columns with the maximum value without any overlaps in packs (so if a row is not packed it's still a feasible solution).

Formally, the ground set is the set of columns $B \equiv C$; the feasible region includes all subsets of nonconflicting columns

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

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

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

2.4.4 Set Partitioning problem

The Set Partitioning Problem (SPP) is similar to both the two previous problems: a binary matrix is given and each column has a cost represented by the function c, but in this problem

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

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

and the objective function is to minimize the total cost of the selected columns

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

2.4.5 Interlude IV: the search for feasible solutions

Let's ask another question. Heuristic algorithms often require to solve the problem of **finding a feasible solution** $x \in X$. This is clearly a search problem. Clearly, as they're defined as starting from an initial solution, exchange and recombination heuristics need to start from a valid subset which is a feasible solution. This initial feasible solution must be found!

Sometimes finding one is **trivial**, like an empty set, or $x \equiv B$ or something like that: in the KP and SPP (packing) a valid initial solution is $x = \emptyset$, in the SCP $x \equiv B \equiv C$ is a valid initial solution; sometimes it's enough for an initial solution to satisfy some **constraint**, like $|x| = k$ for the MDP, and some other times an initial solution has to satisfy some **consistency constraint**, such as assigning each task to a machine in the PMSP, a value to each logic variable in the Max-SAT and so on. These solution may not be optimal, but surely are feasible and valid as a starting point.

In some other situation it can be very hard to find even a solution far from being optimal: that's the case for BPP and SPP (partitioning), for which no polynomial algorithm is known to solve it.

In order to find a solution when the problem is really hard it's possible to make a **relaxation** on some constraint of the problem and enlarge the feasible region X to X'. This means that f must be defined on X', and sometimes this is not possible.

2.5 Graph problems

2.5.1 Vertex Cover problem

Given an undirected graph $G = (V, E)$, select a subset of vertices of minimum cardinality such that each edge of the graph is incident to it. The ground set is $B \equiv V$, the feasibility region is

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

((i, j) represents an edge but is actually a set of vertices, so $x \cap (i, j)$ is meaningful) and the objective is to minimize the number of selected vertices

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

The Vertex Cover Problem can be also set as a weighted problem, where the weighted objects are the vertices to choose (and not the edges): in this case the objective function is the sum of the weights of the chosen vertices.

For example, in Figure 2.4 the subset $x = \{1, 2, 3, 4, 5, 6\}$ is a feasible solution.

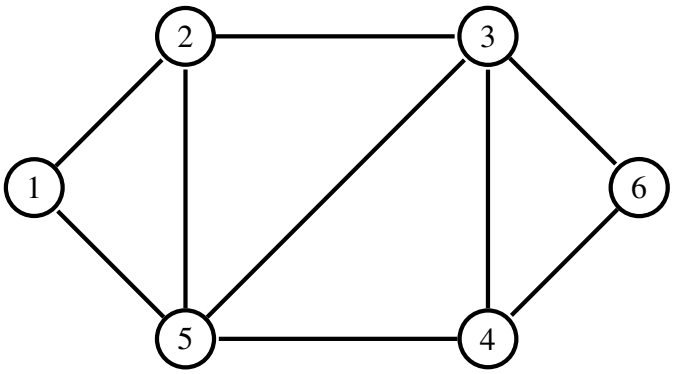


Figure 2.4: An undirected graph.

2.5.2 Maximum Clique problem

Given an undirected graph $G = (V, E)$ and a weight function defined on the vertices $w : V \to \mathbb{N}$, select the subset of pairwise adjacent vertices of maximum weight. The problem is solved by finding a subset of vertices such that all the pair of vertices correspond to an edge, so all the vertices in the subset have to be pairwise adjacent. For example, in Figure 2.4 the subsets $x' = \{1,2,5\}$ and $x'' = \{3,4,6\}$ are feasible solutions.

In general, the MCP is solved by a solution subset with the maximum cardinality, and if it's a weighted MCP, the solution aims to maximize the sum of the weights of selected vertices. Formally, then, the ground set is $B \equiv V$, the feasible region includes all subsets of pairwise adjacent vertices

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

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

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

or for unweighted problems, simply the cardinality of x.

2.5.3 Maximum Independent Set problem

Given an undirected graph $G = (V, E)$ and a weight function defined on the vertices $w : V \to \mathbb{N}$, select the subset of pairwise nonadjacent vertices of maximum weight. This problem is kind of the opposite of the Maximum Clique Problem: what is to be found is a subset of vertices of maximum weight that aren't edges. For example, in Figure 2.4 the subsets $x' = \{1,6\}$ and $x'' = \{1,3\}$ are feasible solutions.

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

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

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

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

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

2.5.4 Interlude V: relations between problems

As usual, the questions one should ask for any problems are the same ones: how to compute the objective function? What happens if we throw out a vertex? How to verify the feasibility? What happens if we add or remove a vertex to a feasible solution?

These last three graph problems are quite similar problems. It should be already known from basic Computational Complexity Theory that some problems can be **reduced** to other problems, that is one can "use" a problem to solve another problem. A clear example of this is the reduction of an instance of a MCP to an instance of a MISP and vice versa: in fact, by computing the complement graph of the initial instance, a valid solution for the "new" problem is a valid solution for first problem as well; in other word, you can find a solution for a MQP by reducing it to an instance of a MISP (or vice versa): this obviously works for both the heuristics solutions and the optimal solutions.

Surprisingly, the Vertex Covering Problem is reducible to the Set Covering Problem: in fact, all it's needed is to transform each edge e_i in a row r_i and each vertex v_j in a column c_j: now, if an edge e_i is incident on a vertex v_j, then $a_{ij} = 1$(every row will have only two 1's), otherwise $a_{ij} = 0$. Furthermore, if the VCP is weighted, it's enough to assign the cost of each vertex to the associated column: solving the SCP will then retrieve a set of covering vertices with minimum weight for exact algorithms or, for heuristics algorithms, a feasible solution. Even more surprisingly, in this case the reducibility is unidirectional: while it's possible to reduce VCP $\rightarrow$ SCP, it's impossible, except cases in which each row has exactly two 1's, to reduce it the other way, so generally SCP $\nrightarrow$ VCP.

An even more difficult situation is the relation between the Bin Packing Problem and the Parallel Machine Scheduling Problem: tasks and objects are very similar as they have to be *assigned* in some way, but there's a big difference as the machines have an unlimited "capacity" - their working time - and a given number of machine; on the contrary, bins are not limited in number but they are limited in capacity. However, let's try to reduce the problems starting from a BPP instance: the first problem is that while the objects can be transformed into tasks, the number of bin (which is virtually infinite) can't be transformed into the number of machines, which has to be finite and given. So, the plan might be to make an assumption, guessing a number for the machines: solving this problem (with an heuristic algorithm or an exact one) will yield some solution. If this solution, which is the maximum time of overall execution, exceed the capacity of the bins it's necessary to increase the number of available machines and recompute the solution. If, on the other hand, the solution does not exceed the bins' capacity, one can choose to try and subtract a number of machines and recompute the solution. This reduction requires a **polynomial number of reduction iteration**.

Again, it's important to stress the fact that in case of reducibility, a heuristic solution in the reduced instance is a heuristic solution for the original problem; studying reletions between problems is important even without thinking about algorithms!

2.5.5 The Travelling Salesman problem

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

The ground set is the arc set: $B \equiv A$, the feasible region includes all the subsets of circuits on the graph visiting all nodes and the objective is to minimize the total cost of the selected arcs

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

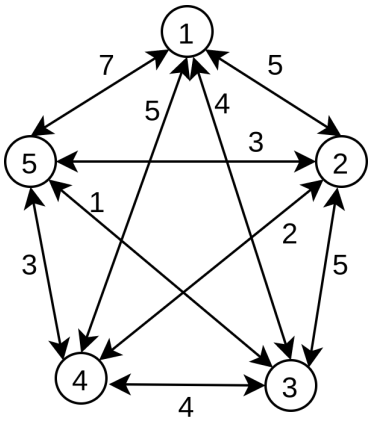


Figure 2.5: A TSP instance.

How to determine whether a subset is a feasible solution? That subset must identify a circuit on the graph and every node has to have exactly arc entering it and one going out of it; furthermore, a single visit of the graph using the selected arcs should visit all the nodes - in other words, no subtours are allowed. If a solution is modified, depending on the modification itsel it can be necessary to recalculate the objective function, and finding a feasible solution can be quite hard as well - in a general graph is Hamiltonian or not is a strongly NP complete problem, trivial only in complete graphs.

2.5.6 Capacitated Minimum Spanning Tree problem

Given an undirected graph $G = (V,E)$, a root vertex $r \in V$, a cost function defined on the edges $c : E \to \mathbb{N}$, a weight function defined on vertex $w : V \to \mathbb{N}$ and a maximum capacity, the CMSTP requires to select the minimum cost spanning tree such that each subtree appended to the root have weight not larger than the capacity. Since a minimum spanning tree is a subset of edges, $B \equiv E$, the feasible region includes all spanning trees such that the weight of the vertices spanned by each subtree appended to the root do not exceed W and the objective is to minimize the total cost of selected edges

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

Since the objective function is additive, it's quite easy to evalue it, but whilst it is easy to recompute it, it is harder to verify its feasibility, except for trivial situations. Finding a capacitated spanning tree is a strongly NP-complete problem, so it's quite hard to find a feasible solution unless the graph is complete. Given a set of edges, in order check if it's a feasible solution or not it is necessary to build a correct representation of a tree and then visit each subtree, summing weights of vertices, in a sort of DFS fashion.

So, if the objective function is fast to evaluate and to update, it is easy to give nonoptimal solutions given the covered vertices, while the feasibility test is not very fast to perform (acyclicity test and total weight computation) and to update; is it possible to change something and make everything easier? Changing the ground set itself may be helpful: this problems is quite similar to the BPP! In fact, we're trying to take vertices with some weight and put them in a group of vertex of limited capacity, just like the containers. By introducing a set of subtrees T with $|T| = |V \setminus \{r\}|$, vertices could be assigned to subtrees, even if some subtree remains empty, making $B \equiv V \times T$. The feasible region then includes all partitions of the vertices into connected subsets of weight less or equal to W: the objective is to minimize the sum of the costs

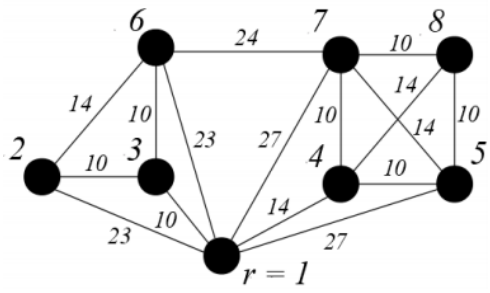


Figure 2.6: An instance of CMSTP.

of the subtrees spanning each subset of vertices plus the edges connecting them to the root.

This representation makes it easier to test for feasibility in terms of weight, as it's sufficient to check subtree by subtree calculating the partial weghts, while in the previous case a full representation of the tree had to be constructed and visited; furthermore checking for the partitioning of vertex is easy and achieved by checking that each vertex appears in exactly one subtree. Nontheless, this representation has some information about the edges that is considered *implicit*: while it can be simply imposed that the edges in a subtree do not form a cycle, in order to be sure that the nodes are connected and at least one of them is connected to the root (so that they actually form a fully reachable subtree from the root) is necessary to visit the subsets of vertices and prove that they make a singly connected component.

In order to compute the objective function, which is based on edges, one must compute the minimum spanning tree for each subtree with respect to the nodes, which has some cost but can be useful in order to avoid creating nonptimal solutions. So, as the feasible region changes including all partitions of the vertices into connected subsets of weight less or equal to W, the objective changes as well, being minimizing the sum of the costs of the substrees spanning each subset of vertices plus the edges connecting them to the root.

2.5.7 Vehicle Routing problem

Given a directed graph $G = (N,A)$, a "depot" node $d \in N$, a cost function defined on arcs $c : A \to \mathbb{N}$, a weight function defined on the nodes $w : N \to \mathbb{N}$ and a capacity $W \in \mathbb{N}$, the VRP is solved by selecting the set of minimum cost circuits that visit the nodes starting from the depot such that each one has a total weight not larger than the capacity. This is something like the Travelling Salesman Problem but several different subtours span all vertex - the idea is that a vehicle is used to visit customers in need but have a finite number of products in stock to give them, so more vehicles are needed.

The ground set could be $B \equiv A$ or the set of all $\langle node, circuit \rangle$ assignment pairs $B \equiv N \times C$, with C being the set of all possible circuits, and the feasible region changes with the choice of the ground set: it could include all arc subsets that cover all nodes with circuits visiting depot and whose weight does not exceed W or could include all partitions of the nodes into subsets of weight not larger than W and admitting a spanning circuit (which is an NP-hard problem). The objective is always to minimize the total cost of the selected arcs:

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

Evaluating the objective function in the first case is easy as it's just a sum, while in the second case it's necessary to first find the arcs which is substantially finding an Hamiltonian Cycle on

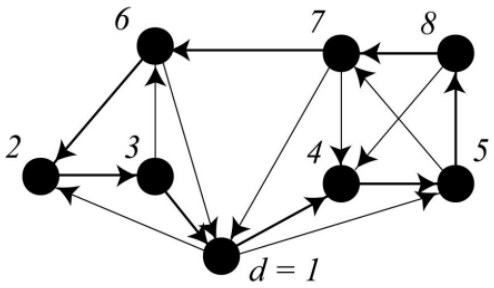


Figure 2.7: A VRP instance.

a subset of nodes and arcs, while evaluating the feasibility on the first case requires a visit but it's just a sum in the second representation, which also makes easier to update the solution.

2.5.8 Interlude VI: combining alternative representations

Different ground set for the same problem can have advantages and disadvantages: some representation can be good for the feasibility checks and some can be good for the evaluation of the objective function. Which one to chose depends on the operations that are more frequently used and a possibility is to use both if they're used more frequently than updated, so that the burden of keeping them up to date and consistent is acceptable.
To summarize, the questions that one must ask regarding a problem are: how to compute the objective function? How to prove that a subset is feasible? How to find a feasible solution? How to evaluate the feasibility test? What happens when a change to a feasible solution is made: is it still feasible? Is it certainly not? Is it necessary to start from scratch to evaluate the objective function or is there a better way? What is the correct definition of the ground set? Are there more possible definitions? Are there relations between this and other problems?

Part II

A Priori and A Posteriori Algorithm Analysis

CHAPTER 3

Theoretical Efficiency

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

3.1 Cost and Complexity

The following is common background for COMPSCI students, treating basics notion of **computational complexity**. We've seen that a problem is a question on some system of mathematical object: often, that system contains infinitely many of those object. For example, one could wonder if 7 is a prime number, 10 is, 50 is and so on: they're not different problem these are different *instances* of the same problem. So, in formal terminology, a **problem** is the function which relates instances in the set of all instances $I \in \mathfrak{I}$ and solutions in the set of all solutions $S \in \mathfrak{S}$

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

Defining a function doesn't mean knowing *how* to compute it. This last part relates to **algorithms**.

3.1.1 Algorithms

An algorithm is a formal procedure composed by elementary steps formed in a finite sequence. These steps are determined completely by some input and all result of all the previous steps; in particular, an algorithm is said to *solve* a certain problem if given as input an instance $I \in \mathfrak{I}$ of a certain problems it returns an output among the solution S_I, so

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

just as the definition of problem, but the algorithm has to define a function and the way to compute it. In particolar, an algorithm is **exact** if its associated function is exactly the same with the problem itself (so every instance has its own "right" solution) and **heuristic** otherwise. An heuristic algorithm is useful if it is efficient and effective.

We will consider the concepty of efficiency and effectiveness, both a priori and a posteriori.

3.1.2 Cost of a heuristic algorithm

The cost of a heuristic algorithm is the computational cost in space (the memory used) and in time (the time necessary for the computation to conclude). Usually, in the definition of complexity, time is more important than space. First, space is a renewable, while time is definitely not. Second, using space requires to use at least much time and, finally, it is technically easier to distribute the use of space than of time. Space and time are *partly* interchangeable, as it's possible to reduce the use of one by increasing the use of the other.

The time required to solve a problem depends on several aspects: the specific instance to solve, the algorithm used, the machine running the algorithm and so one. What it's needed is a measure (possibly symbolic) that allows to make some measurement. First of all, it's necessary that the measurement is **not relative to the technology** used, otherwise the measure has to be changed everytime the machine changes; it's to be **concise**, meaning that it's summarized in a symbolic expression and it's to be **ordinal**, meaning that it's sufficient to compare different algorithms. Computetional time in seconds is not good and violates all requirements.

3.1.3 Worst-Case asymptotic time complexity

The classic time notation is the worst-case asymptotic time complexity: the physical time is replaced with the number of steps applied to the input data by an algorithm; this requires a definition of **elementary operations**, which is an abstract concept: sum, difference, multiplication, division, read and write are usually considered elementary steps. The time is defined as the number T of elementary operation performed, which is usually a function of the size of an instance as a suitable value n, for example the number of elements of the ground set, varianles of formulae of the CNF, nodes or arcs of the graph... What matters is the **worst-caste**, the maximum of the T on all instance of size n

$$T(n) = \max_{I \in \mathfrak{I}_n} T(I) \text{ where } n \in N$$

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

The Θ functional space

One can say that the number of operation depending on the size of the instance in the worst case (the longest taking instance) is

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

which means, formally, that

$$\exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} : c_1 f(n) \leq T(n) \leq c_2 f(n) \ \ \forall n \geq n_0$$

when one can find two real positive coefficients and a natural coefficient such that the computational time is restricted between $c_1 f(n)$ and $c_2 f(n)$ for all values of n greater than n_0. Asymptotically, $f(n)$ estimates $T(n)$ up to a multiplying factor: for large instances, the computational time is at least and at most proportional to the values of $f(n)$.

The O functional space

If one considers only the approximation from above you get the Big-O functional space and

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

formally means that

$$\exists c \in R^+, n_0 \in N : T(n) \leq cf(n) \ \ \forall n \geq n_0$$

Asymptotically, $f(n)$ overestimates $T(n)$ up to a multiplying factor; in other words, after n_0 exists some real constant c such that $c \cdot f(n)$ is always bigger than $T(n)$.

The Ω functional space

If one considers only the approximation from below one gets the Ω functional space and

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

formally means that

$$\exists c \in R^+, n_0 \in N : T(n) \geq cf(n) \ \ \forall n \geq n_0$$

Asymptotically, $f(n)$ underestimates $T(n)$ up to a multiplying factor; in other words, after n_0 exists some real constant c such that $c \cdot f(n)$ is always smaller than $T(n)$.

In the context of Combinatorial Optimization problems there's always an algorithm which is **exhaustive** but which has to consider (in the worst case) *all* possible subsets of the ground set, so each of the $x \in 2^B$ subsets.Suppose that to test each subsets for its feasibility it's used time $\alpha(n)$ and to evaluate the objective $f(x)$ in time $\beta(n)$, then the time complexity of the exhaustive algorithm is

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

which is **at least** exponential and most of the time impractical. In CO, the main distinction is between polynomial complexity and exponential complexity. In general, the heuristic algorithms for problem whose known exact algorithms are exponential are polynomial.

Transformations and Reductions

We've already seen that problems can be transformed into other problems, where the idea is always is to take an instance of problem P I_p, transform it to an instance of problem Q I_Q using some time, solve it with an algorithm A_Q to obtain S_Q and transform it back to S_P using some time. This can be done in one or several iterations. The point is that if you have a transformation in which the modification of the instance is polynomial and the number of time that the transformation has to be applied is polynomial (or constant, like 1), then the complexity of the overall algorithm depends on the complexity of A_Q[1].

3.2 Beyond the Worst-Case complexity

What can we do to go beyond the standard definition of worst complexity? It has many disadvantages as it considers only the insance of big size, avoiding the smaller instances, ignoring some informations. What are the easy instance and how are these made? We would like to know both if the instance being considered is "easy" or not. Secondly, the wors-case complexity gives an extimate which can sometimes be overly rough, even though usually it's meaningul; there are however cases in which they aren't, such as linear programming, in which problems have inifinite solutions but finite basic solutions, and admit an algorithm which has exponential

[1] Of course the complexity can change by being asymptotically increased by the cost of transformation, but the idea is that if A_Q is polynomial then it remains polynomial.

complexity. However, in the extremely large majority of cases the complexity of that algorithm is a polynomial of very low complexity.

So, some problem's worst-case is badly extimated. What can you do to improve you knowledge? There are two points to be investigated:

- **Parametrized Complexity**, that is introduce some other releveant parameter k besides n and espress the time as $T(n,k)$, which may lead to the understanding that the algorithm is hard with respect to n but is easy with respect to k.
- **Average-Case Complexity**, that is to assume a probability distribution on $\mathfrak{I}$ and express the time as the expected value

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

3.2.1 Parametrized complexity

The idea is that some algorithm are actually not exponential in n but are exponential in some other parameter k which can be smaller than n. If both k and n are large, then obviously the algorithm is inefficient, but if k is small, then the algorithm will be efficient and possibly even polynomial. The parameter k can be a part of the input like a capacity, the maximum number of literals per formula in logic function problems, the number of nonzero elements in numerical matrix problems, the maximum degree, the diamater and so on in graph problems. In these cases, it's possible to know *a priori* whether the algorithm is efficient on a given instance.

Strangely, the additional parameter k could be part of the solution: in such case you can't know a priori if the algorithm is efficient or not, but only an estimate could be available.

Example: the Vertex Covering problem

Let's briefly remind what is the VCP: given a graph $G = (V,E)$ the VCP requires to find the minimum subset of vertices that cover every edge of the graph. This problem has an exhaustive algorithm: for each of the 2^n subsets of vertices, test if it covers all edges, compute its cardinality and keep the smallest one. This algorithm has time complexity

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

where $n = |V|$ and $m = |E|$. But if it's already known that a solution exists with $f(x) = |x| = k+1$, we can look for a solution of k vertices and progressively decrease k, or even better use binary search on k. In particular, a simple and naive algorithm consists in finding a solution with exactly k vertices for a fixed k. This algorithm has time complexity

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

where the n^k is an approximation of $\binom{n}{k}$. For each of the $\approx n^k$. If k is small, it's better than an exponential time, but if k is large it's worse. We can do better.

Bounded Tree Search for the VCP A better algorithm can be based on the following useful property:

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

or, in words, any feasible solution includes at least on extreme vertex for each edge. In order to find a certain solution, the bounded tree search works by choosing and edge (u,v) so that

either u or v (or both) will be contained in the solution. The problem will be then split in two possibilities: the first has $u \in x$ and the second has $u \notin x$, so $v \in x$. In both cases, to check if a solution is found, it's necessary to compute

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

removing all the vertices in the partial solution and all the covered edges. If $|x| \leq k$ and $E = \emptyset$ then x is the required solution, if $|x| = k$ and $E \neq \emptyset$ there is no solution; otherwise, start again by choosing some edge $(u,v) \in E$. The complexity is $T(n,m,k) \in \Theta(2^k m)$ polynomial in n, as $m < n^2$. For $n >> 2$, this algorithm is much more efficient than the naive one.

Kernelization The **kernelization** refers to the idea of transforming or reducing problems to other problem: in this context the kernelization transforms all instances of P into simpler instances of P. To achieve this, useful properties allow to prove that exists **an** optimal solution **not including** some elements of B and surely **including** certain elements of B. So, kernelization removes elements of B withouth affecting the solution. The idea you obtain instances that are smaller and smaller and, if you're lucky, in the end the instance will have a size no longer depending on n and that can be solved exactly. Still, if that is not the achieved result, there are ways to improve the algorithm by running a heuristic algorithm on a smaller problem, which will be faster and probably better. Also, in principle, the kernelization *itself* could be heuristic, meaning that the vertices chosen to be excluded and those kept are not certainly the "right" ones but are heuristically chosen.

For the **kernelization of the VCP**, this problem has a property which states that if $\delta_v \geq k+1$ (δ_i is the degree of vertex i), the vertex v belongs to any feasible solution of value $\leq k$.

This is proven by contradiction: if you assume that a node has large degree ($\geq k+1$) and it's not part of any feasible solution of size k, it means that feasible solution must cover the $k+1$ edges in another way, but the only way to cover these edges that have a single common vertex is to use $k+1$ vertices. But a solution can't have $k+1$ vertices in it, so this is impossible. By contradiction, this vertex must belong to the solution. So, the steps of the kernelization algorithm are as follows:

- start at step $t = 0$ with $k_0 = k$ and an empty vertex subset $x_t := \emptyset$
- set $t = t+1$ and add to the solution the vertices of degree $\geq k_t + 1$, so, formally $\delta_v \geq k_t + 1 \Rightarrow x_t := x_{t-1} \cup v$.
- update $k_t := k_0 - |x_t|$
- remove the covered edges, the vertices of zero degree and the vertices in x, $E := \{e \in E : e \cap x_t = \emptyset\}$ and $V := \{v \in V : \delta_v > 0\} \setminus x_t$.
- if $|E| > k^2$ there is no feasible solution, and if $|E| \leq k^2 \Rightarrow |V| \leq 2k_t^2$ apply the exhaustive algorithm.

The kernelization algorithm is trivial, starting with a guess or heuristic on k. At first, all the vertices with degree larger than $k+1$ are sought: if any is found, it's added to the solution x - but at this point less than k vertices are to be found (exactly $k - |x|$) so it's necessary to update k, which will be the new threshold for the degree of vertices. The interesting part is that the property changes with each iteration, for each bulk of vertices added to the solution. After "cleaning" the Vertex and Edges sets, it's needed to check if the number of remaining edges is larger than k^2 (with k updated till that step): if it's larger than k^2 there's no possibility to cover

the remaining edges with less then k vertices, as these k vertices have at most k degree, and there's no feasible solution. Otherwise, if the number of remaining edges is less or equal to k^2, this means that there are less or equal to $2k_t^2$ vertices. The complexity is $T(n,k) \in Theta(n + m + 2^{2k^2}k^2)$, which is good if k is small.

3.3 Average Case complexity

Instead of characterizing by second parameters, the worst-case complexity can be replaced with the **average-case** complexity, that is instead of computing the $\max T(n)$, it's computed

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

the expected value of $T(n)$. This requires to define a **probability** on the instances and this requires a probabilistic model - that's not easy at all. So, that depends on the approach: in the context of theoretical studies, some theoretical probility distribution model will be introduced in order to get some theorems, saying that if that model is used the time complexity will be something. This can be useful as it can be found empirically that instances follow some kind of distribution. In other cases, data and benchmarks can be available, from which an empirical evaluation of the probability distribution can be drawn and a simulation model can be built: other random realistic instances can then be used to test the algorithm on other instances, measuring the time required.

3.3.1 Probabilistic Models for known problems

Probabilistic Models for Random Binary Matrices

What can the probabilistic model be for binary matrices? There are three main probability models: the first one is the **equiprobability**, where all of the possible 2^{mn} matrices have the same probability to appear. The equiprobability is a possible model only if the set of matrices is finite. In the **uniform probability model** each cell of a matric is set to 1 with a given probability p

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

which is a slightly more general model compared to equiprobability. The last model is the **fixed density model**, where a parameter $\delta \in [0,1]$ is defined and that out of the mn cells, $\delta \cdot mn$ are extracted with uniform probability and set to 1. It resembles the uniform probability, but in this case the (max) number of 1s is exactly fixed.

Probabilistic Models for Graphs

Since a Graph admits an adjacency matrix as its representation, the models are almost the same. In the **equiprobability model** the $2^{\frac{n(n-1)}{2}}$ graphs have the same probability to be selected. In **Gilbert's model** (or **uniform probability model**) the probability $G(n,p)$ is defined as

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

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

Finally, the **Erdős-Rnyi model**, in which the probability $G(n,m)$ is defined as follows: extract m unordered vertex pairs out of the $\frac{n(n-1)}{2}$ with uniform probability and create an edge

for each one. If $\frac{2m}{n(n-1)} = p$, it resembles the uniform probability model, but the number of edges is fixed.

Probabilistic Models for Logic Functions (Random CNF)

A CNF is a conjunction of disjunctions (ANDs of ORs). Logical formulae have a certain number n of variables and a given number of literals k for each disjunction (called logic formula) - let's assume k is the same for each logic formula. The **fixed-probability ensemble model** lists all $\binom{n}{k}2^k$ (there are $\binom{n}{k}$ ways to dispose n variables in groups of k, and for each group there are 2^k possible truth assignment to its literals) formulae of k distinct and consistent literals and adds each one to the CNF with probability p. The **fixed-size ensemble model** builds m formulae, adding to each oone k distinct and consistent literals, extracted with uniform probability p - something like the opposite process than before, starting from a fixed number of formulae and constructing them with variable size; before we had fixed size but variable number of formulae. Furthermore, if $p = \frac{m}{\binom{n}{k}2^k}$, it resembles the fixed probability model, but some instances cannot be generated.

3.3.2 Phase Transitions

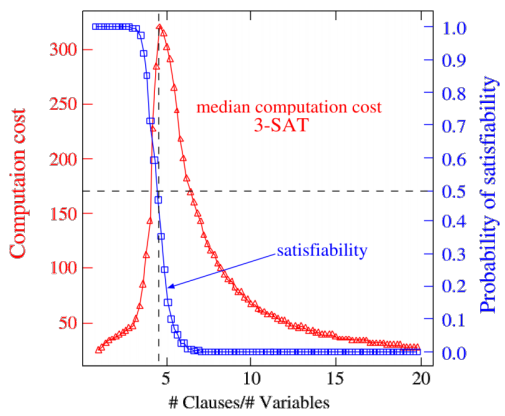
The purpose is to define some kind of families and to study the performance with respect to the computational time complexity on a specific family of problems. So, what comes from this studies is the concept of **phase transition**: it has been observed that introducing deterministic or probabilistic parameters then what is obtained is different regions of the instance set (just the same as what happened with parametrization) and some of these regions will prove to require a lot of time and some region will prove themselves being easier. For example, for graphs, $m = 0$ and $p = 0$ correspond to empty graphs, $m = \frac{n\cdot(n-1)}{2}$ and $p = 1$ correspond to complete graphs and intermediate values correspond to graphs of intermediate density, deterministically for m and probabilistically for p - algorithms are expected to behave differently on different regions.

For many problems the performance of algorithms is strongly different in different regions, concerning the computational time (for both exact and heuristic algorithms) and the quality of the solution (for heuristic algorithms). The interesting point is that the variation in the performance variates steeply and abruptly in small regions of the parameter space: this phenomenon is similar to the phase transition of physical systems, for example when ice melts into water, which happens at very strict temperature and physical zone. It's quite interesting because a new instance can be analyzed in order to understand in which "region of complexity" it lies.

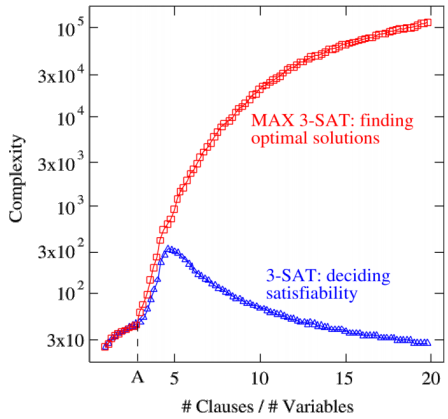
3-SAT and MAX-3-SAT

The first example of phase transition concerns the satisfiability problem. Given a CNF on n variables, with logical formulae containing 3 literals, the 3-SAT is solved by finding a truth assignment which satysfies all formulae, while the MAX-3-SAT is solved by finding the maximum number of satisfiable formulae. So, the parameters are the number of variables n and the number of literals for each disjunction, which is 3. Another important parameter is $\alpha = \frac{m}{n}$, the ratio of formulae and variables (so m is the number of formulae in the CNF).

As Figure 3.1 (a) shows, for low values of α the probability of a positive solution (the CNF is satisfiable) is high and the computational cost of finding this answer is low. But as α (so the number of formulae) increases, the problems becomes harder to solve and its probability to be satisfied quickly lowers - in fact, having lots of formulae and few variables quickly induces



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



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

Figure 3.1: Phase plots.

some counterdiction. Figure 3.1 (b) instead plots the computational costs of the two version of the problem, showing the cost of the 3-SAT problem in scale. The optimization version (Max-3-SAT) is harder because it's not sufficient to say that the CNF is unsatisfiable, it's to find how many of the disjunction are satisfiable at max.

The practical value of these information is that since it's algorithm-agnostic, it's sufficient to calculate α in order to know approximately where the instance lies.

Vertex Covering Problem

The VCP exhibits a somewhat similar phase transition, as Figure **??** shows. This time, the measured value is $\frac{|x|}{|V|}$, the ratio of the cardinality of the solution on the total number of vertices in the graph. This is *a posteriori* value and can't be evalued in advance, since x is unknown. If the ratio is small, you need a small number of vertices to cover all the edges and these vertices can be found quickly. on the contrary, if they start to be around 30% of the total number of vertices, finding them is increasingly difficult as the size of the instance increases. If the number of vertices again larger, around 40% of the total vertices it will become easier to find them.

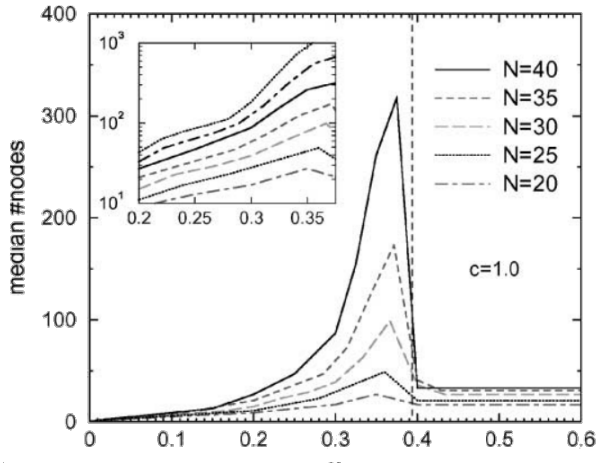


Figure 3.2: VCP Phase.

It's true that heuristic algorithms usually don't have a steeply varying time complexity, but actually they still have some dependency on the structure of a problem, even though they're probably always polynomial and not order of magnitudes different. It must also be observed that metaheuristics will use random steps or memory, hence making the concept of "termination" of an algorithm quite unclear, as it can be possibly go on many times renewing random seeds or using past results stored in memory - in theory going on forever; still, observing a single iteration of an algorithm, this is still valid. It's also useful to know this as this is useful to know if an instance can be easily solved exactly with some algorithm; that kernelization can be used to improve the complexity and results of heuristic algorithms; that heuristic algorithms can be used to improve on exact algorithms and finally that knowing a priori or during executing the region where an instance lies might be hard. The main point is that the dicotomy between exact and heuristic algorithms is as vague and smooth as possible, they're not opposite approaches.

CHAPTER 4

Theoretical Effectiveness

4.1 Theoretical Performance Evaluation

We've seen that a heuristic algorithm is efficient if it has a computational complexity - a cost - much lower than an exact algorithm for the same problem. Now, it's time to consider the **effectiveness** of a heuristic algorithm: as said before, a heuristic algorithm is effective if it "frequently" gives a "close" solution to an exact or optimal one. We will, now, discuss the meaning of "frequently" and "close".

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

4.1.1 Distance between solutions

As said before, we need a mean to measure distance between solutions, and we can give multiple interpretations regarding this concept. Let's begin with some context: the effectiveness of a heuristic optimization algorithm A is measured by the **difference** between the heuristic value $f_A(I)$ (the objective function on the instance I as computed by A) and the optimum $f^*(I)$, the best solution existing for I.

Absolute difference

A possible initial definition is the difference between the two values:

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

while this definition is quite natural and obious, it's rarely used in practice as it's depends on the unit of measure of the objective function: just think you're tring to minimize an overall time in the TSP and you're measuring the time in days. You could choose hours or minutes on seconds instead and the value of $\widetilde{\delta}_A(I)$ will strongly depend on this choice. This will be valid when the objective function is a pure number.

Relative difference

The second definition is based on the idea of computing the relative difference with respect to the optimum. also in this case it's to be expected that if

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

This is quite frequent in experimental analysis and it's the advantage that if the objective has a unit of measure, then the ratio is a pure number.

Approximation ratio

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

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

which includes both the minimization and the maximization problem: if it's a minimization problem, probably $f^*(I) < f_a(I)$ and the opposite for maximization problems. It's clearly related to relative difference: in fact, for minimization problems,

$$\delta_A(I) = \rho_A(I) - 1$$

and a relation exists for maximization problems as well.

4.1.2 Theoretical Analysis: the approximation guarantee

In order to guarantee a priori the performance of an algorithm, the idea is once again to consider the worst case, exactly as for the efficiency. In general, when you apply an heuristic the result $f_A(I)$ may possibly be very bad with respect to the optimal solution $f^*(I)$, but if the algorithm is good, the difference won't be that large; for example there may be an absolute approximation:

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

meaning that the absolute distance is somewhat limited by an integer constant; a rare example of this if Vizing's algorithm for Edge Coloring. Since it's hard to find valid absolute approximation, what's actually done in practice is to try to use a **relative approximation**:

$$\exists \alpha_A \in R^* : \rho_A(I) \leq \alpha_A \ \ \forall I \in \mathfrak{I}$$

meaning that the approximation ratio $\rho_A(I)$ has an upper bound delimited by the real constant α_A.
If any of the previous can be used, it's said that there's an **approximation guarantee**, absolute or relative. The interesting thing is that the definition can be extended to cases in which a constant approximation guarantee cannot be found by considering the approximation not a value but a suited function paramterized by the instance size:

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

In conclusion, it's important to underline that while the computational complexity is necessarily size-dependent, the efficiency may be independent.

How to obtain an approximation guarantee

Given the theoretical basis, we can now describe a general way to introduce and prove an algorithm with an approximation guarantee. A general abstract scheme is the following: what we want to prove (for minimization problems) is that

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

exactly like saying that $\rho_A \leq \alpha_A$. In order to prove this, the first step is typically to start with the instance and generate a lower bound $LB(I)$ by analyzing the problem

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

- an underestimate of the objective function. Once this value is found, the second step is to build an overestimate $UB(I)$ related to $LB(I)$ by a coefficient α_A:

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

Then, the third and final step is to find an algorithm A whose solution is not worse than $UB(I)$ for any instance

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

Then, if these three steps are done and suitable values and algorithms are found

$$\begin{aligned} f_a(I) \leq UB(I) = \alpha LB(I) \leq \alpha f^*(I) \ \ \forall I \in \mathfrak{I} \\ \rightarrow f_A(I) \leq \alpha_A f^*(I) \ \ \forall I \in \mathfrak{I} \end{aligned}$$

The trickiest part might be the second step, so let's delve into it by showing an example.

Approximation Guarantee for the VCP: 2-approximated algorithm Given an undirected graph $G = (V,E)$ find the minimum cardinality vertex subset such that each edge of the graph is incident to it. Let's define a set of nonadjacent edges as **matching**, so that we can in turn define a **maximal matching** as a matching such that any other edge of the graph is adjacent to one of its edges (it cannot be enlarged), which does not mean that it's the *maximal*.

We can then construct the *matching algorithm* to solve the VCP using matching:

1. build a maximal matching $M \subseteq E$ scanning the edges of E and including in M those not adjacent to any edge already in M
2. the set of extreme vertices of the matching edges is a VCP solution $x_A := \bigcup_{(u,v)\in M}\{u,v\}$ and it can be improved removing the redundant vertices.

Let's now turn to the approximation guarantee. A possible lower bound is to include in the solution just *one* of the vertices of the edges in the maximal matching: consider only the edges in the maximal matching M. It's clear that in a graph consisting in only the edges $e \in M$ ($G' = (V,M)$) only *half* of the vertices included in the solution x_A cover the edges in M. In other words, the optimal vertex cover of the edges in M is certainly smaller than the optimal vertex cover of all the edges in E. To give a numerical example, suppose that for a certain graph the VCP has an optimal solution $f^* = 5$ (5 vertices can cover al its edges) and the matching algorithm finds a maximal matching M with 3 edges, so that $f_A = 2 \cdot |M| = 6$. To cover all the edges in M, $\frac{|M|}{2} = 3$ vertices are enough!

This value, $\frac{|M|}{2}$ is a possible lower bound for our problem:

$$LB(I) = \frac{|M|}{2} \leq f^*(I)$$

but there's more: we know that x_A as calculated before is actually a feasible solution and it's related to $LB(I)$:

$$f_A = 2 \cdot |M| = 2 \cdot LB(I) = UB(I)$$

So, we've found both the upper and the lower bound. Let's formally give the proof that the **matching algorithm is 2-approximated**:

Proof. Let's first prove that the cardinality of matching M is an underestimate $LB(I)$: the cardinality of an optimal covering for any subset of edges $E' \subseteq E$ does not exceed that of an optimal covering for E

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

and the optimal covering of any matching M has cardinality $\frac{|M|}{2}$, so

$$|M| \leq |E| \rightarrow |x^*_M| \leq |x^*_E| \rightarrow \frac{|M|}{2} \leq |x^*_E| \ \ \forall M$$
$$\therefore LB(I) = \frac{|M|}{2} \leq |x^*_E| \ \ \textit{is a valid lower bound.}$$

In words, since any M has a cardinality less or equal to that of E, the optimal solution (which has the exact cardinality of $\frac{|M|}{2}$) has a cardinality lower or equal to that of the optimal solution of E, making it a valid underestimate.

The fact that $|M|$ itself consists in an overestimate is proven by the fact that by definition the vertices of the edges in M cover all the edges in E, meaning that the set of vertices in M consists in a feasible solution (which by itself means that it's an upper bound to the optimal value) and it covers both the matching and the adjacent edges. Therefore, $|M| = UB(I) = 2LB(I)$ is an overestimate.

The proof is concluded by the fact that the algorithm returns solutions of value $f_A(I) \leq UB(I)$, since in M there may be reduntant vertices that can be removed. This implies that $f_A(I) \leq 2f^*(I) \ \ \forall i \in \mathfrak{I}$, that is $\alpha_A = 2$. □

The tight bound between upper and lower estimates

Are there instances $\bar{I}$ for the VCP such that $f_A(\bar{I}) = 2f^*(\bar{I})$ using the matching algorithm? More in general, are there instances $\bar{I}$ such that

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

exactly and if it's so, how are the $\bar{I}$s like? In other words, after having proven that $f_A(I)$ is related to $f*(I)$ through α_A, is this constant precise and valuable or is it "just" an upper bound? The study of the instances $\bar{I}$ is useful to evaluate whether they are rare or frequent and to introduce ad hoc modifications to improve the algorithm.

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

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

and

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

This problem is, in general, strongly NP-Complete. But it's even strongly NP-Complete to understand if the instance *has* a feasible solution (not *which it is*) - if there's an Hamiltonian circuit visiting all the nodes. Under the previous assumptions an heuristic algorithm can be built with an approximation guarantee associated to it.

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

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

The structure is the same as for the VCP: an underestimate is found together with a multiplication factor to construct an upperbound and then it's proved that an heuristic algorithm can be dominated by this upperbound, which means that the heuristic algorithm is at most a certain factor multiplied by the optimum.

Let's directly prove that the **double-tree algorithm is 2-approximated**.

Proof. Deleting an arc from a hamiltonian circuit yields a hamiltonian path, which is necessarily cheaper. Since an hamiltonian path spans all nodes, it's in fact a spanning tree, and usually not of minimum cost. So, an actual minimum spanning tree is surely a lower bound on the cost of the Hamiltonian circuit. The series of nodes of a Depth-First visit of the MST together with its backtracking generates an Hamiltonian circuit in the original graph, as there's a pair of directed edges between each nodes (so each visited edge in the DFS exists in the original graph). The cost of such Hamiltonian circuit is necessarily an overestimate since the cost of the paths in the graph respect the triangular in equality: while the visit yields, for example, the path $A \to B \to C$, choosing the edge $A \to C$ makes the solution lower in cost. So, since $LB(I) =$ cost of the MST and the forementioned circuit has exactly double the edges of the MST, $UB(I) = 2LB(I)$.

The heuristic is dominated by the upper bound and any circuit x has

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

□

4.1.3 Inapproximability

For an **inapproximable problem**, all approximated algorithms are exact - some problem cannot be approximated unless some property of computational complexity turns out to be true, which is extremely unlikely to happen. For example, consider a family of TSP instances violating the triangular inequality in the following way:

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

and the graph is complete. If any solution of null cost ($f^*(I) = 0$) is found, it means that the solution contains only edges in A_0 composing the Hamiltonian circuit. So, in a sense, there's a non-complete graph $G(N, A_0)$ that has a feasible Hamiltonian circuit solution which is the same as the one for the original graph - this means that the original graph is the "completion" of the new graph. This is nice as if a solution for the complete graph is found with null cost you can actually solve the TLP in decision form on the non complete graph, which is a strongly NP-Complete problem. This can't be done exactly, unless $P = NP$, which is unlikely. The objective function of any such instance $\bar{I}$ is

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

We **can't** find an approximation guarantee on the optimum α for a polynomial algorithm so that

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

because assuming that it's possible then what happens is that if the graph has an hamiltonian circuit of zero cost, necessarily the approximation algorithm will yield a cost

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

in words, the approximated algorithm solves the decision version of the TLP on the non-complete graph instance, thus solving a strongly NP-Complete problem in polybomial time and proving that $P = NP$.

Since that this is extremely unlikely as said before, the TLP is said to be inapproximable.

4.1.4 Approximation schemes

On the bright side, sometimes it's possible to find something even better than just *approximation*: for any hard problem there is the exhaustive algorithm providing the best approximation guarantee $\alpha_A = 1$ (since it's exact) which requires exponential time T_A. Approximated algorithm usually provide a worse guarantee ($\alpha_a > 1$) but they may require polynomial time T_A.

Now, there can be a lot of *stuff* in the middle, meaning that instead of a single polynomial algorithm with time T_A and guarantee α_A there may be a complete family of "ranges" of compromises between efficiency and effectiveness, increasing in computational complexity and improving approximation guarantee. Such a family of algorithm is called an **approximation scheme**, which is a parameteric algorithm A_α, allowing to choose α. An example of this situation is the Knapsack Problem.

4.2 Beyond Worst-Case approximation

Just as it was for the efficiency, effectiveness can be "measured" using other references than the wors-case. You may have a bad worst-case with no approximation or an approximation $\alpha = 1000000$ and then you run your algorithm and it always yields the optimal solution. Apart from very rare cases, how can we describe this situation? There are two alternative approaches similar to those used for complexity and a third which is not shared with the complexity.

4.2.1 Parametrization

Instead of deviding instances by size, other parameters k_i can be identified so that can be provided an approximation guarantee which depends on the k_i and not constant.

4.2.2 Average-Case

Another possibility is studying the average case of the instance, that is: after assuming a probabilistic model for the instances, the expected value can be evalued of the approximation factor. Once done, then an empirical approximation guarantee can be retrieved.

4.2.3 Randomization

This approach is not that meaningful when used regarding efficiency, the computational time: this approach is known as randomization. Beyond heuristics there are metaheuristics: some use memory and some use randomization. The latter is practically realized by giving a random seed as input to the algorithm together with the instance: this random seed is used in a PRNG to generate other numbers which will be used to take decisions inside the algorithm; this means that changing the random seed different solution will be obtained, and the stochastic distribution can be characterized given the random seed instead of the instance. This is interesting because the input to the solution becomes a random variable whose properties can be studied.

What happens is that the algorithm is run on the pair $\langle I, \omega \rangle$, the instance and the random seed, yielding $f_A(I, \omega)$. The approximation ratio is as well a function of ω as well: $\rho_A(I, \omega)$ so both are random variables; the expected value of the approximation ratio is limited by a value:

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

This means that if running the algorithm several times and averaging the approximation ratio for this results what's obtaines has an expected value which is at most a given number.

Randomized Approximation Algorithm for Max-SAT

Given a CNF, we want to assign values of truth to the logical variables so to maximize the weights of the logical formulae that are satisfied. A possible random algorithm is to extract a random value for each variable with probability $\frac{1}{2}$: what is the expected value of the final solution? Let $\mathcal{C}_x \subseteq \{1, \cdots, m\}$ be the subset of formulae satisfied by solution x. Then, the objective value $f(x) = f_A(I)$ is the total weight of the formulae in $\mathcal{C}_x$ and its expected value is

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

The value $Pr[i \in \mathcal{C}_x]$ is a number in $[0, 1]$ (since it is a probability) and it's to be computed or estimated, somehow. In order to estimate it, suppose that the formula $i \in \mathcal{C}$ has exactly k_i

literals, and let $k_min = \min_{i \in \mathcal{C}} k_i$ the minimum number of all the literals over all the clauses. The probability of satisfying a certain clause i is

$$Pr[i \in \mathcal{C}_x] = 1 - (\frac{1}{2})^{k_i} \geq 1 - (\frac{1}{2})^{k_{min}} \quad \forall i \in C$$

a minorization of the real probability, independent of i. This means that

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

and since $E[\rho_A(I)] = f^*(I)/E[f_A(I)]$ and $f^*(I) \leq \sum_{i \in \mathcal{C}} w_i \quad \forall I \in \mathcal{I}$ since the maximum optimum for Max-SAT is the sum of all weights, and one can obtain

$$E[\rho_A(I)] \leq \frac{1}{[1 - (\frac{1}{2})^{k_{min}}]} \leq 2$$

since k_{min} is at least 1 and $\frac{1}{\frac{1}{2}} = 2$. It's important to remember that this is an approximation guarantee on the average, which means that in each single run you could be very unlucky - even for a long sequence of runs; but if you make many runs (which you can, since it's very simple) then the sample mean will tend to get near the theoretical expected value. The best will clearly be better than that!

CHAPTER 5

Empirical Performance Evaluation

After considering the *a priori* theoretical analysis of algorithms, which consist in proving some guarantees on some aspects of the algorithm being studied such as the computational cost of the quality of the solutions, it's now possible to consider the *a posteriori* analysis of algorithm. This is because a priori analysis is complicated by several aspects: while the efficiency analysis is usally rather simple, the analysis of the quality is way more difficult as the "steps" of the algorithms do not have an immediate relation to the quality itself. Moreover, in the average case or if it's needed to take account of parameters such as random seeds, statistics are needed and this is usually far from being simple. Even if theoretical analysis is useful, its results may be not very representative in practice if the assumptions are not met by the instance used in practice: for example, if the used worst case is not representative of what actually happens the analysis will give a high computational cost which is not what is observed in practice.

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

5.1 Introduction to Experimental Analysis

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

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

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

5.1.1 Model

But what is a *model* in the context of algorithms? In physics a model is a law that rules the behaviour of phenomena and, in analogy, in algorithms a model is a law that's supposed to rule the behaviour of the algorithm itself; laws may be such as assuming that the algorithm will be linear in computational complexity, will depend on some other parameter such as the maximum degree of the vertices of the graph and so on. This is something to be assumed based on the knowledge about the algorithm but also on a *trend* exposed by the empirical approach.

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

5.1.2 Benchmark

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

An important and frequently overlooked aspect is that it would seem natural to introduce the concept of **equiprobable benchmark**, meaning that given an infinite set of instance any of them is extrated following a uniform probability. It wouldn't make sense to use this approach because infinite sets **do not admit equiprobability**[1]. Just think about the fact that given a finite benchmark sample, then the largest (in term of size) instance will not represent the *largest* of all the instances in the benchmark, so an infinite part of the instance set that's totally unrepresented in the benchmark. On the contrary, we can build a finite benchmark by defining some finite class of instance, for example "all graph instances with 100 to 10000 vertices". The instances must not be *too easy* to solve as it's needed to learn something from them and have a structure that matches a practical application, if it exists.

Reproducibility

Scientific method require **reproducible** and **controllable** results. So, if some investigation is made, it's necessary to make sure that other people are able to replicate the experiment. So, it's necessary to use publicly available instances (or make them publicly available), specify the implementation details, the programming language and the compiler of the implemented algorithm and environmental values such as the machine used, the OS and the avaiable memory.

[1] This is a big statistic question, but this is not a statistic course, hence no proof will be given.

5.2 Comparing Heuristic algorithms

A "big rule" in order to compare algorithms is that given two algorithms the first is better than the other one only it obtains betters results and requires a smaller time **simultaneously**. Slow algorithms with good results and fast algorithms with bad results cannot be compared in a meaningful way. In some specific situation computational time can be neglected, precisely when no comparison is to be made but just some estimation is sought and when it's known that the computational time is more or less the same for some specific structural reasons, e.g. two algorithms which are the same but with some difference in numerical parameters.

5.2.1 Describing the performance of an algorithm

In order to describe the performance of an algorithm a **statistical model** can be given. In particular, the idea is to model an algorithm as if it is were a random experiment in which the results depend on the choice of an object inside the *sample space*, defined as the sample of instances

$$\bar{\mathbb{I}} \subset \mathbb{I}$$

Chosen a specific instance $I \in \bar{\mathbb{I}}$ the obtained results are the computational time $T_A(I)$ and the relative difference $\delta_A(I)$. Both are random variables and their performance describe the performance of the algorithm A. What is to be done is, instead of considering all the possible values of computational time and all the possible values of the relative difference, describe the random variables $T_A(I)$ and $\delta_A(I)$ with their statistical properties, which will give a statistical description of the performance of the algorithm.

Characterizing $\delta_A(I)$ means that the optimum is known, and this usually not the case. How can we use it if we don't know $f_A^*(I)$? First of all we could have another algorithm that could give the optimum, therefore describing an algorithm in terms of the optimum found by another algorithm. Otherwise, it's necessary to estimate it from below or above (depending if it's a minimization or a maximization problem) and it's needed to have the best possible estimated. If in the definition is used an estimation, what it's obtained is an *estimation* of the relative difference.

5.3 A Posteriori Efficiency Evaluation

5.3.1 Run Time Distribution diagram

Let's start with the description of the performance with respect to the computational time. We're trying to make an empirical analysis of the efficiency of the heuristic algorithm. The main tool is the so-called **run time distribution** (RTD) diagram, which plots the distribution function of the required time to solve an instance $T_A(I)$ in the benchmark sample $\bar{\mathbb{I}}$. Of course, it depends on benchmark and bear in mind that it needs to be meaningful.

This is a cumulative distribution function and as always is described as

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

in words it's probability that the random variable $T_A(I)$ has a value smaller than a parametric value t for all possible t's. Now, in the case of a computational time it's clear that it's always positive, so for all $t < 0$ the probability is 0. But as t grows the probability grows itself from 0 to 1.

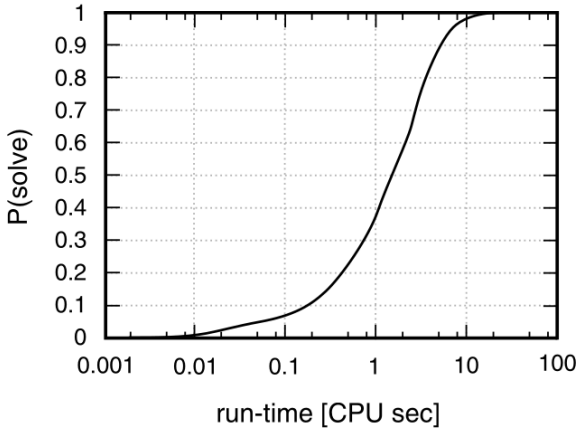


Figure 5.1: An example of RTD diagram.

Let's examine Figure 5.1 as an example to give the practical meaning of this profile. This profile goes from probability 0 of solving an instance in a time smaller or equal to 0.001s (a thousand of a second) to probability 1 for a time slightly larger than 10s. This means that on the given benchmark the computationl time is between 0.001 and slightly more than 10s. In particular, considering $t = 1$, the fraction of instances that are solved by the algorithm in a time smaller or equal to a second is exactly given by the vertical coordinate $\approx 35\% = 0.35$. So, about $\frac{1}{3}$ of the instances are solved in less than one seconds while about $\frac{2}{3}$ are solved in a time between one second and close to twenty seconds.

The time axis is in logarithmic scale as the regular steps are ten times the one before, so actually the computational time ranges on huge possible set of values; the probability is obviously increasing as the number of solved instances cannot decrease with the growth of time.

Of course computational time **strongly depends** on the size of the instances and actually it's rather stupid to use such a profile on a benchmark that uses instances of very different sizes: this is because if the algorithm uses a small time to solve small instances and a large time to solve large instances what the plot depends on the choice of the benchmark rather than the algorithm itself. The solution is to *fix* the size using a benchmark sample $\bar{\mathbb{I}}_n$ with fixed n, for example saying that the benchmark is composed of graph with exactly 1000 vertices, then study the profile again. If the profile is the same then it's more interesting as even for a static size some instances are harder to solve than other; there may be some other parameter influencing the algorithm efficiency (such as density for graph) so the benchmark sample can be constructed by fixing that parameter as well, creating $\bar{\mathbb{I}}_{n,\Delta}$ and so on. If all influential parameters are identified and fixed, the RTD diagram degenerates into a step function, meaning that all instances are solved requiring the same time.

Building an RTD

The RTD is certainly monotone and nondecreasing. Furthermore, as Figure 5.2 shows, it's acually a stepwise and right-continuous (so discontinuous) function. This is because it "goes up by steps", simply because corresponding to each time required by each specific instance, one or more instances with the same time are solved. In other words, between $t \in [0.001, 0.01)$ no instances are solved, so $P(solve) = 0$, but at $t = 0.01$ some instances are solved, so that for $t \in [0.01, 0.1)$ $P(solve) = 0.05$. If exactly one instance is solved in this span of time, then the number of instances in the subset can be inferred by dividing the height of each step by 0.05

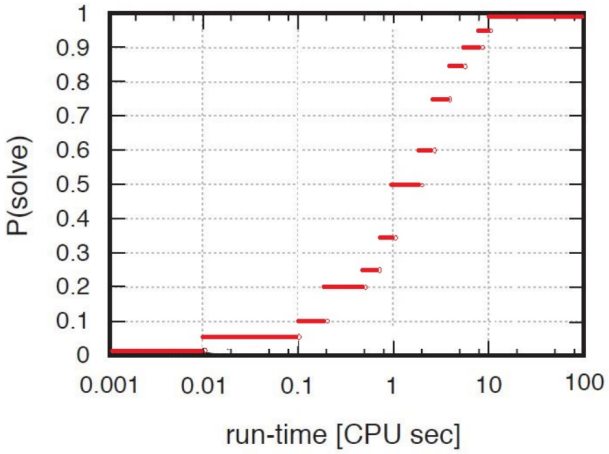


Figure 5.2: Construction of an RTD plot.

and summing them all together: for example, as $P(solve) = 0.1$ for $t = 0.1$ is double than the previous step, this could mean that the number of solved instance is doubled, and if it was one in the previous step it's two now. This can be done for all steps: at $t \approx 0.2$ $P(solve) = 0.2$, and as $\frac{0.2}{0.05} = 4$ this could mean that 4 instances are solved at that time.
In order to build it this steps are to be followed:

1. run the algorithm on each instance $I \in \bar{\mathbb{I}}$
2. build the set $T_A(\bar{\mathbb{I}}) = \{T_A(I) : I \in \bar{\mathbb{I}}\}$
3. sort $T_A(\bar{\mathbb{I}})$ by nondecreasing values $t_1 \leq \cdots \leq t_{|\bar{\mathbb{I}}|}$
4. plot the points $(t_j, \frac{j}{|\bar{\mathbb{I}}|})$ for $j = 1, \cdots, |\bar{\mathbb{I}}|$, choosing the maximum one in case of conflicts.

5.3.2 Scaling diagram

The previous analysis makes sense only once the size and all the obvious parameters are fixed, but now a description of the depencency of the runtime with respect to those parameter is needed. This is done using a diagram called the **scaling diagram**, which represents how the computational time scales with respect to the size (or other parameters) of the instance, so the dependence of the $T(I)$ on the size $n(I)$ is described.

The diagram is done by generating a sequence of values for the size n and a benchmark sample $\bar{\mathbb{I}}_n$ for each one. Then, the algorithm is to be applied to each $I \in \bar{\mathbb{I}}_n$ for all n and the points $(n(I), T(I))$ (or the arithmetic mean points) can be plotted. Once this is done, an **interpolating function** is to be assumed. The idea is this analysis is different from the theoretical analysis in that it gives an empirical average-case complexity and the multiplying factors are analitically determined and not neglected; it's quite obvious that this empirical complexity must relate in some way to the theoretical complexity, and what's to be estimated here has to be better than the worst case analysis. If it's found otherwise, then either the theoretical analysis is wrong or there's something wrong in the implementation.

Choosing the interpolating function

The interpolating function can be found by theoretical analysis on the algorithm or by graphical manipulation on the plot, such as the use of logarithmic scales on one or both axes: this is

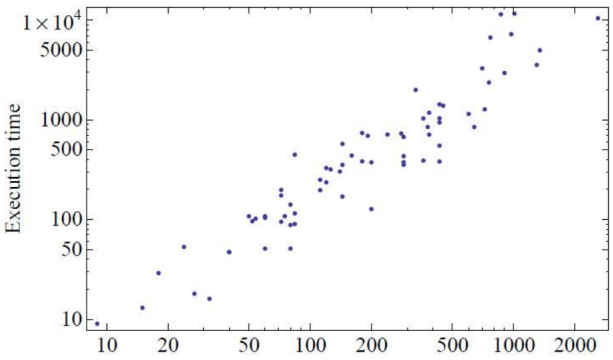


Figure 5.3: An example of a scaling diagram.

because our eyes cannot naturally interpret functions which are not linear which precision. Let's give an example of the second case. If the algorithm is exponential, then keeping the regular axis on the horizontal scale and a k-logarithmic scale on the vertical axis allows to say that

$$log_k T(n) = \alpha \cdot n + \beta \iff T(n) = k^{\beta}(^{\alpha n})$$

in other words, if even by eye it seems a linear function using a log scale on the y axis and a linear scale on the x axis linear (this situation being called *semilogarithmic scale*) then two parameters α and β can be estimated (or calculated with methods such as least squares), indirectly estimating the "real" function.

If, instead, the algorithm is polynomial, representing it on a k-logarithmic scale (in which the logarithm is applied to both axes) allows to say that

$$log_k T(n) = \alpha log_k(n) + \beta \iff T(n) = k^{\beta} n^{\alpha}$$

The role of alpha is the degree of the polynomial, but this may be smaller than the degree of the theoretical complexity. The presence of the factor 2^{β} is due to the fact that this analysis is done in *seconds* rather than in elementary operation and depending on the machine used. The interesting point is that what changes is 2^{β} and not n-relative factors.

5.4 A Posteriori Effectiveness Evaluation

5.4.1 Solution Quality Distribution diagram

Effectiveness can be represented in a similar way to the efficiency. The **Solution Quality Distribution** (SQD) diagram plots the distribution function of $\delta_A(I)$ on $\bar{\mathbb{I}}$

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

in words, the probability that the relative difference obtained by the algorithm on the instances of the benchmark is smaller than or equal to a value α for possible real α.

Let's examine the SQD in Figure 5.4. It tells us that it's impossible to obtain a relative difference smaller than 7.5% and, for example, a relative error smaller than or equal to 8% can be obtained in $\approx 12\%$ instances, and accepting a gap of 10% of relative error all instances can be solved.

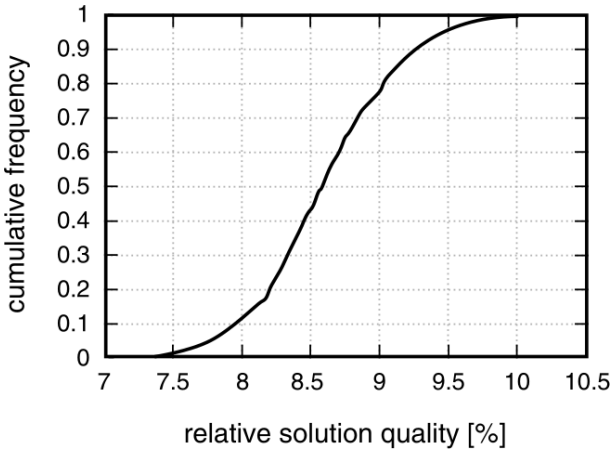


Figure 5.4: An example of SQD diagram.

Building an SQD

This function is represented as continuous, but as before it's obviously non-continuous, as represented in Figure 5.5. SQD is a monotone nondecreasing function, as admitting to worse gaps allows to solve more problems and it's a stepwise function as the graph steps up at each $\delta(I)$. Obviously, it's equal to 0 for $\alpha < 0$ and it's equal to 1 for $\alpha \geq \max_{I \in \bar{\mathbb{I}}} \delta(I)$. In the same way as before, the number of instances in the benchmark sample can be quantitatively estimated by dividing y-axis value by the smallest step.

Given an exact algorithm which solves all instances to optimality, its SQD would be a step function a 0% with comulative frequency 1. Regarding $\overline{\alpha}$-approximated algorithm the plot will show that once the relative solution quality reaches the approximation ratio $\overline{\alpha}$ its cumulative frequency will be, by definition, 1.

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

1. run the algorithm on each instance $I \in \bar{\mathbb{I}}$
2. build the set $\Delta_a(\bar{\mathbb{I}}) = \{\delta_A(I) : I \in \bar{\mathbb{I}}\}$
3. sort $\Delta_A(I)$ by nondecreasing values $\delta_1 \leq \cdots \leq \delta_{|\bar{\mathbb{I}}|}$
4. plot the points $(\delta_j, \frac{j}{|\bar{\mathbb{I}}|})$ for $j = 1, \cdots, |\bar{\mathbb{I}}|$.

5.4.2 Paramteric SQD diagrams

In the case of the RTD it was better to fix the size of the instances in the benchmark sample. In the case of the SQD, this is not so obious and it can be that the quality does not depend so much on the size of the instance and the SQD can be drawn with respect to the whole benchmark of all sizes. Still, there may be some interest in analysing fixed-size benchmark, obtaining different profiles. Drawing different SQD parametrized by the size of the benchmark sample on the same scale as Figure 5.6 represents may show that small instances tend to have small gaps which in turn tends to grow as the size grows. This is not always the case: there are lots of problem in which getting larger instances gives smaller relative costs for the simple reason that large instances have larger optimum value, making δ decrease overall.

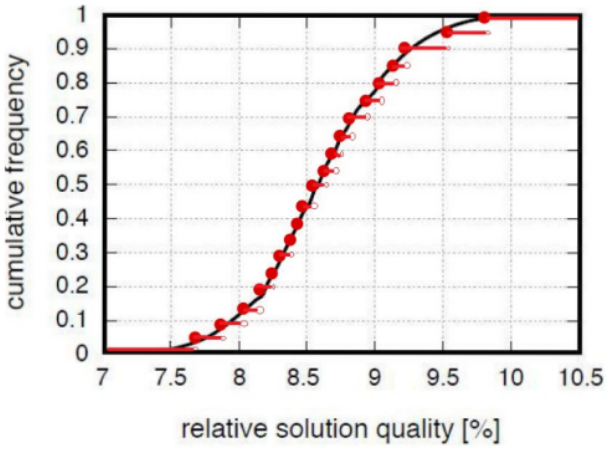


Figure 5.5: Construction of an SQD plot.

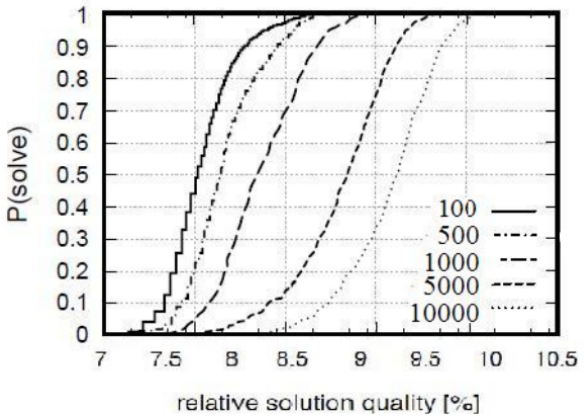


Figure 5.6: An example of paramteric SQD diagram.

5.4.3 Algorithm Comparison with SQD diagrams

Time is a relevant point and it cannot be said that an algorithm is better than another if it takes longer time. But for now, let's suppose that we are to compare two algorithm which consume the same time. It's said that an algorithm A_2 **strictly dominates** an algorithm A_1 when it obtains better results on all instances:

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

this is not frequent and usually happens only on trivial cases, such as A_2 "includes" A_1. In other cases what can be achieved is a **probabilistic dominance** of A_2 on A_1. This is based on the SQD diagrams for the two algorithms, as shown in Figure 5.7, which does not show a strict dominance but A_1 is less "robust" than A_2 as it has results more dispersed than A_2, both better and worse.

So, the probabilistic dominance is based on the distribution function:

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

5.5 Compact Statistical descriptions

All the previous description take a lot of space and sometimes it would be nice to have numbers or compact description of the performance of an algorithm: in particular, some statistical

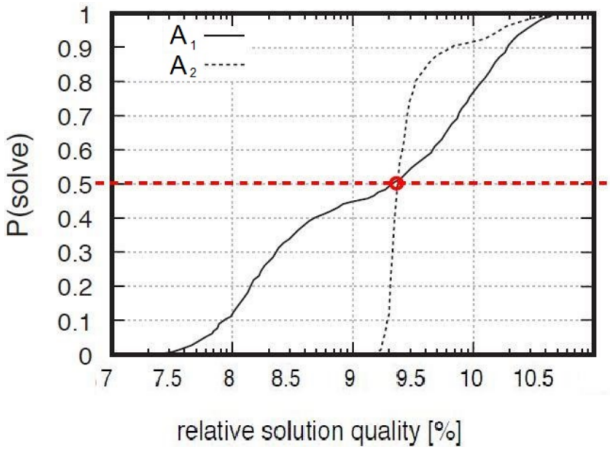


Figure 5.7: An example of SQD-based comparison.

index concerning the relative difference with respect to the optimum can be used. The classical statistical indices of *position* (the sample mean, the arithmetic mean of all measured values)

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

and *dispersion* (the sample variance)

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

can be used in this context. This classical indices tend to be very influenced by **outliers**, that is instances in which the algorithm is very good or very bad. For example: an algorithm which $9/10$ times gives the optimum and $1/10$ times gives ten times the optimum, so it's extremely bad - this is clearly an outlier and probably the algorithm is actually pretty good. The sample mean, will be the sum of the differences $\frac{0+0+\cdots+9}{10} = 0.9$ yielding an average gap of 90%, which isn't a good representation.

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

5.5.1 Boxplots

Both the median and quartiles can be represented using a graphic representation called **boxplot** (or *box and whiskers* diagram).

Following the example of Figure 5.8, suppose to have 20 instances in which the relative difference ranges from $\approx 4\%$ to $\approx 11.5\%$, the best and worst case respectively. The median is the value that is above half of the instances and below half of the instances: if the sample is made of an odd number of cases the median is the value of the "middle" element, but as in the case represented in the Figure the median is something between the tenth and eleventh instance - another option is to use exactly the tenth instance. Typical quantiles used in boxplots are the **first and third quartiles**, that is the values that separate one quarter of the observation from the other three quarters or three quarters from the other one quarter; for example with 20 elements the first quartile is the fifth element and the third quartile is the fifteenth element or the middle point between the fifth and the sixth and the middle point bewteen the fifteenth and sixteenth.

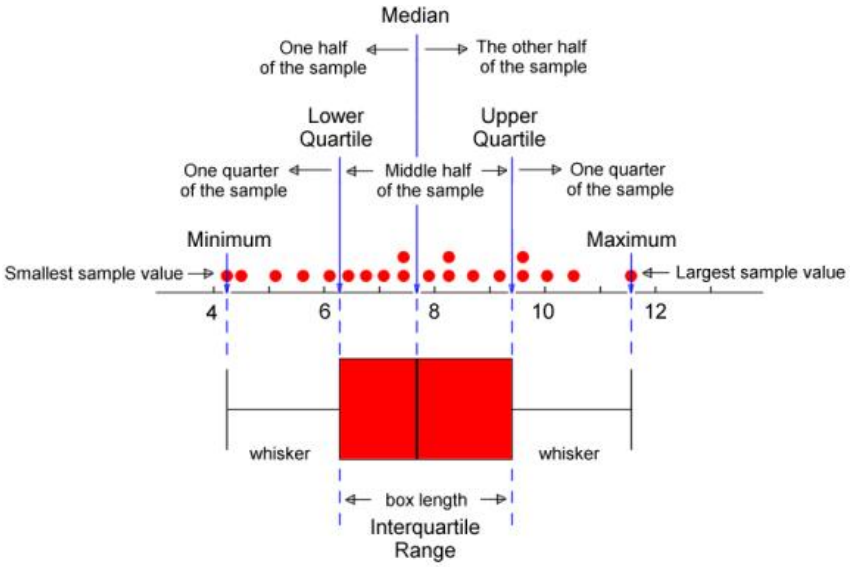


Figure 5.8: An example of boxplot.

Once the three values are computed two more values are needed, which are the extremes; the minimum and the maximum.

This is an interesting diagram as it gives both position and dispersion of the benchmark, saying that exactly half of the instances lie "in the box" and the remaining lie between the ends of the box and the whiskers. This diagram is a simplified description of the SQD and, as two algorithms can be compared using the SQD, they can be compared using boxplots as well. As Figure 5.9 shows, it's clear that the description of algorithms is less precise compared to SQD and less information can be extracted, still some interesting conclusion can be made. First of all, in Figure 5.9 the 8 boxplots represent the relative difference on a fixed benchmark sample and the circles are the outliers. Let's compare A_7 and A_8. The two boxplots are completely separated: this means that all the instances have better values for the algorithm A_7, which is the definition of *strict dominance*.

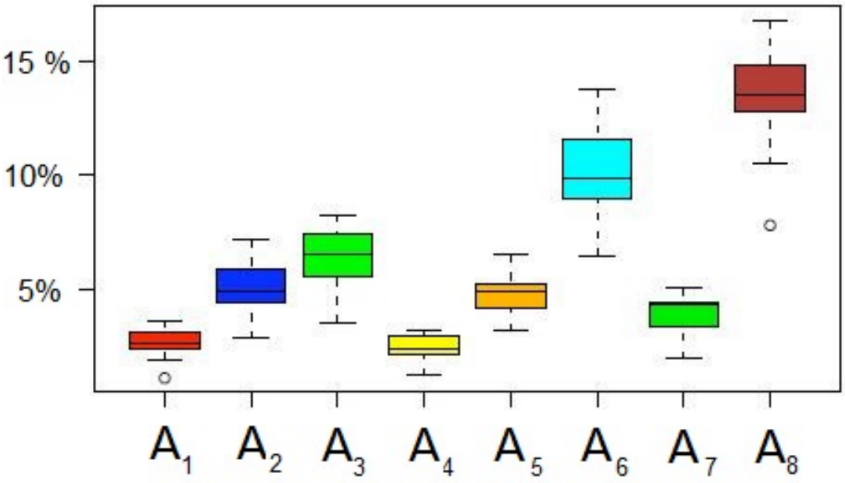


Figure 5.9: Algorithm comparison with boxplots.

On the other hand, *probabilistic dominance* is obtained when each of the four "zones" of a boxplot of some algorithm are above or below the same zones of another: an example of this

is the comparison between A_2 and A_3; the first quartile of A_2 is fully below the first quartile of A_3, the second quartile of A_2 and the mean is below the one of A_3 and so on. This is actually a necessary condition but not sufficient: taking the same example, it may be that one instance starts at the low whisker of A_1 and all the other are exactly at its first quartile while the opposite happens for A_3: surely, in this case, no probabilistic dominance can be asserted.

5.5.2 Relation between quality and computational time

A heuristic algorithm is better than another one not when it *gives better results* but when it does so in a smaller time. If this doesn't happen, it cannot be said that one dominates another. However, there are many algorithm (all metaheuristic algorithms and some heuristic ones) that do not find a single solution as classical algorithms taught in basic courses and what they do is improving previously found solutions. This means that even if the termination of the algorithm is waited for, it could be stopped in advance and a possible solution could be obtained. In turn, this means that the relative difference is not only depending on the instance but also on the effectively consumed time for the computation. The concept of relative difference can be extended to $\delta_A(t,I)$ being the relative difference achieved by A at time t on instance I. If A hasn't found a feasible solution at time t then $\delta_A(t,I) = +\infty$.

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

Randomized Algorithms

If the metaheuristic algorithm has random steps, then the relative difference is also a function of the random seed ω making $\delta_A(t,I,\omega)$. When testing such an algorithm, it should be taken into account that there are two random elements: one is the instance I extracted from $\overline{\mathbb{I}}$ and the other is the random seed itself. During testing these can be combined or distinguished, meaning that in some experiment I is fixed and a batch of seeds $\overline{\Omega}$ is tested or, on the contrary, ω is fixed and a sample of instances $\overline{\mathbb{I}}$ is tested. The results on ω are usually summarized providing both the minimum relative difference $\delta_A^*(t,I)$ and the total time $|\overline{\Omega}|$ or the average relative difference $\overline{\delta}_a(t,I)$ and the single-run time t.

5.5.3 Classification

Now we have all the elements to give a quite complicate **classification** of heuristic, exact and approximated algorithms. This aims to be a classification of all types of algorithms for combinatorial optimization problems or, in general, optimization problems. The names of this classes are extremely hard to remember and not very clear, but they're used at least in some community.

Suppose that you can find in finite time, for all the instances, in which the algorithm certainly provides a solution with relative difference $\delta_A(I,t) = 0$. This is clearly an **exact algorithm** and these are called **complete** algorithms. Formally, the property to fit is

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

A second possibility, stranger but useful, is the case in which the probability of getting a realative difference equal to zero grows converging to one as time passes, for all instances in the benchmark sample.

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

These algorithms do not provide the optimal solution in finite time but will in *enough* time. These are called **probabilistically approximately complete** algorithms. We should observe that in combinatorial optimization we always have the exhaustive algorithm which is exact and takes finite time, so you get perfect certainty in finite time, making a probabilistically approximately complete algorithm not a big news. However, it's an interesting property and of course the idea is that in practice this convergence could occur fast enough to make the probability close to one in a smaller time than the exponential time required by the exhaustive algorithm. Lots of randomized metaheuristics will actually have this property.

The final and third cases are the "*truly*" heuristic algorithms which are those called **essentially incomplete**; both the previous conditions are negated and for some instances $I \in \mathbb{I}$ can find the optimum with probability strictly less than one even as time approaches infinity. Formally

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

Many constructive and local search algorithms have this kind of property.

Generalized classification

This is not the whole classification: we have previously said a lot about approximation guarantees and it can extend the previous three classes. Then, the definizion is not based on the 0 pivot point but on a certain guarantee α.

In this case, α-complete algorithms are the ones in which for each instance there is a time $\bar{t}_I$ such that the relative difference is not necessarily zero but smaller than or equal to α. This is just a differente way to formally describe α-approximated algorithms.

Then, **probabilistically approximately α-complete** algorithms are the ones in which for each instance $I \in \bar{\mathbb{I}}$ an α-approximated solution can be found with probability converging to one as time approaches infinity.

Finally, **essentially α-incomplete** algorithms are the one in which exists some instance $I \in \bar{\mathbb{I}}$ such that an α-approximated soluition can be found with probability strictly less than one as time approaches infinity.

5.6 Complex diagrams

5.6.1 The Probability of success

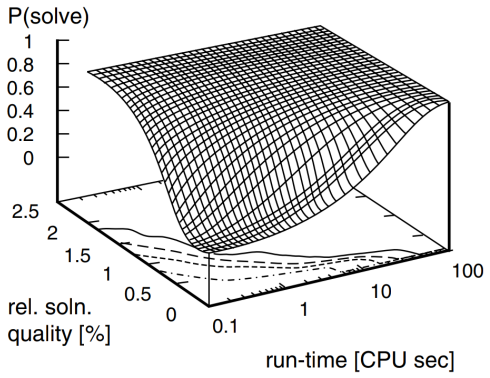


Figure 5.10: An example of success probability diagram

Overall, there are two important parameters that are to be taken into account, one is the time measure described by the threshold t, obviously related to the efficiency of the algorithm, the second is the quality measure described by the threshold α, obviously related to the effectiveness of the algorithm.

Let the success probability $\pi_{A,n}(\alpha,t)$ be the probability that an algorithm A finds in time less or equal to t a solution with a gap less or equal to α on an instance of size n.

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

As represented in Figure 5.10, this diagram is three-dimensional as it depends on two variables, the relative difference and the time. From this diagram, three "auxiliary" classes of diagrams can be extracted by "cutting" it along one of the three axis, that is with respect to time, quality or probability.

Qualified Run Time Distribution diagrams

The first parametric family is called Qualified Run Time Distribution diagrams, in which a quality level is fixed and is considered the line that is drawn on the surface by fixing $\alpha = \overline{\alpha}$.

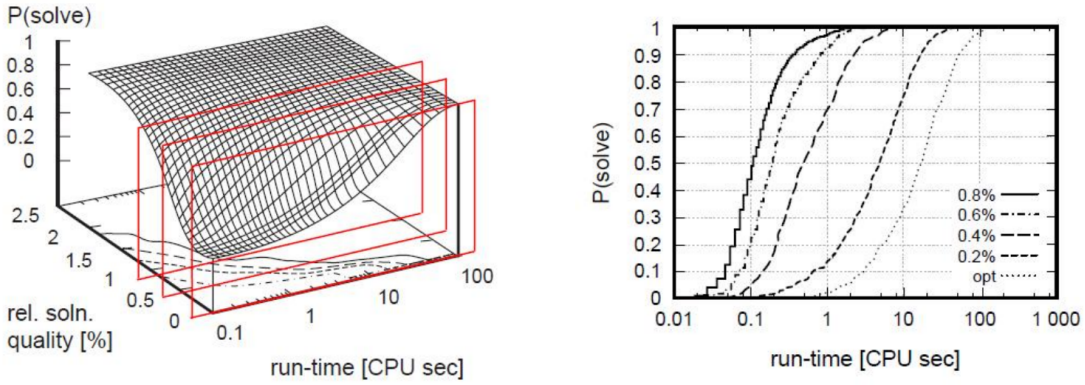


Figure 5.11: An example of QRTD diagram.

The meaning of this diagram is that if you're aiming to obtain a certain quality, then this plot gives the probability of reaching such quality in a given time, as represented in Figure 5.11. If the algorithm is exact, optimality is reached at a certain point in fixed time; if the algorithm is $\overline{\alpha}$-complete then optimality may never be reached but all diagrams of fixed $\alpha \geq \overline{\alpha}$ reach 1 in finite time. Algorithms of class $\overline{\alpha}$-incomplete have diagrams that certainly do not reach 1 for $\alpha \leq \overline{\alpha}$.

Timed Solution Quality Distrbution diagrams

The second parametric family is called Timed Solution Quality Distribution diagrams, in which a time is fixed and it's shown the probability to obtain certain relative difference on instances.

If an algorithm is exact, it's clear that all diagrams with a sufficient t are step functions in $\alpha = 0$. If the algorithm is α-complete, all diagrams with a sufficient t reach 1 in $\alpha = \overline{\alpha}$. If the algorithm is probabilistically approximately α-complete, the diagrams converge to 1 in $\alpha = \overline{\alpha}$. Finally, if the algorithm is $\overline{\alpha}$-incomplete, all diagrams keep $P[] < 1$ in $\alpha = \overline{\alpha}$.

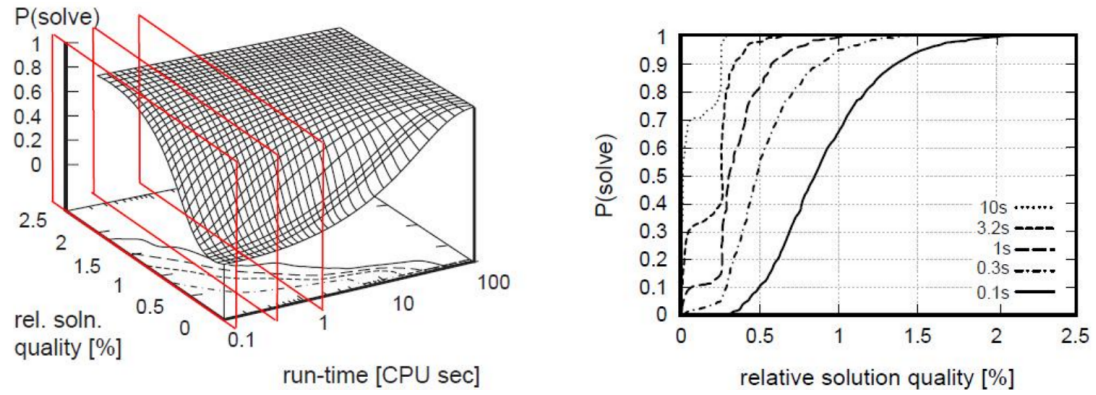


Figure 5.12: An example of TSQD diagram.

Solution Quality Stastistics over Time diagrams

Finally, one can draw the level lines associated to different quantiles, as represented in Figure 5.13.

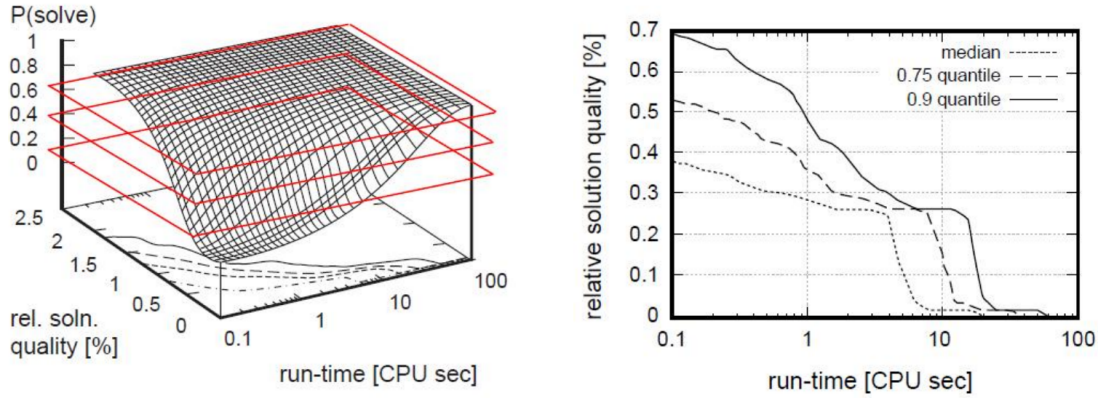


Figure 5.13: An example of SQT diagram.

This is a less clear-to-read diagram with respect to the previous diagrams. Suppose that you're interested in the median, so you want results that hold at least half of the time, which is the almost-dotted line in Figure 5.13. Waiting $\approx$ 20s an optimal solution can be found (half of the times), in less time a much worse quality has to be accepted.

5.7 Wilcoxon's Test

The idea is that we want to compare algorithms. We've made comparations with SQD and boxplots, but they're qualitative and not really quantitative. In order to quantitatively evaluate the significancy of the empirical difference between two algorithms **Wilcoxon's test** can be used. We're going to study the *signed-rank* test which focuses on effectiveness and neglects robustness.

The basic points of the test are:

1. $f_{A_1}(I) - f_{A_2}(I)$ is a random variable defined on the sample space $\mathbb{I}$.
2. formulate a **null hypotesis** H_0 according to which the theoretical median of $f_{A_1}(I) - f_{A_2}(I) = 0$.

3. extract a sample of instances $\bar{\mathbb{I}}$ and run the two algorithms on it, obtaining a sample of pair values (f_{A_1}, f_{A_2}).

4. compute the probability p of obtaining the observed result or a more "extreme" one assuming that H_0 is true

5. set a significance level $\overline{p}$, which is the maximum accetable probability

   - to reject H_0 assuming that is is true
   - that is, to consider two identical medians as different
   - that is, to consider two equivalent algorithms as differently effective, referring to the median of the gap

   and reject H_0 when $p < \overline{p}$

Statistical tests are based on formulating an hypotesis called **null hypotesis** H_0 in order to understand the property of observing a certain empirical behaviour given that such hypotesis is true. In other words, our null hypotesis is that half the times f_{A_1} is better than f_{A_2} and half the time it's the opposite. This is a good description of two equivalent algorithms.

Instead of considering all the instances a sample is extracted and a sequence of pairs of values is found (3). Then, try and describe the probability of obtaining this particular set of pairs or also any other set of pair that is more unbalanced than the observed still assuming that anyway H_0 is true, meaning that it's to be understood if what's observed is just "bad luck" or it's a significant result (4). Once the probability is computed it is compared to the so called **significance level** $\overline{p}$ which is the maximum probability you accept to keep H_0 as possible (5). Typical values for $\overline{p}$ are 5% and 1%. If the calculated probability is smaller, then H_0 is proven false and rejected and the theoretical median between algorithms is not null.

5.7.1 Assumptions on Wilcoxon's Test

This is a very hard methodological problem. It is a **nonparametric test**, which means that you're not assuming *anything* on the distribution of the tested values and it is useful to evaluate of the performance of heuristic algorithms as the distribution of the results $f_A(I)$ is unknown. It is based on the assumptions that:

1. all data are measured at least on an ordinal scale, which means that the specific values do not matter, only their relative size

2. the two datasets are matched and derive from the same population; in other words we apply A_1 and A_2 to the same instances extracted from $\mathbb{I}$

3. each pair of values is extracted indipendently from the others, which means that the instances are generated indipendently from one another.

The (3) assumption is not always verified, so it's necessary to pay attention to it. Think about some hard problem like the vehicle routing problem. Generating different families of weights (for edges and vertices) on the same graph makes the instances not really *independent* from one another.
In order to execute the test, the following steps must be followed:

1. compute the absolute differences $|f_{A_1}(I_i) - f_{A_2}(I_i)| \ \ \forall I_i \in \bar{\mathbb{I}}$

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

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

4. the difference $W^+ - W^-$ allows to compute the value of p: each of the $|\bar{\mathbb{I}}|$ differences can be positive or negative, so there are $2^{|\bar{\mathbb{I}}|}$ outcomes. Then, p is the fraction with $|W^+ - W^-|$ equal or larger than the observed value

5. if $p < \overline{p}$ the difference is significant and if $W^+ < W^-$ A_1 is better than A_2 and the contrary if $W^+ > W^-$

It's important that step (2) assign the same rank to same absolute difference, and in particular the half of the number of equal absolute difference: numerically, given 20 values of 1 their rank should be 10.

5.7.2 Calculating p

Once calculated W^+ and W^- and $W^+ - W^-$, it's necessary to calculate the value of p, which is the probability of the difference to be significant.

Let's make an example showing how to calculate p. Suppose to have 3 instances I_1, I_2 and I_3 which yields differences with rank 1,2 and 3. The most unbalanced situation is the one in which A_1 or A_2 is always better than the other, so there are either three positive or three negative differences; in other words, it could happen that 1, 2 and 3 are all ranks of positive differences or all negatives. Each value can be either "positive" or "negative" (meaning that it is a rank of a positive or negative difference, respectively), so there are $2^3 = 8$ possibilities. Each of these outcomes is generally equiprobable, so the eight cases can be enumerated, as the upper-left subfigure in Figure 5.14 shows: in one case all are positive and it yields $W^+ - W^- = (1+2+3) - 0 = 6$; in one case all are negative and it yields $W^+ - W^- = 0 - (1+2+3) = 6$ and so on. In two cases the difference is zero: $W^+ - W^- = 0 = (1+2) - 3 = 3 - (1+2)$.

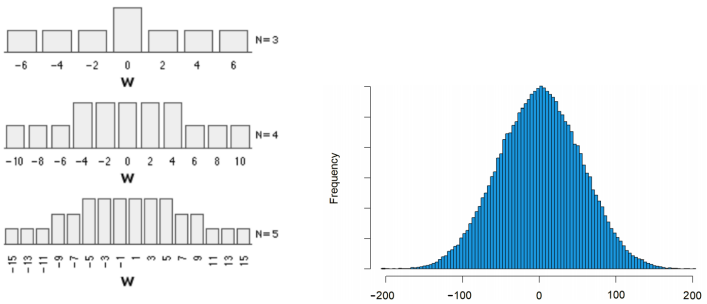


Figure 5.14: Wilcoxon's test distributions for various sample sizes.

We "expect" by the null hypotesis to be in one of the two cases in which the difference is $W^+ - W^- = 0$, but if we aren't, what is its probability? Assuming that the two populations are the same so that the median is on 0, the probability of being in a non-equal difference situation **or worse** is the sum of the probability for each enumeration in case: for example, if the yielded difference is 6 its probability is $\frac{1}{8}$. If the obtained difference is 4, the probability of obtaining so **or worse** (as specified by step (3)) is $\frac{2}{8}$ and so on. So, it can be computed by enumeration or, if

there are many instances, the *law of big number* says it's okay to approximate it with a normal distribution.
Wilcoxon's test can suggests that one of the two algorithms is significantly better than the other or that the two algorithms are statistically equivalent. In both cases, it's important to remember that it's a stochastic response and it's important to "keep an eye" on p. If the sample includes instances of different kinds, the two algorithms could be overall equivalent but nonequivalent on the single classes of instances. So, what about testing $\delta_A(I)$ instead of $f_A(I)$? It's an open question. The results can be different and using $\delta_A(I)$ means giving a smaller weight to the instances that have a larger optimum. Probably, if the outcome of the test is different in the two cases, that test is probably not really realiable.

Part III

Solution Based Heuristics: Constructive Heuristics

CHAPTER 6

Constructive Heuristics

After listing sets of problems and how to evaluate an algorithm a priori and a posteriori, we consider the first of the three classes of heuristics previously listed. The first class is the class of the so called **constructive heuristics**, but this chapter will also include **destructive heuristics**, which will not be analysed in depth.

6.1 Introduction to Constructive Heuristics

In order to understand the constructive heuristics we have to remind that **combinatorial optimization** problems admit solutions that are always subsets of a given ground set, that is a finite suitable set. The fact that we have subsets suggests a possible way to build a solution, and in particular constructive heuristics consist in starting from the empty subset $\emptyset$ and then adding one element at the time until it is *guessed* that it doesn't make sense to introduce new elements and so the algorithm stops. Let's investigate this process in more detail: a constructive heuristic updates a subset $x^{(t)}$ step by step

1. start from an empty subset: $x^0 = \emptyset$, which is obviously a subset of any optimal solution. We already know that by kernelization or using reduction procedures it can be proven that some elements of the ground set are necessarily included in at least one optimal solution. In such case, it may be a good idea to start including in the original subset these "forced" elements. Perhaps, there may be some heuristic reason to think that some element should be included.

2. stop when a termination condition holds: we test a termination condition and its rationale is that adding elements in the current subset would make no sense as it would not give any optimal solution. We'll see some condition of this kind, but in general you stop when you think that the algorithm has nothing useful to find.

3. loop:

   (a) determine among all the elements $i^{(t)} \in B$ that don't belong to the current subset $x^{(t)}$ those that could be added to it giving an "interesting" possible solution, while trying to keep $x^{(t)}$ whithin a feasible and optimal solution. The definition of *acceptable* depends on the algorithm. This is one of the two parts that are not general and characterize an algorithm.

   (b) add $i^{(t)}$ to the current subset: $x^{(t+1)} := x^{(t)} \cup \{i^{(t)}\}$. The property of having $x^{(t)}$ as part of a feasible (actually optimal) solution is guaranteed by starting with $x^{(0)}$ as previously defined. Generally, this property will be lost by the addition of elements,

and the algorithm as to try and keep this property - if not optimality at least feasibility.

4. go back to point 2.

This is a very simple scheme and it also admits a nice modeling representation, and we're going to see what these modeling tools are and what kind of theory can be given on this very general structure.

6.1.1 The construction graph

The main modeling tool is the **construction graph** and it will be used a lot. The graph is directed, made of nodes and arcs: the nodes of this graph make out a set F_A (F stands for *find*) which is also known as **search space** which depends on the algorithm A; in other words F_A depends on the problem to be solved, but from the same problem there may be different search spaces corresponding to different algorithms, that is different construction heuristics.

The search space is defined as the collection of all subsets $x \subseteq B$ acceptable for the algorithm A ($F_A \subseteq 2^B$): each node is a subset and some will be solutions and some other won't be. We're interested in the subsets that are acceptable and useful.

The arcs connect pair of nodes and the arc set is the collection of all pairs $(x, x \cup \{i\})$ such that $x \in F_A, i \in B \setminus x$ and $x \cup \{i\} \in F_A$. So, the arcs connect a subset to a slightly larger subset which represent an extension of an acceptable subset with a new element added to it, being an acceptable subset still.

The first property of this construction graph is that it's an acyclic graph. Figure 6.1 represent

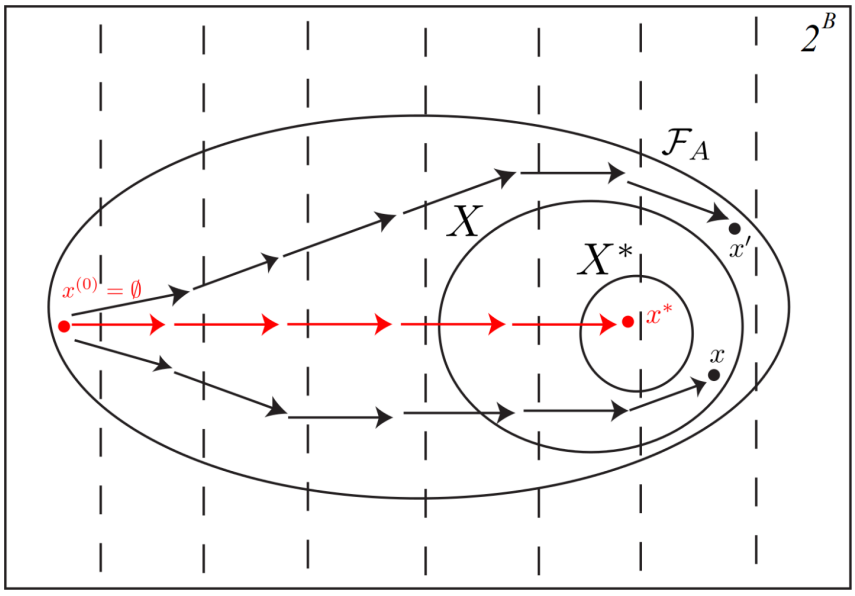
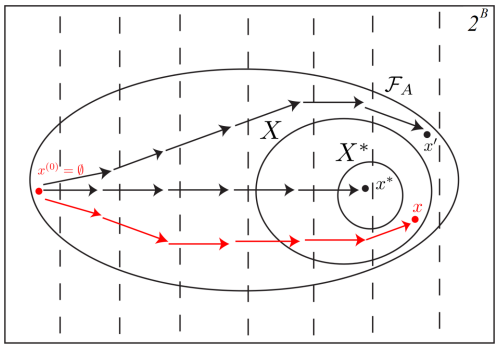


Figure 6.1: An example of construction graph.

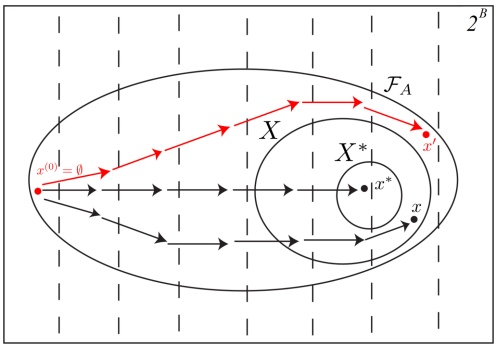
a construction graph. The rectangle represents 2^B, that is the power set of B. This collection of subsets is partitioned into classed depending on their cardinality, so in the first class (the leftmost) there's just one possible class, that is the empty set $\emptyset$ which contains all the subsets of 0 elements. In the second class there are all the *singletons*, then the pairs and so on. Overall, there are $n+1$ classes going from cardinality 0 to cardinality n. Inside the power set there is the search space of the algorithm, which contains some subsets (in general not all of them)

which are considered useful for the algorithm: in particular, it contains the empty set because the algorithm starts from that and then it contains other subset that make sense to keep into account. Typically, subsets outside the search space are those that do not make sense to visit as they're not-feasible solutions or because they're not particularly "interesting" and any other subset that can be reached from it is not relevant as well. As before, the set of feasible solution is denoted as X and the subset of optimal solutions is denoted X^*. In this search space there's a set of arcs connecting pair of nodes and the second of the pair is equal to the first set of the pair plus a certain element. As can be seen from the figure, starting from the empty set one element is added: some may be acceptable, hence they're potentially chosen, and some are not acceptable and are out of the search space, so there's no arc going "out" of the search space.

So, every execution of an algorithm is a maximal path in the construction graph starting from the empty set and going to some solution, which ideally should be optimal but at least feasible and in practice it's a solution that cannot be extended in any way. In the optimal case, represented in Figure 6.1, the path ends in an optimal solution, which is the the case for exact constructive algorithm such as Prim's, Kruskal's and Dijkstra's algorithms.



(a) A strictly feasibile solution in the construction graph.



(b) An unfeasible solution in the construction graph.

Figure 6.2: Other examples of construction graph, relating to the use of different algorithms.

In other cases, such as the ones represented in Figure 6.2, the path may end in a nonoptimal but feasible solution $x \in X$ (Figure (a)), which is the case for heuristic algorithms such as those used for the Knapsack Problem and the Maximum Diversity Problem, or even in unfeasible solution $x' \notin X$ (Figure (b)) - an example of this is the TSP on a noncomplete graphs.

Examples of construction graphs

An example of construction graph is the one for the TSP in the Nearest-Neighbour for complete graphs, where the idea is to start with no arcs, so $x^{(0)} = \emptyset$ and has $n-1$ outgoing arcs. Each one of the first reached nodes ha $n-2$ outgoing arcs and so on. This construction graph will be a tree with $n-1$ arcs and n nodes; the number of paths from the empty set to the leaves will be exactly $(n-1)!$, that is the number of possible execution of the algorithm, obtaining then $(n-1)!$ possible solutions.

In the case of the MDP the situation is similar but a bit different: starting from the empty subset it's possible from there to add one point in n possible ways, then add another in $n-1$ possible ways but in this case there won't be a tree as adding at the first step a point and at the second step another is exactly the same as adding the second point first and the first at the second step, obtaining the same subset with two different paths. The graph will be acyclic but will have

path with the same origin and same destination, having k levels: the paths can reconverge and they terminate in $\binom{n}{k}$ different solutions.

Properties of search space in construction graphs

Let's list some properties of the search space in terms of construiction graph. The search space F_A should include the empty subset $\emptyset \in F_A$ as any algorithm starts from it. In general, the search space F_A also includes the feasible solutions $X \subseteq F_A$ such as was represented in Figure 6.1. Sometimes some solution can be removed from it as they may be obviously nonoptimal: a case is the SCP where starting with the empty set of column any column is added an at a certain point the feasible solution is reached but when it gets to one of the solutions in X it doesn't make sense to add other columns because the new subsets will be feasible but also more expensive.

Another basic property is that F_A contains subsets that must be reachable from the start point, otherwise there would be no point in adding a solution which is unreachable from the start. A natural candidate set is the collection of **partial solutions**, that is solutions that are in general nonfeasible but are subsets of feasible solutions. This is a nice definition of the search space because following partial solution will give at least one path to a solution in the feasible set X.

This distinction is not necessarily easy to give and doesn't always admit an inclusion test which is quick and polynomial. In practice, what's happening is that giving a subset it's to be known if it's a partial solution or not. This is a decision problem and its complexity depends on the problem itself. If there's no easy way to decide, a solution is to *relax* the search space, making the search space larger than it should.

Examples of Search Spaces The search space F_A may include all feasible solutions, so that

$$F_A \equiv X$$

as in the KP; it may include all partial and all feasible solutions, so that

$$F_A \equiv \cup_{x \in X} 2^x$$

for example in the MDP, in which not only the subsets of k points are considered but also those that are smaller or in Kruskal's algorithm; the search space may include only some special promising partial solutions

$$F_A \subset \cup_{x \in X} 2^x$$

an example of this is Prim's algorithm for the MSP, which doesn't consider all partial solution but considers a tree that's increasing remaning a connected tree until it becomes spanning. The feasible solution may be defined as some special subsets not sufficient to guarantee feasibility:

$$U_{x \in X} 2^x \subset F_A$$

an example of this is the TSP, where F_A may be defined as all subsets of arcs not including branching and subtours, as it cannot be defined as "all subsets that are part of feasible solutions" since it's a NP-complete decision problem.

6.1.2 Termination condition

Previously, it was said that the algorithm terminates when the "current" subset cannot be extended without leaving the search space, in other words there is no arc going out of the node in

the construction graph. Formally,

$$x^{(t)} \cup \{i\} \notin F_A \quad \forall i \in B \setminus x^{(t)}$$

and, equivalently, it has no outgoing arc

$$\Delta_A^+(x^{(t)}) = \{i \in B \setminus x^{(t)} : x^{(t)} \cup \{i\} \in F_A\} = \emptyset$$

Among these possible extension, the algorithm will choose one until $\Delta_A^+(x^{(t)}) = \emptyset$ is empty, then the algorithm may eventually do some backtracking in order to check if one of the visited node one is a feasible subset, giving the optimal - the algorithm returns the best solution visited during the execution, which usually is the last one.

Different behaviour are possible: sometimes all visited subsets are feasible, but often the last subset is the only feasible solution. Further, $x^{(t)}$ could move in and out of X or even X^*.

6.2 Structure of Constructive Heuristic algorithms

A constructive heuristic (for minimization problems) can be described as: In words, Algorithm

Algorithm 1 Constructive Heuristic Pseudocode

```
1:  procedure GREEDY(I)
2:      x := ∅
3:      x* := ∅
4:      if x ∈ X then
5:          f* := f(x)
6:      else
7:          f* := +∞
8:      end if
9:      while Δ_A^+(x) ≠ ∅ do
10:         i := arg min_{i∈Δ_A^+(x)} φ_A(i,x)
11:         x := x ∪ {i}
12:         if x ∈ X and f(x) < f* then
13:             x* := x
14:             f* := f(x)
15:         end if
16:     end while
17:     return (x*, f*)
18: end procedure
```

1 starts from an empty solution maintaining two variables x^* and f^* which represent the best solution found yet; if $x \notin X$ then $f(x) = +\infty$. The algorithm loops until the termination condition is met, chosing and adding elements to x - the choice of the item to add i is based on a function that depends on the current subset $\varphi(i,x)$ defined as

$$\varphi_A : B \times F_A \to \mathbb{R}$$

which gives a real value which estimates how good the insertion of i in x is and for the sake of simplicity it was chosen that it's to be minimized (so it's a minimization problem). This is a general scheme and it's customized to the current problem by fixing $\Delta_A^+(x)$ and $\varphi_A(i,x)$. In terms of construction graph, the first is a mathematical representation of the graph topology and the last is a value associated with each arc in the graph.

6.2.1 Effectiveness and Efficiency

If you're lucky, the constructive algorithm will find the optimum because at each step the subset x will remain a partial solution of an optimal solution. This requires some nice property from both the problem and the algorithm: it is a property that is necessarily true at first but then may be lost. The effectiveness of a heuristic algorithm critically depends on the fact that this property - being included in an optimal solution - remains satisfied or not. Then, concerning the efficiency of a heuristic constructive algorithm it can be observed that the cost is given by the fact that a number of steps that is at most n is followed and at each iteration the extension set $\Delta_A^+(x)$ has to be built and the weight of each arc to be evaluated with $\varphi_A(i,x)$ for each $i \in \Delta_A^+(x)$, then choose the minimum one and possibly update x and auxiliary data structures. In general, the complexity is a polynomial of low order dominated by the first two components.

$$T_A(n) \in O(n(T_{\Delta_A^+}(n) + T_{\varphi_A}(n)))$$

General Features of Constructive algorithms

Constructive algorithms are really intuitive, simple to analyse and implement and tend to be efficient provided that the two components in the complexity arent' very "bad". However, they suffer of strongly variable effectiveness: on some problem they guarantee an optimal solution, on other they may provide only an approximation guarantee, on most problems they provide solutions of extremely variable quality and sometimes they cannot even guarantee a feasible solution. It's very important to study the problem before, because it can tell if it's possible or not to find the optimum and maybe what kind of features the algorithm may have to reach a certain performance.

When to use Constructive algorithms

Constructive algorithm are usually used certainly when they provide some optimality guarantee; they're used when the execution time must be very short or when the problem has a huge size or requires heavy computations. They may even be used as components of other algorithms.

6.2.2 A Remarkable case

A particularly remarkable case is that in which the objective function is extended from X to F_A and the selection criterion is the objective function, that is

$$\varphi_A(i,x) = f(x \cup \{i\})$$

In the second chapter, an argument was made regarding the fact that the objective function sometimes is defined on a larger set than "just" X. This is one of the case in which the objective function has to be defined elsewhere: a possible place in which to define the objective function is exactly F_A, so to use it as a *guide*, which will be used as a metric to decide the addition of an i to the solution.

The Fractional Knapsack problem

The Fractional Knapsack Problem is a problem in which the solution is obtained by selecting from a set of object of identival value a maximum value subset which could be contained in a knapsack of limited capacity.

In the FKP the capacity simply imposes a cardinality constraint: the feasible solutions are those with $|x| \leq \lfloor \frac{V}{v} \rfloor$. This problem is similar to the KP and the other functions and sets are defined as for the KP.

Algorithm 2 Constructive Heuristic Pseudocode for the FKP

```
1:  procedure GREEDYFKP(I)
2:      x := ∅
3:      x* := ∅
4:      if x ∈ X then
5:          f* := f(x)
6:      else
7:          f* := +∞
8:      end if
9:      while |x| < ⌊V/v⌋ do
10:         i := arg max_{i∈B\x} φ_i
11:         x := x ∪ {i}
12:         if x ∈ X and f(x) < f* then
13:             x* := x
14:             f* := f(x)
15:         end if
16:     end while
17:     return (x*, f*)
18: end procedure
```

The subset x can be extended as long as $|x| < \lfloor V/v \rfloor$. Any element of $B \setminus x$ extends x in a feasible way and the objective function is additive, therefore

$$f(x \cup \{i\}) = f(x) + \phi_i \rightarrow \arg\max_{i \in B \setminus x} f(x \cup \{i\}) = \arg\max_{i \in B \setminus x} \phi_i$$

The last subset visited is the best solution found. The two `if`s are not useful, as in general the update will be done without any check.

For example, let it be

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1

The algorithm performs the following steps:

1. $x := \emptyset$
2. $x := \{a\}$
3. $x := \{a,d\}$
4. $x := \{a,c,d\}$
5. $x := \{a,c,d,e\}$
6. terminate

and always finds the optimal solution. Why?

The Knapsack Problem

Considering the basic Knapsack problem the algorithm is rather similar, just the set of possible extensions is not simply given by all the elements that aren't in the solution yet but is given by those elements whose volume does not exceed the remaining capacity of the knapsack (Algorithm 3).

Algorithm 3 Constructive Heuristic Pseudocode for the KP

```
1: procedure GREEDYKP(I)
2:     x := ∅
3:     x* := ∅
4:     f* := 0
5:     while ∃i ∈ B \ x; v_i ≤ V − Σ_{j∈x} v_j do
6:         i := arg max_{i∈B\x; v_i ≤ V − Σ_{j∈x} v_j} φ_i
7:         x := x ∪ {i}
8:     end while
9:     return (x, f(x))
10: end procedure
```

In a numeric instance such as the one in following table,

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

the algorithm will follow these steps:

1. $x := \emptyset$
2. $x := \{a\}$
3. $x := \{a,d\}$
4. terminate

which does not yield the optimal solution $x^* = \{a,c,e\}$. Why? The algorithm was exactly the same and the problem was just a bit different.

The Travelling Salesman Problem

Let's introduce a third example with the TSP. The constructive algorithm adds each time the cheapest arc among those that do not form subtours and keep a degree ≤ 1 in all nodes.

As of now, let's consider a not necessarily complete graph. This can be solved considering that the objective function is again additive which means that the starting cost can be ignored.

It's clear that only the last subset can be feasible, if any.

The only thing that "violates" completeness of the graph in Figure 6.3 is the fact that the nodes $1-3$ are not mutually connected, in fact there's a dashed arc. Consider first the instance in which the dashed arc doesn't exist; the algorithm performs the following steps:

1. $x := \emptyset$
2. since $\Delta_A^+(x) \neq \emptyset$, select $i := (3,5)$ and update x

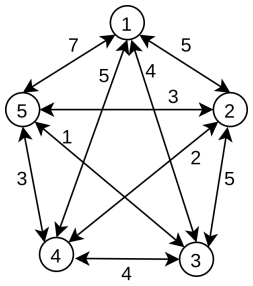


Figure 6.3: Numerical TSP example.

Algorithm 4 Constructive Heuristic Pseudocode for the TSP

```
1:  procedure GREEDYTSP(I)
2:      x := ∅
3:      x* := ∅
4:      f* := +∞
5:      while Δ_A^+(x) ≠ ∅ do
6:          i := arg min_{i∈Δ_A^+(x)} c_i
7:          x := x ∪ {i}
8:          if x ∈ X then
9:              x* := x
10:             f* := f(x)
11:         end if
12:     end while
13:     return (x*, f*)
14: end procedure
```

3. since $\Delta_A^+(x) \neq \emptyset$, select $i := (2,4)$ and update x, noticing that $(5,3) \notin \Delta_A^+(x)$ as it would create a subtour
4. since $\Delta_A^+(x) \neq \emptyset$, select $i := (5,2)$ and update x ($(4,2) \notin \Delta_A^+(x)$)
5. since $\Delta_A^+(x) \neq \emptyset$, select $i := (4,1)$ and update x, notice that $(2,5),(4,5),(5,4),(3,4)$ and $(4,3) \notin \Delta_A^+(x)$
6. since $\Delta_A^+(x) = \emptyset$, terminate

The algorithm does not find a feasible solution, but feasible solution exist: for example a simple tour following the external arcs. Adding the arc $(1,3)$ with cost 100 makes the algorithm find a feasible but nonoptimal solution. Why?

The Maximum Diversity Problem

The MDP is characterized by a cardinality condition as well: the solution is to select from a set of points a subset of k points with the maximum sum of the pairwise distances.

When choosing the i to update the solution, instead of evaluating the objective function $f(\cdot)$ which is quadratic, it can be split as the objective function value for the original subset x plus the distances from each point in x to the newly added i and from i to each point in x:

$$f(x \cup \{i\}) = f(x) + 2\sum_{j \in x} d_{ij} + d_{ii} \rightarrow \arg\max_{i \in B \setminus x} f(x \cup \{i\}) = \arg\max_{i \in B \setminus x} \sum_{j \in x} d_{ij}$$

and the distance between i and i, which is 0. So, this means that x is not to be considered again when choosing a new i, and just the sum has to be maximized. Any element of $B \setminus x$ extends x in a feasible way.

Algorithm 5 Constructive Heuristic Pseudocode for the MDP

```
1: procedure GREEDYMDP(I)
2:     x := ∅
3:     x* := ∅
4:     while |x| < k do
5:         i := arg max_{i∈B\x} Σ_{j∈x} d_ij
6:         x := x ∪ {i}
7:     end while
8:     return (x, f(x))
9: end procedure
```

The last subset visited is the best and only feasible solution found. Still, this algorithm doesn't work.

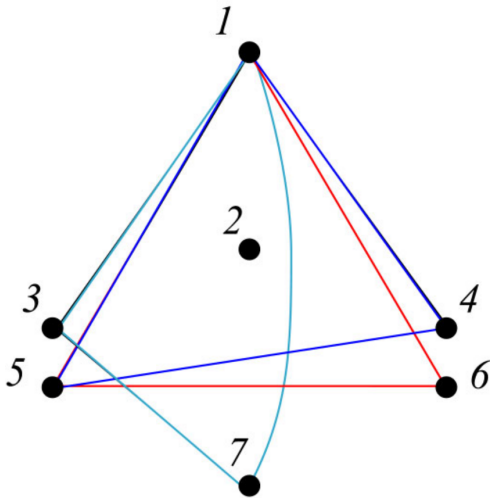


Figure 6.4: An example of Maximum Diversity Problem

Let's examine a solution for the set of point represented in Figure 6.4: at first, the algorithm takes the point with the maximum *diversity* from itself, which is always 0, therefore any point is valid. So any point is "valid" from this point of view, even if it can preclude the finding of a solution just from start. Accepting this, the second point is just the farthest from the first, such as the couple $(1,7)$ in the Figure. From these two point a third has to be chosen with the maximum distance from the two point themselves: 3 and 4 can be both chosen, let's take 3. The solution is nonoptimal, which would actually be $(1,5,6)$.

Actually, even if the algorithm starts from another node it will not get the optimal solution. Why?

6.3 Exact Constructive Algorithms

What features allow a constructive algorithm to find the optimum? A search space identical to the feasible region, $F_A = X$? No, because both the fractional and integer KP do and a constructive algorithm for the latter does not yield an optimal solution. A cardinality-constrained

problem? This would explain the failing on the KP but not on the MDP and TSP. An additive function? Neither, as it does not explain the failure on the TSP.

There is **no general characterization** of the problems solved exactly by constructive algorithms, but there are characterization for wide classes of problems.

6.3.1 The Additive Case

Assume that the objective function is additive, so

$$\exists \phi : B \to \mathbb{N} : f(x) = \sum_{i \in X} \phi_i$$

in words there is a function such that the objective function is given by the sum of this auxiliary function on the elements of the solution. Assume, then, that these solutions that are $x \subseteq B, x \in X, x \in 2^B$ are maximal subsets (so they are **bases** appartaining to the set of bases $\mathcal{B}_F$) of the search space F

$$X = B_F = \{Y \in F : \nexists Y' \in F : Y \subset Y'\}$$

meaning that Y is so that there's no other subset strictly larger than it; the set of feasible solution X is exactly the set of all these Y. This is not a strange assumption: we've seen that most of the combinatorial optimization problems that we've seen have an additive objective function (except MDP, SCP, BP, PMSP) and, concerning the bases, for example the Knapsack Problem are actually all the subsets that respect the capacity of knapsack but considering the maximal one you're actually considering the most promising ones (feasible and maximal set), so it makes sense to say that the interest is only in the bases. For other problems such as TSP, solutions have to be hamiltonian circuits made out of n arcs and it's impossible to enlarge it; the same holds for some other (not all of them) problems we've seen.

The constructive algorithm always find the optimal solution for any possible additive objective function if and only if the pair (B, F) the ground set and the search space is a **matroid embedding**.

6.3.2 Special Matroid Embedding cases

Since the definition of matroid embedding is rather complex, let's focus on some important simpler cases.

Greedoid

The first definition needed is the **greedoid**. A greedoid (B, F) with $F \subseteq 2^B$ is a pair such that

- (**trivial axiom**) $\emptyset \in F$, necessary as a greedy algorithm start from the empty set.
- (**accessibility axiom**) if $x \in F$ and $x \neq \emptyset$ then $\exists i \in x : x \setminus \{i\} \in F$, in words any acceptable subset can be built adding elements in suitable order (any point in the F_A drawn as a construcion graph is reachable from the starting point $x^{(0)} = \emptyset$). It's not guaranteed that *any* i can be taken away, but at least one exists.
- (**exchange axiom**) if $x, y \in F$ with $|x| = |y| + 1$, then $\exists i \in x \setminus y$ such that $y \cup \{i\} \in F$, in words any acceptable subset can be extended with a suitable element of any other acceptable subset of larger cardinality.

The exchange axiom implies that all bases have the same cardinality. Let's consider the fractional KP. Certainly, the trivial axiom holds. The accessibility axiom holds as well as any set of objects that has volume not larger than the capacity can be reduced cancelling an element (actually any of them) remaining inside the search space. The exchange axiom holds as given a feasible subset of 3 elements and exists a feasible subset with 4 elements then one of the 4 can be added to the first subset which will remain in the search space. The axioms hold as well in the MST problem considering the search space as each possible spanning forest of the graph, that is each set of arcs that are not cyclic. In the case of Prim's algorithm the accessibility axiom's interesting as at least one of the edges can be removed still remaining in the search space, but not any: it has to be a "leaf edge".

The axioms do not hold in the general KP, TSP and many other. The axioms actually hold for the MDP, but the objective function is not additive!

Being a greedoid is a necessary but not sufficient condition, so in general constructive algorithms do not work on greedoids.

Matroid

By strenghtening the accessibility function one can obtain **matroids**. A matroid is a **set system** (B,F) with $F \subseteq 2^B$ such that

- (**trivial axiom**) $\emptyset \in F$
- (**hereditarity axiom**) if $x \in F$ and $y \subset x$ then $y \in F$, in other words any acceptable subset can be built adding its element in any order (not just a very well chosen element).
- (**exchange axiom**) if $x,y \in F$ with $|x| = |y| + 1$ then $\exists i \in x \setminus y$ such that $y \cup \{i\} \in F$, in words any acceptable subset can be extended with a suitable element of any other subset of larger cardinality.

The hereditarity axiom is a stronger version of accessibility: it holds in Kruskal's search space but it does not hold for Prim's. For the TSP the two first axioms hold, but the third one doesn't.

Graphic Matroid and MST Suppose that the search space is

$$F = \{x \subseteq B : x \text{ forms no cycles}\}$$

The trivial axiom holds, as with the empty solution no cycles are formed. The hereditarity

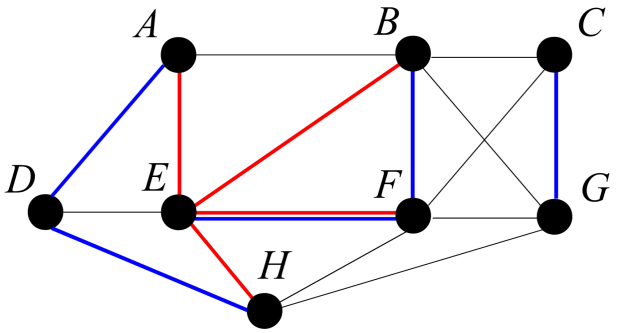


Figure 6.5: MSTP is a matroid.

axiom holds: if x is acyclic all of its subsets are acyclic. The exchange axiom holds: if x and y are acyclic and $|x| = |y| + 1$ then one can always add a suitable edge of x to y without forming any cycle, as shown in Figure 6.5 where x is the set of blue arcs and y is the set of red arcs. The edges (A,D), (C,G) and (D,H) can be added to y.

Uniform Matroid and fractionary KP Suppose that the search space is

$$F = \{x \subseteq B : |x| \leq \lfloor V/v \rfloor\rfloor\}$$

The trivial axiom holds: the empty set respects the cardinality constraint. The hereditarity axiom holds: if x respects the cardinality constraint, all of its subsets also respect it. The exchange axiom holds: if x and y respect the cardinality constraint and $|x| = |y| + 1$ one can always add a suitable element of x to y without violating the cardinality, in fact any element of x. For the general KP the first two axioms hold, but the third one doesn't: for example, let $V = 6$ and $v = [3,3,2,2,1]$; the subsets $x = \{3,4,5\}$ and $y = \{1,2\}$ are in F, but no element of x can be added to y.

Greedoids with the strong exchange axiom

The last class of simple matroid embedding is the class of **greedoids with strong exchange axiom**. Prim's algorithm is optimal for a special greedoid. Let B be the edge set of a graph and F the collection of the trees including a given vertex v_1. Thanks to a strenghtening of the exchange axiom, the **strong exchange axiom**:

$$\begin{aligned} & x \in F, y \in \mathcal{B}_F : x \subseteq y \wedge i \in B \setminus y : x \cup \{i\} \in F \\ \rightarrow \quad & \exists i \in y \setminus x : (x \cup \{j\} \in F \wedge y \cup \{i\} \in F \end{aligned}$$

in words, given a base and one of its subsets, from which the base is accessible, if there is an element that "leads astray" (adding i to x makes it so that x will never reach y, which doesn't contain i) the subset from the base, there must be another one which keeps it on the right way and it must be feasible to exchange the two element in the bases.

Notice that the optimality of a constructive algorithm A depends on the properties of the problems (such as an additive objective function or bases as feasible solutions) **and** the properties of the search space F_A, that is of the algorithm.

6.4 Nonexact Constructive Heuristic Algorithms

When the required algebraic properties of a problem aren't satisfied, so the algorithm and the search space do not form a matroid or a greedoid with strong exchange axiom, one thing that can be done is trying to guide the behaviour of the algorithm inside the search space by using a refined sophistication of the selection criterion, forcing it to depend on both i and x, so $\varphi_A(i,x)$, as we'll see later. This may not prove that the algorithm reaches the optimum but quite often it will get *better* results and maybe some approximaition may be proven.

6.4.1 Examples of Constructive Heuristics

Constructive Heuristic Algorithm for the KP

In the KP the three axioms aren't satisfied, as the exchange axiom is not satisfied. In particular, the problem is that the object do not have the same volume and a small cardinality subset can't be in general enlarged by adding an element taken from a large cardinality subset. The problem is the capacity constraint or, better, the problem is the effect that the capacity constraint has on different items: promising objects have large value but also a small volume. Combining this natural idea one can consider the **unitary value** $\varphi_A(i,x) = \frac{\phi_i}{v_i}$ maximizing then the ratio of the

value and the volume instead of just the value. The resulting algorithm can perform very badly, but with a small modification is 2-approximated.

Let's give a numerical example with the following table, while considering the knapsack volume $V = 8$.

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1
ϕ/v	1.4	0.67	2	1.67	4	1

The algorithm performs the following steps.

1. $x := \emptyset, V_R = 8$
2. $x := \{e\}, V_R = 7$
3. $x := \{c,e\}, V_R = 5$
4. $x := \{c,d,e\}, V_R = 2$
5. $x := \{c,d,e,f\}, V_R = 1$
6. since $\Delta_A^+(x) = \emptyset$, terminate

The value of the solution found is 14, while the optimal solution is $x^* = \{a,c,e\}$ and its value is 15 - the algorithm, then, doesn't perform very badly. There are some critical cases, though. For example, let's follow the numerical example in the seguent table considering the volume $V = 10$.

B	a	b
ϕ	10	90
v	1	10
ϕ/v	10	9

The algorithm chooses the object a, which has higher ratio, then terminates. The solution has value 10, while the optimum is 90. There are instances with unlimitedly worse gap! The reason of the mistake is that the first discarded object has a large volume but also a large value.

Constructive Heuristic 2-approximated Algorithm for the KP

The approximated algorithm is constructive: it starts from an empty set and each time i is chosen to maximize the unitary value of the item, exactly like the previous algorithm. If the element respects the capacity the element is inserted in the regular algorithm; if, instead, the capacity is violated instead of rejecting the item the algorithm terminates in advance (so, it's not that good). The second modification is that the first discarded element is a solution in itself - it could clearly be improved by taking other elements, but we're not interested in that, we're interested in creating an algorithm with a guarantee. The algorithm returns the one of the two solution with the better values.

It's possible to prove that the sum of the two solution values overestimates the optimum, which is the first step to find approximation in maximization problems

$$f(x) + f(x') = \sum_{\tau=1}^{t} \phi_{i^\tau} \geq f^*$$

Algorithm 6 2-approximated Algorithm for the KP

```
1: procedure GREEDYKP(I)
2:     x := ∅
3:     while V > Σ_{i∈x} v_i do
4:         i^(t) := arg max_{i∈B\x^(t−1)} φ_{i^(t)} / v_{i^(t)}
5:         if v_{i^(t)} + Σ_{i∈x^(t−1)} ≤ V then
6:             x^(t) := x^(t−1) ∪ {i^(t)}
7:         else
8:             x' := {i^(t)}
9:             break
10:        end if
11:    end while
12:    return max[f(x), f(x')]
13: end procedure
```

this is demonstrated by supposing that one can enlarge the volume of the knapsack as soon as one finds an object to discard. The problem is slightly different and it surely has a larger optimal value - the sum of the two solution is at most equal to the optimal solution of the instance itself. And then finding a lower bound of a fixed ratio of the upper bound:

$$f_A = max[f(x), f(x')] \geq \frac{f(x) + f(x')}{2} \geq \frac{1}{2} f^*$$

the larger of the two number is certainly not smaller than the arithmetic mean of the two number, which in turn is larger than half of the optimum.

Constructive Heuristic Algorithm for the TSP

Consider the TSP on a complete graph $G = (N, A)$ with the search space defined as arc subsets with no subtour and induced degree ≤ 1 on all nodes. The Nearest Neighbour heuristic, introduced previously, always finds a feasible solution (which exists as the graph is complete) but which can be unlimitedly bad, as it's not a matroid or a greedoid with the strong exchange axiom.

Let's change the search space and define F_A as including all paths starting from a given node, for example node 1. These are promising sets of arcs and they have a nice structure: they just have to be completed in going back to the original node. Let N_x be the set of nodes visited in a path x. The acceptable extensions are all arcs going out of the last node of the path x and not closing a subtour

$$\Delta_A^+(x) = \{(h,k) \in A : h = \text{Last}(x), k \notin N_x \vee k = 1 \wedge N_x = N\}$$

in other words, the possible extensions of a given path x ending in node h are the arcs that go from h to another node k that does not belong to the current path unless $k = 1$ the first node and the path x visits all the nodes in the graph, so $N_x \equiv N$.

As for the axioms, in this case the trivial axiom obviously holds. The accessibility axiom holds as removing the laast arc yields a path starting from node 1, so there's always an element that can be removed from x remaining inside the search space. There's no hereditarity axiom as not all subsets are path (so it's not a matroid) and no exchange axiom (so it's not a greedoid).

Algorithm 7 NN Heuristic Pseudocode for the TSP

```
1: procedure NNTSP(I)
2:     x := ∅
3:     while true do
4:         (i, j) := arg min_{(h,k)∈Δ_A^+(x)} c_hk
5:         x := x ∪ (i, j)
6:         if j = 1 then
7:             break
8:         end if
9:     end while
10:    return x
11: end procedure
```

The algorithm starts with an empty set, which represents a degenerate path going out of node 1 and not reaching any other node. At each step it is found among all the possible extensions, which are the possible arcs that get out of the last node of the current path, the one with the minimum cost - the nearest neighbour.

It's a very intuitive algorithm and its complexity is quadratic, as it does n iterations in which arcs are chosen and in the end $n+1$ arcs will be chosen. If the triangular inequality holds, the algorithm is $log(n)$-approximated.

Numerical Example Consider the graph in Figure 6.6. Executing the algorithm on it starting

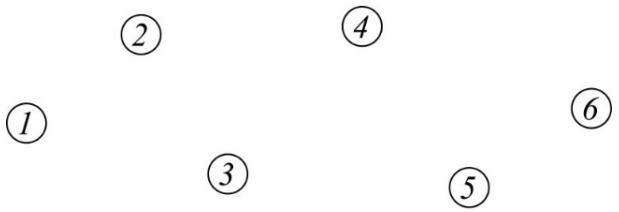
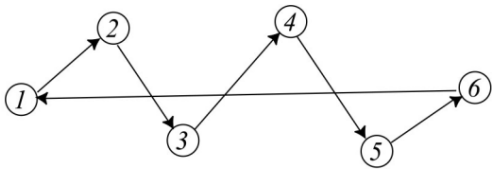
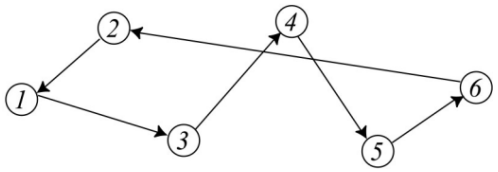


Figure 6.6: Topology of a graph. It's complete, the arcs aren't reported for clarity.

from node 1 finds the suboptimal path in Figure 6.7 (a) and starting from node 2 finds the suboptimal path in Figure 6.7 (b). Actually, starting from any of the node will yield a suboptimal solution, which cannot be found by this algorithm.



(a) NN heuristic starting from node 1.



(b) NN heuristic starting from node 2.

Figure 6.7: NN heuristics on a complete graph starting from different nodes.

6.4.2 Pure and Adaptive Constructive Algorithms

So far, we've always seen examples in which the selection criterion φ_A depends only on the new element i. These are called **pure** constructive algorithm. There are constructive algorithms,

called **adaptive**, in which φ_A depends on both i and the current solution x. Many criteria $\varphi_A(i,x)$ admit equivalent forms depending only on i: for example, in the KP, $\varphi_A(i,x) = f(x \cup \{i\})$ is equivalent to ϕ_i if it's expressed in additive way, just as in the TSP where $f(x \cup \{(i,j)\})$ is equivalent to c_{ij}. Let's now examine some examples of adaptive constructive algorithms.

Adaptive Constructive Algorithm for the SCP

Given a binary matrix and a cost vector which gives costs to the columns of the binary matrix, find a subset of columns that cover all the rows and have a total cost that's minimum. The objective function is obviously additive and the solution are not maximal subsets, actually the smaller feasible subsets are better. An adaptive selection criterion $\varphi_A(i,x)$ is necessary, as a pure one could repeatedly choose columns covering the same rows. The more promising ideas are to consider the objective function in order to select columns of low cost bu also some constraints: the columns have to cover "many" row and the newly selected columns have to cover new rows with respect to the ones in the current solution. In other words, $\Delta_A^+(x)$ includes only columns covering additional rows not in x.

The point is that perhaps it's better to avoid taking into account just the cost of the column thereby chosing the one with the minimum cost but also considering how many rows are covered for that cost. So, the selection criterion may be defined as

$$\varphi_A(i,x) = \frac{c_i}{a_i(x)}$$

where $a_i(x)$ is the number of rows covered by i but not by x. This is more complicated than the pure selection criterion, in which x didn't appear at all; this algorithm is then classified as as *adaptive greedy*.

This algorithm can find an optimal solution but may also fail to find one and actually this algorithm is $log(n)$-approximate. In fact, at each step t each column i is evaluated with the criterion

$$\varphi_a(i,x^{(t-1)}) = \frac{c_i}{a_i(x^{(t-1)}}$$

where each row j is covered by a certain column i_j at a certain step t_j. Let's start assigning weight $\theta_j = 0$ to each row j and then when each row j is covered at step t_j let's update its weight to

$$\theta_j = \frac{c_{i_j}}{a_{i_j(x^{(t_j-1)})}}$$

so that the total weight of the rows increases by c_{i_j} at step t_j; correspondingly, x includes column i_j and its cost increases by c_{i_j} - so that the sum of the weights of the rows is exactly the sum of the cost of the selected columns

$$f_A(x) = \sum_{i \in x} c_j = \sum_{j \in R} \theta_j$$

Since the column chosen is always that with minimum $\varphi_A(i,x^{(t-1)})$ and $a_i(x^{(t-1)})$ decreases, the row weights increase step by step. At step t there are $|R^{(t)}| \leq |R| - t$ uncovered rows (which is the maximum number of rows to be covered at each step) and the columns of the optimal solution could cover them all with cost f^*. This means that at least one of such columns has unitary cost $\leq f^*/|R^{(t)}|$. The point is that in general, when choosing a new column the one that is chosen is the one with minimum unitary cost. This cost is the minimum over all columns, so it's certainly smaller than or equal to the average cost that is given by the optimal solution.

Some column will have a cost smaller than this and some other will have a larger cost. So, the final solution will give that its cost won't be larger than the sum of the upper estimates in each steps of the algorithm

$$\theta_j \leq \frac{f^*}{|R^{(t_j)}|} \rightarrow \sum_{j \in R} \theta_j \leq \sum_{j \in R} \frac{f^*}{|R^{(t_j)}|}$$

in other words, the cost to cover each row j is not larger than the optimum divided by the number of rows uncovered at the step in which j gets covered.

Now, the number $|R^{(t)}|$ strictly decreases at each step and the sum can be overestimated reducing $|R^{(t)}|$ by 1 at each step. Finally, the approximation ratio is limited by a logarithmic guarantee:

$$f_A = \sum_{j \in R} \theta_j \leq \sum_{j \in R} \frac{f^*}{|R^{(t_j)}|} \leq \sum_{r=|R|}^{1} \frac{f^*}{r} \leq (ln|R|+1)f^*$$

thanks to the harmonic series.

Numerical Example Let's give an example of the approximation. Consider the SCP instance in Table 6.1. The algorithm takes the following steps:

c	25	6	8	24	12
A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	1	1	0
	1	0	0	1	0
	1	0	0	0	1

Table 6.1: An SCP instance.

1. since $\varphi_a(i,x) = [4.16\ 2\ 4\ 12\ 12]$, select $i := 2$ and set $\theta_1 = \theta_2 = \theta_3 = 2$ which is less than $f^*/|R^{(0)}| = 25/6 = 4.16$.
2. since $\varphi_a(i,x) = [8.3\ -\ 8\ 12\ 12]$, select $i := 3$ and set $\theta_4 = 8$ which is less than $f^*/|R^{(1)}| = 8.3$.
3. since $\varphi_a(i,x) = [12.5\ -\ -\ 24\ 12]$, select $i := 5$ and set $\theta_6 = 12$ which is less than $f^*/|R^{(2)}| = 12.5$.
4. since $\varphi_a(i,x) = [25\ -\ -\ 24\ -]$, select $i := 4$ and set $\theta_5 = 24$ which is less than $f^*/|R^{(3)}| = 25$.
5. all the rows are covered, therefore $\Delta_A^+(x) = \emptyset$ and the algorithm terminates.

$f_A = \sum_{j \in R} \theta_j = 50$ and the approximation holds:

$$f_A \leq (ln|R|+1)f^* \approx 2.79f^*$$

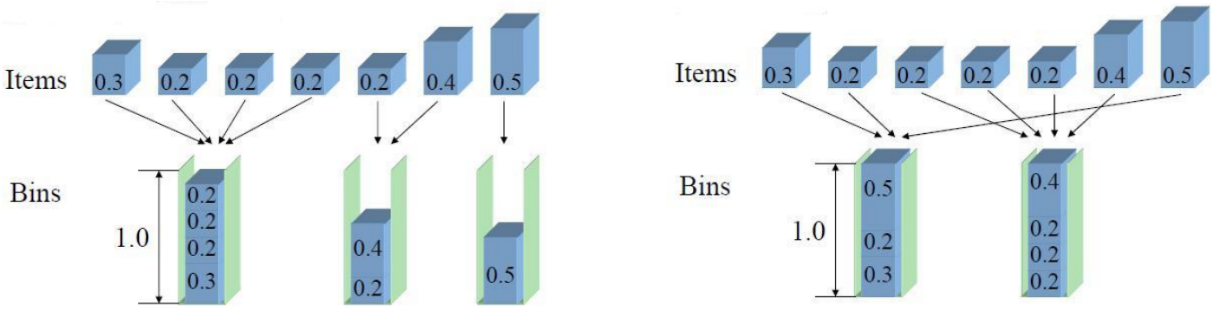
Adaptive Heuristics for the BPP

The BPP requires to divide a set O of voluminous objects into the minimum number of containers of given capacity drawn from a set C. The ground set $B = O \times C$ includes all the possible object-container assignments (i, j) with exactly one container for each object and with the total volume in each container not exceeding the capacity.

Let's give two different heuristics for this problems. These two are adaptive, so they will depend on the current solution; they are also examples of another aspect: the selection criterion is not always expressed with some numerical formula. Let's define the search space F_A as the set of all partial solutions: so, at a certain point in a solution not all objects may be inserted in a bin but each object that appears appears exactly one time and no volume constraint is violated.

The objective function is certainly not additive, but the greedy algorithm may be applied anyway, starting from an empty subset and then taking the assignment that minimizes the objective function: so, it tries to assign the first object to the first container and it finds that it needs one container. Then it tries to assign the first object to the second container and it finds that it needs one container... All "first assignment" require one container, so none can be chosen in this situation.

First-Fit Heuristic A very similar heuristic is the one colled **first-fit heuristic**. The difference is that instead of minimizing the objective function and then breaking ties based on the index in this case the criterion is not based on the objective function but the object are chosen based on the index. So, a pair $(object_i, container_j)$ is chosen such that i is the first (in terms of minimum index) unassigned object and j is the first used container with enough residual capacity for i or, if none has enough residual capacity, the first unused container.



(a) First Fit solution.

(b) Optimal solution.

Figure 6.8: Example of First-Fit heuristic on an instance.

The selection criterion is not an analytic function. The rest of the algorithm is the same: the first spare is chosen then it goes on and on until the solution can be extended, which can be done only as long as there are object not assigned to containers, so the needed number of steps is equal to the number of objects. Figure 6.8 (a) shows an example of the First-Fit heuristic on the BPP.

The solution is not optimal, as shown in Figure 6.8 (b). Further, it is approximated: at least $f^* \geq (\sum_{i \in O} v_i)/V$ containers are necessary (in fact the right part of the disequation is the number of volumes needed if the objects could be split) and the occupied volume is greater than $V/2$ for all used containers, possibly except the last one. This last thing is proved by contradiction: assume it is not so and two containers are less than half filled. Then the object in the second bin fits in the first, which contradicts the algorithm itself.

Then, the total volume exceeds that of the $f_A - 1$ "saturated" container

$$\sum_{i \in O} v_i > (f_A - 1)\frac{V}{2}$$

which implies

$$(f_A - 1) \leq 2\frac{\sum_{i \in O} v_i}{V} \leq 2f^* \rightarrow f_a \leq 2f^* + 1$$

Decreasing First-Fit heuristic The approximation ratio $\alpha = 2$ holds for any permutation of the objects and intuitio would suggest to select first the smallest objects, in order to keep the objective $f(x \cup \{i\})$ as small as possible. This neglects that all the objects must be assigned. In fact, it's better to select the largest object first, because each object in a container has a volume strictly larger than the residual capacity of all the previous container and keeping the smalles objects in the end guarantees that many containers have a small residual capacity. So, the idea is to apply the First-Fit heuristic by chosing a decreasing-volume permutation of the object to insert. This heuristic doesn't yield an optimal solution in general, but it's better than the previous heuristic, in fact it can be proved that the approximation ratio is better

$$f_A \leq \frac{11}{9} f^* + 1$$

This is actually a general consideration which holds for many other problem than the BPP.

6.5 Extensions to Constructive Heuristics

After introducting the basic schemes of constructive heuristics, the algebraic conditions necessary and sufficients that allow to prove that a certain constructive algorithm is actually exact, a number of cases in which constructive algorithm are non-exact but it's possible to improve the definition of the selection criterion, taking into account the constraints of the problem, so that the result can be reasonably good and modeling tools such as the construction graph it's possible, now, to expand the basic scheme in order to build different constructive heuristics.

There are two basic enhancings: the first is that the **construction graph** can be modified, having a more effective definition - an example is adding more than one element to the current subset x or add elements to x but also removinig some (still, a "net increment" has to occur). The second extension focuses on the **selection criterion**, making it not dependend on the element that's to be included in the solution and the current subset x but on some more sophisticated functions - we'll examine two possible extenstions, one using a *regret-based* function that estimates a potential future loss associated to the inclusion of a certain element in the subset and one using a *lookahead* function, in which the algorithm tries to estimate the final value of the objective function based on the addition or removal of certain elements to the solution x.

6.5.1 Extensions of the construction graph

The basic constructive algorithm adds an element at the time to the solution. In order to generalize this scheme, more than one element can be added at each step, so the idea is that the selection criterion actually depends not only on the current solution but on a possible set of elements B^+ of the ground set that aren't currently inside the solution, so that $B^+ \subseteq B, B \cap x = \emptyset \rightarrow B^+ \subseteq B \setminus x$ is to be added, instead of a single element, defining the selection criterion as $\varphi_A(B^+, x)$. Another possibility is to add some element and remove some element, so that the selection criterion will be a function of $\varphi_A(B^+, B^-, x)$ (instead of $\varphi_A(i, x)$ as it was before) where $B^+ \subseteq B \setminus x$ as before and $B^- \subseteq x$ and $|B^+| > |B^-|$.

These two possibilities have in common the hypotesis that the construction graph is still acyclic and defined with nodes corresponding to element of the search space and arcs corresponding to pairs of subsets such that the second one is reached from the first one by applying an acceptable extension. The point is to define families of subsets such that **optimizing the selection criterion is a polynomial problem**

$$\min_{B^+ \subseteq B \setminus x, B^- \subseteq x} \varphi_A(B^+, B^-, x)$$

Clearly, in this way the complication is to compute and minimize this selection createrion, which becomes quite difficult as the cardinalities of the subsets B^+ and B^- could become very numerous, quickly forcing an unwanted exponential task. The two classical ways to limit this complexity are first to limit the size of the subsets B^+ and B^-, e.g. $|B^+| = 2$ and $|B^-| = 1$ and secondly to define a specific familiy of sets such that the task of minimizing the selection criterion becomes a classic polynomial problem that can be optimized efficienly, such as minimum paths problem; we'll examine both cases.

A third search space for the TSP

After considering two heuristics, the basic greedy heuristic in which we considered as a search space all the subsets of arcs that respect the condition on the degree of the node - no node can have a degree larger than one both ougoint or ingoing and no subtours unless it covers all the nodes which has a very bad performance and the nearest neighbour heuristic based on the search space composed of all the paths starting in a node 1 and then visiting other nodes with the only possibility of getting back to the original node after having visited all the other nodes, we can introduce a third search space in which three different selection criteria can be useful.

The search space F_A is the set of all the circuits in the graph that include a given node: we're going to choose a node 1 without loss of generality. A circuit cannot be obtained from another by adding a single arc but can be obtained by adding two arcs $(i,k),(k,j)$ and removing one (i,j) to a previous solution. It is clear that the "new" circuit that visit a larger set of nodes with

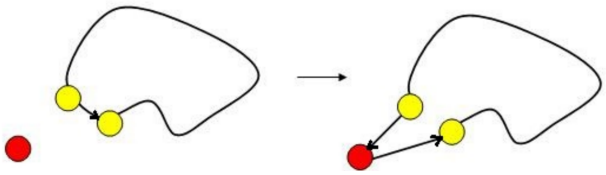


Figure 6.9: Building a new circuit from a given circuit.

respect to a given circuit, it cannot be built by adding a single arc from the original circuit. In this context, $B^+ = \{(i,k),(k,j)\}$ and $B^- = \{(i,j)\}$.

The idea is to start from an empty set of arcs $x^0 = \emptyset$, or, one could think of it as a degenerate single self loop on node 1, and then step by step one node outside the current set of nodes spanned by the current solution is chosen, one arc inside the solution is chosen and two arcs "detouring" to the to-be-inserted node. So, what's to choose is one arc to remove and one node to insert. The nice point is that after $n-1$ steps this will lead to an Hamiltonian circuit if the graph is complete.

Insertion Algorithms for the TSP This selection criterion depends on two subsets but fundamentally depends on $\mathbb{O}(n^2)$ choices. This is because the two objects to chose, an arc in the current solution and a node outside of it, can be chosen in many ways: there are in fact $(n-|x|)$ nodes that can be chosen and $|x|$ arcs that can be removed so, overall, $(n-|x|)|x|$ choices can be made. The **Cheapes Insertion** (CI) heuristic uses a selection criterion

$$\varphi_A(B^+,B^-,x) = f(x \cup B^+ \setminus B^-)$$

that is exactly the objective function $f(x)$ which is additive, hence extensible to the whole of F_A. Since $f(x \cup B^+ \setminus B^-) = f(x) + c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}}$ where $k \in B^+$ and $(s_i,s_{i+1}) \in x$ as represented in Figure 6.10, in order make the two choices i and k (the arc to be removed and the

node to be inserted) the original cost of the solution $f(x)$ is not important since it is constant for all evaluations, but is important to minimize the net cost of the modification:

$$\arg\min_{B^+,B^-} \varphi_A(B^+,B^-,x) = \arg\min_{i,k}(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$$

hence, the computational cost of evaluating φ_A decreases from $\theta(n)$ to $\theta(1)$. This is fundamen-

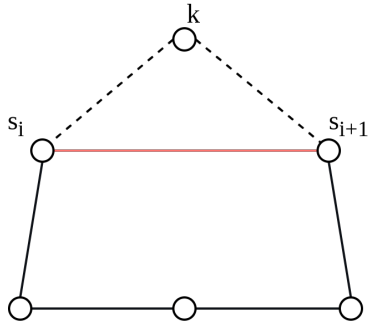


Figure 6.10: Example of an insertion.

tal, as this step must be as fast as possible. So, what is to be done is to estimate $\mathbb{O}(n^2)$ alternatives where each alternative is certainly feasible, since we're considering complete graphs only, and the objective function can be updated not from scratch in constant time. Let's consider the scheme of the algorithm:

1. start with an empty set of arcs: $x^{(0)} = \emptyset$ representing a degenerate circuit centered on a certain node 1
2. select the arc $(s_i, s_{i+1}) \in x$ and the node $k \notin N_x$ (the set of nodes spanned by x) such that $(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$ is minimized
3. if the circuit does not visit all nodes go back to point (2), otherwise terminate

This algoritm is not exact but it is 2-approximated for graphs that respect the triangle inequality (so, not in general). The CI algorithm performs $n-1$ steps and at each step t it evaluates $(n-t)t$ pairs $(node, arc)$ in constant time, performs the best addition/removal and decides whether to terminate. So, the overall complexity is $\Theta(n^3)$ and it can be reduced to $\Theta(n^2 * log(n))$ collecting in a min-heap the best insertion cost for each external node: each of the n steps select the best insertion in $\mathbb{O}(1)$ time performing it creating two new insertions for each external node, the better of which possibly improves the value saved in the heap; each of the $\mathbb{O}(n)$ improvements takes $\mathbb{O}(log(n)$ time.

Nearest Insertion heuristic for the TSP The CI algorithm tends to select nodes clos to the circuit x: minimizing $(c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$ implies that $c_{s_i,k}$ and $c_{k,s_{i+1}}$ are small. In the **Nearest Insertion** heuristic the search space F_A is defined exactly as before but the selection criterion is different. We've already seen that in the case of *decreasing first fit* and *first fit* that the selection criterion is not always a function that can be expressed analytically; in the NI the selection criterion is split in two parts (as for the dff): the first step applies an *add* criterion to choose the node k to add

$$k = \arg\min_{l \notin N_x}(\min_{s_i \in N_x} c_{s_i,l})$$

and the second setp applies a *delete* criterion to choose which arc to break

$$(s_i, s_{i+1}) = \arg\min_{(s_i,s_{i+1}) \in x} (c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$$

in other words the two choices are made in two separate steps: the idea is that instead of minimizing the new cost, the closest external node is chosen and then the "right" arc to break is chosen instead of trying to minimizing the new cost testing all the external k nodes. This two steps repeat until the circuit visits all the nodes and, surprise surprise, it is not exact but 2-approximated under the triangle inequality. This algorithm performs $n-1$ steps: at each step t it evaluates the distance of $(n-t)$ nodes from the circuit, each one in $\Theta(t)$ time; then, it selects the node at minimum distance, evaluates the removal of t arcs each one in $\Theta(1)$ time, performs the best addition/removal and decides whether to terminate. So, the overall complexity is yet again $\Theta(n^2)$ but it can be reduced to $\Theta(n^2)$ collecting in a vector for each external node the closes internal node: each of the $n-1$ steps selects the closest node in $\mathbb{O}(n)$ time, inserts it creating two new insertions for each external node, the better of which possibly improves the value saved in the vector; each of the $\mathbb{O}(n)$ improvements takes $\mathbb{O}(1)$ time.

This algorithm has the same approximation guarantee that the CI has, but it's certainly faster.

Farthest Insertion heuristic for the TSP This heuristic is very similar to the last one: the idea is that nodes to be inserted are chosen maximizing the minimum distance, so that the most problematic nodes - the farthest ones - are served in the best way. Formally, the algorithm **Farthest Insertion** (FI) starts with an empty set of arcs $x^{(0)} = \emptyset$ representing a degenerate circuit centered on node 1. Selects the farthest node k from the cycle x (*Add* criterion)

$$k = \arg\max_{l \notin N_x} (\min_{s_i \in N_x} c_{s_i,l})$$

so, the node that is the farthest from the closest node of the cycle and then selects the arc (s_i, s_{i+1}) (*delete* criterion)

$$(s_i, s_{i+1}) = \arg \min_{(s_i,s_{i+1}) \in x} (c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}})$$

and if the circuit does not visit all nodes it selects another one, otherwise terminates. It can be proved that this algorithm is $log(n)$-approximated under the triangle inequality, hence worse than the previous ones in the worst case but often experimentally better and its complexity is exactly the same as the NI heuristic, so $\Theta(n^2)$. In a way, this is the same philosophy seen in the decreasing first fit algorithm for the BPP.

6.5.2 Extension of the construction graph using auxiliary subproblems

A second possible extension of construction graphs consists in adding some element chosen by the use of some auxiliary subproblem.

The Steiner Tree Problem

To illustrate this second possibility, let's introduce a new problem called the **Steiner Tree Problem** (STP). Given an undirected graph $G = (V, E)$, a cost function $c : E \to \mathbb{N}$ on the edges and a subset of special vertices $U \subset V$, the problem is solved by finding a tree connecting at minimum cost all the special vertices. An example of STP is represented in Figure 6.11, where the special nodes U are represented in red. The non-special vertices can be included or not arbitrarily. It seems that the solution to this problem is just to consider the special vertices canceling the other and finding the minimum spanning tree: the example in Figure 6.11 shows how that is not the right tactic in general. A possible heuristic for the STP is, considering a classic constructive algorithm, starting with an empty set of arcs and then adding some edges iteratively building

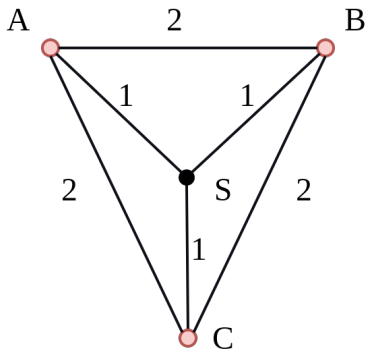


Figure 6.11: An example of STP instance.

a solution. This doesn't work very well because considering a search space such as the search space of Kruskal, that is the set of all forest, or a search space such as the search space of Prim, that is the set of all trees including a given starting vertix, adding one edge at a time yields solutions with redundant edges and has a hard time distiguishing useful and redundant edges - you know that the edge is cheap but not if it's useful.

Distance Heuristic for the STP The difficult point is exactly to distinguish edges that are useful and redundant edges besides considering the cost. The **Distance Heuristic** (DH) focuses on the fact that the special vertices have to be spanned by the solution, so the idea is to start with a special vertex 1 adding a new special vertex j connected with a path; in other words the DH iteratively adds a path B^+ from the subset solution (a tree) x to a special vertex instead of a single edge so that x remains a tree and x spans a new special vertex; the minimum cost path can be computed efficiently at each step (Dijkstra or any other shortest path algorithm).

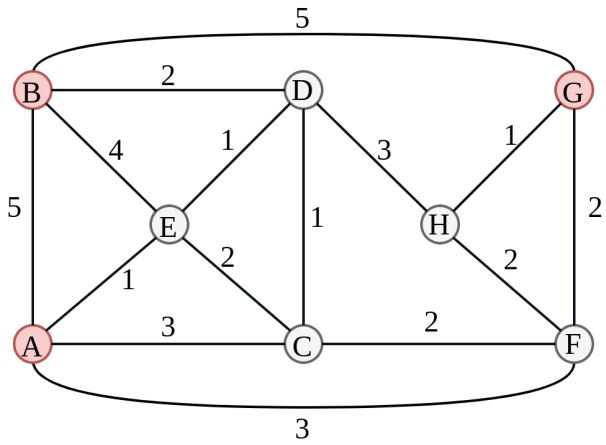


Figure 6.12: A second example of STP instance.

Let's look at an execution of the algoritm on Figure 6.12.

1. start with a single special vertex A: $x^{(0)} := \emptyset$, a degenerate tree.
2. add the closest special vertex B through path (a,e,d,b): $x := \{(a,e),(e,d),(d,b)\}$.
3. add the closes special vertex G through path (g,h,d): $x := \{(a,e),(e,d),(d,b),(g,h),(h,d)\}$.
4. all special vertices are in the solution: terminate.

In general, the DH algorithm is 2-approximated, even though in this example the optimal solution is obtained. It is equivalent to computing a minimum spanning tree on a graph with vertices reducet to the special vertices and edges corresponding to minimum paths.

Let's see a **counterexample to optimality** regarding the DH algorithm for the STP. Consider

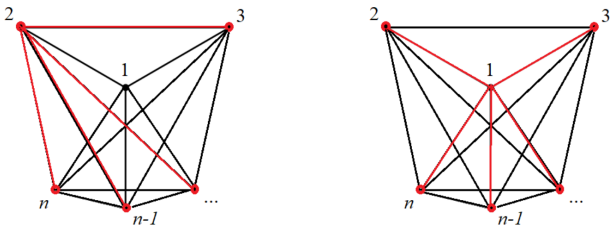


Figure 6.13: Counterexample to DH optimality.

the complete graph represented in Figure 6.13, where all vertices except one, the vertex 1, are special, so $U = V \setminus \{1\}$. Let the cost function be

$$c_{uv} = \begin{cases} (1+\varepsilon)M & u = 1 \vee v = 1 \\ 2M & u, v \in U \end{cases}$$

where M is just used to obtain integer cost for any ε. We can see that the optimal solution is the spanning star centered in 1

$$f^* = (n-1) \cdot (1+\varepsilon)M$$

(Figure 6.13 on the right) while the DH returns a star spanning the special vertices:

$$f_{DH} = (n-2) \cdot 2M$$

(Figure 6.13 on the left). The approximation ratio is

$$\rho_{DH} = \frac{f_{DH}}{f^*} = \frac{n-2}{n-1} \cdot \frac{2}{1+\varepsilon} < 2$$

and converges to 2 as n increases and ε decreases.

6.5.3 Extensions of selection criterion

We've seen extensions of constructive algorithm based on the use of different construction graphs, with arcs corrisponding not to a single element but many elements or additions and removal, sometimes exhaustingly explored and sometimes by solving some auxiliary problem, such as shortest path; we could also consider a sophisticated selection criterion: two possibilities were anticipated, the regret-based function and the lookahead function. Let's explore this two possibilities.

Regret-based constructive heuristics

We've seen that in a constructive algorithm the decision taken in the first step may strongly influence the rest of the computation, in particular they can restrict very severely the future feasible choices. Consider the case in which some object is inserted in some container: quite clearly, the possibility to put other object in that container is being limited by the insertion itself - this is the case for the BPP but also other problems suffer this. In the TSP servicing very well

nearby nodes means that long routes will be taken to reach the other nodes. In the CMST all vertices have to be spanned but the maximum capacity on each subtree has to be respected and early links could make some subtrees unavailable for later vertices.

Sometimes, the selection criterion tries to take this into account implicitly: in the BPP we've seen the DFF heuristic in which the assignment of objects starts from the larger objects; for the TSP we've seen the FI heuristic in which the farthest nodes are visited first. Some selection criteria aim explicitly to leaver larfer sets of good choices for the last steps.

A typical **regret-based** heuristic consists in:

1. partitioning $\Delta_A^+(x)$ into disjoint classes of choices, for example in the case of BPP the ground is made of pairs $\langle object, container \rangle$ so there are lots of possibilities and they can be divided into classes that refer to each object: all the possible assignments of the first object, all the possible assignments of the second object...
2. compute a basic selection criterium for all choices, that is the apply the selection criterion for each choice in every different class, keeping them separated (the best possible assignment among all the assignments of the first object, the best possible assignment among all the assignments of the second object...)
3. compute for each class the **regret**, that is the difference between the second-best choice and the best choice in each class or, alternatively, the average of the other choices (possible weighted) and the best choice. This estimates the damage incurred by postponing the best choice.
4. choose the best choice of the class for which the regret is maximum. Why the maximum? Because a big regret means that if you don't choose the best option then you'll have to choose the second (or another one) choice which is much worse - in other words you fear that not taking this choice now means that you will be forced to take other choices. This sort of reasoning is done when it is sure that at least one choice for each class can be taken - serving all the nodes in a graph, putting all the objects in containers, performing every task..

Example on the CMST Consider the CMSTP and ground set $B = V \times T$, that is the pairs $\langle vertex, subtree \rangle$, meaning that each vertices can belong to a specific subtree. Let the vertex weight be uniform $w_v = 1 \quad \forall v \in V$ and capacity $W = 2$. The special vertex is r and the so-

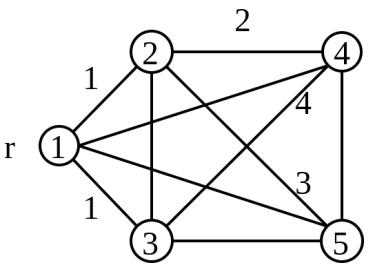


Figure 6.14: An instance of CMST problem. Arcs without explicit weight have weight 100.

lution is a spanning tree such that all the subtrees that are appended to the root node r have a total weight not larger than the capacity. Let the search space F include all partial solutions and the instance be the one represented in Figure 6.14. We can start from the node r; what are our choices? In the first iteration, each vertex can belong to one out of four subtrees: $(1,1),(1,2),(1,3),(1,4),(2,1),\cdots(4,4)$ with different costs. Let's say that, following a greedy

fashion, we put the vertex 2 in the subtree 1, vertex 3 in subtree 2, since adding 3 to the second subtree would mean adding a cost of 100. 4 could be put in another subtree with a weight of 100, but it can be added to the subtree 1 with vertex 2. Vertex 5 has to go in subtree 2 with 3, since the subtree 1 is at capacity (or in a third subtree, but the cost is the same) so

$$c(x) = 1 + 1 + 2 + 100 = 104$$

The regret algorithm puts vertex 2 in subtree 1 and vertex 3 in subtree 2; then:

- the regret of vertex 4 is the difference $c(4,2) - c(4,1) = 4 - 2 = 2$
- the regret of vertex 5 is the difference $c(5,2) - c(5,1) = 100 - 3 = 97$

So the algorithm puts vertex 5 in subtree 1 with 2 and vertex 4 in subtree 2 with vertex 3 and

$$c(x) = 1 + 3 + 1 + 4 = 9$$

having applied the regret definition based on the second choice.
The regret technique works very well when the problem is tight with strong constraints and doesn't work very well in other cases and could lead to non effective choices.

Lookahead-based constructive heuristics: Roll-out heuristics

Roll-out heuristics are based on the idea of making a look ahead, that is if a new element is chosen and it is added to the solution, what would happen in the end? These heuristics have been proposed in 1997 by the two scholars Bertsekaz and Tsitsiklis. These heuristics are *meta*-algorithm, meaning that they are based themselves on a certain constructive algorithm A and

- start with an empty subset $x^{(0)} = \emptyset$
- at each step t
  - extend the subset in each feasible way: $x^{(t-1)} \cup \{i\} \;\; \forall i \in \Delta_A^+(x)$
  - apply the basic heuristic to each extended subset and compute the resulting solution $x_A(x^{(t-1)} \cup \{i\})$
  - use the value of the solution as the selection criterium to choose $i^{(t)} \;\; \varphi_A(i,x) = f(x_A(x^{(t-1)} \cup \{i\}))$
- terminate when $\Delta_A^+(x)$ is empty

in other words, try every feasible move, look at the result then go back and choose the best move - so, lots of attempts are done and the basic heuristic is run lots of times. The algorithm proposed in the paper by Bertsekaz and Tsitsiklis proves that under very general conditions the result of the roll-out heuristic dominates that of the basic heuristic. The complexity remains polynomial, but it is much larger: in the worst case, in fact,

$$T_{roA} = |B|^2 T_A$$

c	25	6	8	24	12
A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	0	1	0
	1	0	0	1	0
	1	0	0	0	1

Table 6.2: An instance of the SCP.

Example: roll-out for the SCP Given the SCP instance in Table 6.2, the solution is set of column that cover all the rows with minimum cost. We've already seen a good log-approximated algorithm based on the ratio of cost divided by the coverage rows that is adaptive.

Applying roll-out, each of the five column is taken as part of the solution and then the original heuristic algorithm is applied to it:

1. start with the empty subset $x^{(0)} = \emptyset$
2. for each column i, applu the constructive heuristic starting from subset $x^{(0)} \cup \{i\} = \{i\}$
   - for $i = 1$, obtain $x_a(\{1\}) = \{1\}$ of cost $f_A(\{1\}) = 25$
   - for $i = 2$, obtain $x_a(\{2\}) = \{2,3,5,4\}$ of cost $f_A(\{2\}) = 50$
   - for $i = 3$, obtain $x_a(\{3\}) = \{3,2,5,4\}$ of cost $f_A(\{3\}) = 50$
   - for $i = 4$, obtain $x_a(\{4\}) = \{4,2,5\}$ of cost $f_A(\{4\}) = 43$
   - for $i = 5$, obtain $x_a(\{5\}) = \{5,2,3,4\}$ of cost $f_A(\{5\}) = 50$
3. the best solution is the first one, therefore $i^{(1)} = 1$
4. all rows are covered and the algorithm terminates

The scheme can be generalised by applying several basig heuristics $A^{[1]}, \cdots, A^{[l]}$ and by increasing the number of lookahead steps, i.e. using $x^{(t-1)} \cup B^+$ with $|B^+| > 1$ - both of these increment the computation time, and the overall scheme does not change significantly:

- start with an empty subset $x^{(0)} = \emptyset$
- at each step t
  - for each possible extension $B^+ \in \Delta_A^+(x^{(t-1)})$ apply each basic algorithm $A^{[i]}$ starting from $x^{(t-1)} \cup B^+$.
  - the selection criterium is $\min_i f_{A^{[i]}}(x^{(t-1)} \cup B^+)$
  - use the value of the solution as the selection criterium to choose $i^{(t)}$ $\varphi_A(i,x) = \min_i f_{A^{[i]}}(x^{(t-1)} \cup B^+)$
- terminate when $\Delta_A^+(x)$ is empty

6.5.4 Destructive Heuristics

Destructive heuristics are complementary to constructive ones. The general scheme is:

- start with the full ground set: $x^{(0)} := B$
- remove an element a at time selected
  - so as to remain within the search space F_A, in other words $\Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in F_A\}$
  - so as to maximize a selection criterium $\varphi_a(i,x)$, usually a cost reduction
- terminate when $\Delta_A^+(x) = \emptyset$

A destructive heuristic for minimization problems can be described as in Algorithm 8. This

Algorithm 8 Destructive Heuristic Pseudocode

```
1:  procedure STINGY(I)
2:      x := B
3:      x* := B
4:      if x ∈ X then
5:          f* := f(x)
6:      else
7:          f* := +∞
8:      end if
9:      while Δ_A^+(x) ≠ ∅ do
10:         i := arg max_{i∈Δ_A^+(x)} φ_A(i,x)
11:         x := x \ {i}
12:         if x ∈ X and f(x) < f* then
13:             x* := x
14:             f* := f(x)
15:         end if
16:     end while
17:     return (x*, f*)
18: end procedure
```

algorithm is often also called *stingy* as it starts with the entire ground set in x and then starts throwing "expensive" objects away. One case in which this heuristic could be relevant is the SCP, starting with all columns and iteratively throwing away unnecessary expensive rows until no other can be thrown away. In the MST problem the stingy algorithm is computationally expensive but is as functional as Kruskal's or Prim's algorithm; starting from the full set of edge, costly edges get iteratively discarded from the solution provided that it still spans all the vertices (and visits are more costly than union-find procedures like Kruskal's or not computing its feasibility at all like Prim's).

Why are they less used?

When the solutions are much smaller than the ground set, so $|x| << |B|$ a destructive heuristic requires a larger number of steps, is more likely to make wwrong decision at an early step and sometimes requires more time to evaluate $\Delta_A^+(x)$ and $\varphi_a(i,x)$.

Nontheless, when a constructive heuristic returns redundant solutions, it is useful to "append"a destructive heuristic at its end as a post-processing phase. This auxiliary destructive heuristic starts from a solution x of the constructive heuristic instead of B, adopts as a search space the feasible region:

$$F_A = X \implies \Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in X\}$$

and adopts as the selection criterion the objective function

$$\varphi_A(i,x) = f(x \setminus \{i\})$$

usually, it terminates after very few steps.

Constructive/destructive heuristic for the SCP Consider the following SCP instance:

c	25	8	24	12
A	1	0	0	0
	1	0	0	0
	1	1	0	0
	0	1	1	0
	0	0	1	0
	0	0	0	1

The constructive heuristic selects, in order, columns $1,2,4$ and 3, which is redundant as 2 can be removed. The auxiliary destructive heuristic removes column 2 and provides the optimal solution $x^* = \{1,3,4\}$.

CHAPTER 7

Constructive Metaheuristics

After extending the basic scheme of constructive algorithms, that is modifying the construction graph and the selection criterion but keeping the idea of starting from the empty set and slowly, iteratively, enlarging a "current" solution subset until a final solution is reached, we can think now of iterating the scheme introducing random steps or memory-based steps that will allow us to generate many different solutions.

7.1 Introduction to Constructive Metaheuristics

First of all, let's try and understand why the use of metaheuristics could be necessary. In general, because constructive algorithms, unless they are provably optimal, often have strong limitations and the idea is, then, to generate many solution thus improving the effectiveness of the algorithm. Of course, this will also decrease its efficiency, as the computational time will be given by the sum of every execution of the "basic" constructive algorithm - so, the tradeoff between efficiency and effectiveness must be tuned depending on the available resources. The simplest case of metaheuristics is simply running different algorithms considering different search spaces and different selection criteria: this technique is called *multistart* (or *restart*). After introducing this simple initial class, we'll introduce the two main mechanisms featured in metaheuristics, that is the use of **randomization**, as in the case of semigreedy algorithms, GRASP (*Greedy Randomized Adaptive Search Procedure*) and Ant Systems and the use of **memory** - that is the exploitation of solutions found in the previous iterations - as in the case of the ART (Adaptive Research Techniques), cost perturbation and, again, Ant Systems.

7.1.1 Multistart

Multistart is a classical, very simple and natural approach and it's based on the idea of having several different heuristics and running all of them in sequence and then returning the best solution found overall:

- define different search spaces $F_{A^{[l]}}$ and selection criterion $\varphi_{A^{[l]}}(i,x)$
- apply each resulting algorithm $A^{[l]}$ to obtain $x^{[l]}$
- return the best solution $x = \arg\max_{l=1,\cdots,\ell} f(x^{[l]})$

Each of the algorithm $A^{[l]}$ has its own search space and its selection criterion - we can think about using completely different $A^{[l]}$ algorithms or using the same algorithm that uses a numerical parameter with different values. A technical consideration, important if it's needed to extend

the concept of construction graph of constructive heuristics to constructive metaheuristics, is that the construction graph can model the situation of multistart including all nodes and arcs admitted by at least one algorithm $A^{[l]}$:

$$F_A = \bigcup_{l=1}^{\ell} F_{A^{[l]}}$$

then setting arc weights depending on l: $\varphi_A(i,x,l) = \varphi_{A^{[l]}}(i,x)$ and setting and infinite arc weight for arcs that are forbidden in a specific algorithm $A^{[l]}$: $\varphi_A(i,x,l) = +\infty$. In other words, the search spaces of the algorithms are "merged" in a specific way, considering all the subset that can be visited by at least one algorithm, defining the weights of the arcs following the original search spaces and giving weight to arcs that connect a node with an unreachable node (a subset that's not considered by any of the algorithms), so that $A^{[l]} = +\infty$.

For clarity, it's important to specify that given the ℓ possible algorithms (so there are at most ℓ different possible selection criteria and search spaces) one $l \in \ell$ among al possible algorithms is chosen for each run, meaning that the choice of the algorithm is changed halfway through an execution.

Example

Considering the TSP three different constructive heuristics were discussed previously: the Cheapest Insertion, the Nearest Insertion and the Farthest Insertion, each based on the search space composed by circuits starting and ending in some node 1, using different selection criteria φ.

To construct a metaheuristic algorithm using the CI, NI and FI constructive heuristics a possibility is to split the choice of the new node to insert and the choice of the arc to break in two steps, the first being the *insertion* step and the second being the *selection* step defined as follows. In the *insertion step*, instead of choosing a specific function, we may use a parametric function such as

$$i_k^* = \arg\min_{i \in \{1,\cdots,|x|\}} \gamma_{i,k} = \mu_1(c_{s_i,k} + c_{k,s_{i+1}}) - (1-\mu_1)c_{s_i,s_{i+1}}$$

where $\mu_1 \in [0,1]$ tunes the relative strenght of the increase in cost due to the added node k and the strength of the decrease in cost due to removed edge (s_i, s_{i+1}); so, instead of considering a *variation* of the cost given by the insertion of the new node in place of the arc that's to be broken, the proposed parametric function uses a value μ_1 to weight the cost of the addition of a certain node in the solution and to weight the removal of a certain arc from the solution; in general, the weights could be given by two different parameters, but it's better to keep the number of parameters low and use normalizations, such as the convex combination μ_1 and $1-\mu_1$, as their sum is 1. Setting $\mu_1 = 0$ means that only the cost of the removed arc is considered; on the contrary, setting $\mu_1 = 1$ means that only the cost of the two added arcs is considered. Intermediate values can *tune* the weighting of the first and second terms and, in particular, setting $\mu_1 = 1/2$ sets $\gamma_{i,k}$ equal to the objective function. This is just a possible idea: a reason for this might be that the arc that is to going to be removed is removed forever and the two added arc may be removed in the future, making the first part "less safe".

In the *selection step*, a parametric selection criterion such as

$$k^* = \arg\min_{k \in N \setminus N_x} \varphi_A(k,x) = \mu_2 d(x,k) - \mu_3 d(x,k) + (1-\mu_2-\mu_3)\gamma_{i_k^*,k}$$

where $\mu_2, \mu_3 \in [0,1]$ and we can summarily say that they are used to tune the relative strength and sign of the distance of the added node k from the current circuit x and the increase in cost due to added node k; the node that is to be added is selected considering the insertion cost, weighted or unweighted and/or the *distance* of the node that is to be chosen from the nodes in the solution circuit (or, as seen in laboratory, the inverse or even the mean of the two); again the two parts can be weighted using some weights; in the previous formula not normalized as we may want both to add and subtract the distance, depending on the chosen algorithm. Using such weights different situations can be generated: for example, setting $\mu_2 = 1/2$ the distance portion is null and only the insertion cost is considered, yielding the CI algorithm provided that $\mu_1 = 1/2$; setting $\mu_2 = 0$ the insertion cost is not considered but only the distance is considered and will be minimized, yielding the NI algorithm provided that $\mu_1 = 0$; finally, setting $\mu_2 = 1$ the distance is subtracted to the insertion cost, thus maximizing the distance, obtaining the FI algorithm provided that $\mu_1 = 1$.

7.2 Metaheuristic techniques

The main constructive metaheuristics that are going to be presented and cover, more or less, all the possibilities proposed in literature, are:

- **Adaptive Research Technique** (ART) also called *Tab Greedy*, where the idea is to restrict F to $F^{[l]}$ forbidding some moves based on the previous results

$$\min_{i:x\cup\{i\}\in F^{[l]}} \varphi_A(i,x) \text{ with } F^{[l]} = F^{[l]}(x_A^{[1]}, \cdots, x_A^{[l-1]})$$

- **semigreedy** and **GRASP**, where the idea is to use a randomized selection criterion

$$\min_{i:x\cup\{i\}\in F} \varphi_A^{[l]}(i,x,\omega^{[l]})$$

- **Ant System**, where the idea is to use a randomized selection criterion depending on the solutions of the previous iterations

$$\min_{i:x\cup\{i\}\in F} \varphi_A^{[l]}(i,x,\omega^{[l]}, x_A^{[1]}, \cdots, x_A^{[l-1]})$$

The ART uses memory, the GRASP randomization and the AS both.

7.2.1 Adaptive Research technique

The ART was proposed for the CMSTP by Patterson et al. in 1998, and after that it was never taken into account in the literature, so, in a sense, it is a failure; nonetheless it is interesting from several points of view and, probably, the method wasn't exploited at its best by the authors. The starting point is that applying a constructive heuristic, sometimes bad results are achieved as in the first steps of the algorithm some elements that are deceivengly good are added but force bad results: an example of this situation is the KP, in which an object at a very good unitary cost could be inserted in the solution but then other elements couldn't be included, ending in a bad result. What can be done is refusing to insert some good element if they are deceivengly good: the problem, of course, stands in deciding whether or not an element will yield a bad result in the long run. An idea to achieve that was proposed by the *roll-out* approach and it consists in

making a lookahead, trying one by one all the possibilities seeing where, on the construction graph, the algorithm ends, but this is a very costly approach. A different approach is to *forbid* an element hoping that in that way something else is chosen and the flow is driven on the right path instead of the wrong one.

Diversification and Intensification

The point is still the same: how to identify the element(s) that must be chosen and forbidden? In this way, some **diversification** is being imposed upon the search; diversification is a general concept which consists in trying to find solutions that are different from solutions that were previously found: this is something that drives the search in the solution space from the region in which the algorithm is to different regions. It is also known in textbooks as *exploration*, but it is exactly the same concept. A dual of this concept, further analyzed later, is the concept of **intensification**: after having found a "reasonably good" solution and for some reasong it's thought that better solutions are similar to the ones previously found, the search is intensified to look *around* some chosen previous solution, looking for similar ones. This concept is also known as *exploitation*.

The two concepts are complementary and we must not *choose* one of them but they must be combined, finding the right tradeoff. In the ART, the idea is that they are tuned by the number of forbidden elements: forbidding many elements means that algorithm must look for very different solutions - thus diversifying; on the contrary, forbidding few elements means that the algorithm must look for something only slightly different, thus intensifying the research.

Another fundamental point is that the introduced proibitions must be temporary, and indeed a parameter in the ART is the **expiration time** L; this is because permanently and continuously forbidding elements will make impossible to find feasible and optimal solutions.

In detail, the ART starts by defining a basic constructive heuristics A, thus making in a sense the ART a meta-algorithm. Let, then, T_i be the starting iteration of the prohibition for each element; in other words T_i is a vector where at index i there's a number that's the index of the iteration in which choosing a certain element i is forbidden. At first no element is forbidden, so it's necessary to set $T_i = -\infty \ \forall i \in B$. At each iteration $l \in \{1, \cdots, \ell\}$

1. apply heuristic A forbidding all elements such that $l \leq T_i + L$, thus, all proibitions older than L automatically expire; let $x^{[l]}$ be the resulting solution
2. if $x^{[l]}$ improves the best solution found, save it and save all values $T_i - l$, as it will tell which solution were forbidden and which solutions were allowed; in this way is possible to know how such a solution was obtained, so to intensify the search around that solution
3. decide, with probability π, whether to forbid or not each $i \in x^{[l]}$ and set $T_i = l$ for each forbidden element
4. (possibly) make auxiliary changes to L, π or T_i

in the end, the best solutiond found will be returned.

Example: ART for the SCP Given the SCP instance in Table 7.1, where the optimal solution consists in the first column while a greedy algorithm would choose the columns $\{2,3,5,4\}$ columns obtaining a bad solution.

Applying the ART technique

1. the basic heuristic finds the solution $x^{[1]} = \{2,3,5,4\}$ or cost $f(x^{[1]}) = 50$; forbid, at random, column 2.
2. the basic heuristic finds the solution $x^{[2]} = \{3,1\}$ or cost $f(x^{[2]}) = 33$; forbid, at random, column 3.
3. the basic heuristic finds the solution $x^{[3]} = \{1\}$ or cost $f(x^{[3]}) = 25$, optimal.
4. ...

Of course, an unlucky run could forbid column 1 at step 2.

c	25	6	8	24	12
A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	0	1	0
	1	0	0	1	0
	1	0	0	0	1

Table 7.1: An instance of the SCP.

Parameters of the ART

The ART has at least three parameters: the total number of iterations ℓ, tuned mainly by the available time, the length L of the prohibition and the probability π of the prohibition. The last two are technical parameters that influence the performance of the algorithm in the single iteration; they can be assigned by experimental campaings that tend to be very long, because for each parameter one must consider all possible combinations (this is a reason why minimizing parameters is a good idea); further, experimental evaluation may lead to **overfitting**, that is labelling as absolutely good values which are good only on the benchmark instances considerated. The excess of parameters is an undesirable aspect, which often reveals an insufficien study of the problem and of the algorithm.

Intensification

Let's spend some more words on intensification. An excessive diversification can hinder the discovery of the optimum. Intensification means to focus the search on more promising subsets and, as previously said, they play complementary roles. In the ART they can be introduced by tuining π and L - in particular, given small values π_i and a short expiration time L_i this means that only a few elements and only for a short time are to be forbidden, always obtaining similar solution, thus intensifying the search. On the contrary, considering large probabilities and large expiration times a lot of elements are removed for many itearions, so every time the algorithm tries to go far away and never come back - this is diversification. Of course, they could be related to features of **data**, for example deciding that the π_i and L_i should be smaller and shorter depending on how good the element i (e.g., large value or small cost); the parameters could be also tuned based on **memory**, that is learning which parameters values are good by reducing the expiration time and the reducing the probability for elements that belong in the best known solution. Another possibility to intensify the search could be to restart the algorithm starting with the same $T_i - l$ values that have been associated with the best solution, so to obtain the exactly solution and then move in a region close to it.

7.2.2 Semi-Greedy Heuristics

The second approach that we're going to discuss is the **semigreedy** (or semi-greedy) approach, where the idea is always the same: a constructive algorithm can be non exact because at one step it takes the wrong choice - and this is the problem, why has it taken the wrong choice? The basic algorithm chooses the element according to the selection criterion

$$i^* = \arg \min_{i \in \Delta_A^+(x)} \varphi_A(i,x)$$

necessarily $\varphi_A(i,x)$ is incorrect, but probably not completely wrong. The semigreedy algorithm proposed by Hart and Shogan in 1987 assumes that elements that lead to the optimum are very good for $\varphi_a(i,x)$ even if not strictly the best. How to know which one? The semigreedy algorithm is based on the definition of a suitable distribution function on the elements of $\Delta_A^+(x)$, favouring the elements with the best values of $\varphi_A(i,x)$ and where elements are selected by the outcome of a random extraction $i^*(\omega)$. Formally, the idea is to assign a probability $\pi_A(i,x)$ to each arc $(x, x \cup \{i\})$ of the construction graph so that

$$\sum_{i \in \Delta_A^+(x)} \pi_A(i,x) = 1 \ \ \forall x \in F_A : \Delta_A^+(x) \neq \emptyset$$

remembering that on each arc is defined $\varphi_A(i,x)$ and that for how we defined the semigreedy technique

$$\varphi_A(i,x) \leq \varphi_A(j,x) \iff \pi_A(i,x) \geq \pi_A(j,x)$$

(for minimization problems), for each $i, j \in \Delta_A^+(x), x \in F_A$.

This heuristic approach has important properties: it can reach an optimal solution if there is a path from $\emptyset$ to X^*, which is a basic condition, and it can be reapplied several times obtaining different solutions: for each re-run the probability to reach the optimum grows gradually, because at each iteration the probability of *not* reaching the optimum, that is taking always the wrong path, will decrease.

Convergence to the optimum

The probability of following a path γ is the product of the probabilities on the arcs, so

$$\prod_{(y, y \cup \{i\}) \in \gamma} \pi_A(i,y)$$

in words given a path γ going to a final solution, one of the maximal paths of the construction graphs, that is it represent a run of the algorithm, the probability of having that run exactly is given by the product of the probabilities of the arcs in such a path, due law of conditional probabilities. The probability of reaching a specific node x is the sum of the probabilities of the paths Γ_x leading to x

$$\sum_{\gamma \in \Gamma_x} \prod_{(y, y \cup \{i\}) \in \gamma} \pi_A(i,y)$$

as they are incompatible events. If and only if a path of non-zero probability from the empty set to the optimal solution set exists, there is a non-zero probability of reaching a solution x in the optimal subset. The probability of reaching x grows as the number of iterations $\ell \to +\infty$ grows up to infinity. If one can prove that this number tends to 1, then the heuristic will be probabilistically approximatively complete: sometimes, algorithm have probabilities that aren't fixed and equal in each iteration of the algorithm, so it's not general to assume that the number tends to 1.

Stochastic construction graph

A first example of an algorithm that uses a stochastic construction graph is the **random walk**, which in the context of heuristics is a constructive metaheuristic in which the probability is the same for all the arcs going out of the same node: it finds a path to the optimum with probability 1, if it exists, and the time required can be extremely long, which may be longer than what it takes to the exhaustive algorithm. A deterministic constructive heuristic sets all probabilities to zero except for those on the arcs of a single path, that is the path composed by nodes that satisfy its selection criterion. So, the idea is that in general one is in a sort of midway through this two options: no very focused probabilities on a single arc and no equiprobable, uniform probabilities. What is to be done is trying to focusing the probability on the most promising arcs and reducing it on the other ones, accelerating the convergence time, but it doesn't guarantee the convergence to the optimum. Arcs with zero probability can bar the way to the optimum; arcs with probability converging to zero reduce the probability of finding the optimum.

7.2.3 GRASP

The **Greedy Randomized Adaptive Search Procedure**, proposed by Feo and Resende in 1989, is a sophisticated variant of the semi-greedy heuristic and is one of the most popular heuristics in the literature. It is greedy because it uses a greedy basic constructive heuristic; it is randomized because it takes random steps; it is adaptive because the heuristic uses an adaptive selection criterion $\varphi_A(i,x)$ (but it's not strictly necessary) and it is a search because it alternates the constructive heuristic and an exchange heuristic. The use of auxiliary exchange heuristics allows strongly better results: this aspect will be investigated later.

The probability function used in GRASP has to be defined so that better values of the selection criterion correspond to larger probabilities of being chosen

$$\varphi_A(i,x) \leq \varphi_A(j,x) \iff \pi_A(i,x) \geq \pi_A(j,x)$$

and one of the possible ways to do this, that is common even to genetic algorithms, is to relate directly π and φ as a strictly decreasing function; this is not what is done in GRASP, because experiments on genetic algorithms, explored later, proved this method to be rather biased and uneffective. Building on the results of those experiments, other schemes were used, in particular *ranking schemes*, which are adopted in GRASP and, in general, in semigreedy techniques.

Ranking Schemes

Consistently with the relation between selection criterion and probability, the choice of uniform probability is valid (as for the random walk) even though it's not usally chosen. Popular ranking schemes are drawn from the category of **Heuristic Based Stochastic Sampling** (HBSS) and **Restricted Candidate List** (RCL). Both are based on ranking, that is the arc in $\Delta_A^+(x)$ are sorted by nonincreasing values of $\varphi_A(i,x)$ and assigning probabilities according to the position (or the rank) of each arc - the probability is not given in function of *how big* the best arc's value is, but on a ordinal value; this allows to control much better the values of the probabilities that will be used and not just let them be dependent on the value of φ. The first scheme is created by assigning a decreasing probability according to the ordinal values based on a simple scheme (linear, exponential, etcetera); the second scheme is created by assigning a uniform probability to the first n chosen arcs in the list and zero to the remaining ones.

The most common strategy is the RCL, even if the zero probability arcs potentially cancel the global convergence to the optimum.

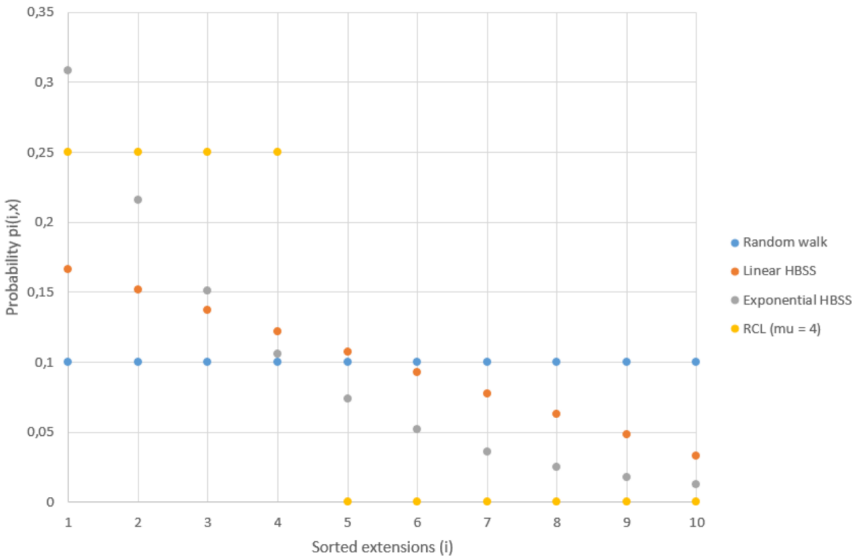


Figure 7.1: Ranking Schemes at a step where $|\Delta_A^+(x)| = 10$ extensions are available

Definition of the RCL

Two main strategies are used to define the RCL: the first is based on **cardinality**, where the RCL includes the best μ elements of $\Delta_A^+(x)$ where $\mu \in \{1, \cdots, |\Delta_A^+(x)|\}$ is a parameter fixed by the user and setting $\mu = 1$ yields the constructive basic heuristic and $\mu = |B|$ yields the random walk; the second strategy is based on **value** and the RCL includes all the elements of $\Delta_A^+(x)$ whose value is between φ_{min} and $(1-\mu)\varphi_m in + \mu\varphi_{max}$ where, obviously,

$$\varphi_{min}(x) = \min_{i \in \Delta_A^+(x)} (i,x) \text{ and } \varphi_{max}(x) = \max_{i \in \Delta_A^+(x)} (i,x)$$

$\mu \in [0,1]$ is a parameter fixed by the user and setting $\mu = 0$ yields the basic constructive heuristic and $\mu = 1$ yields the random walk. Clearly, this second definition is adaptive as it's based on the best and worst choice for *each* choice, which can vary, meaning that the number of considered options is not constant.

In practice, the values aren't sorted as it would add unnecessary complexity to the algorithm as it's rather clear that when a random number is extracted it is already known, in ordinal sense, which extension is to be selected (e.g. the third) and finding the n-th element is less complex than sorting.

Grasp for the SCP Following the Table 7.1, an example of the GRASP for the SCP using an RCL with $\mu = 2$ candidates, an execution could be

1. start with the empty subset $x^{(0)} = \emptyset$
2. build a RCL with 2 candidates: column 2, which has $\varphi_2 = 2$ amd column 3, which has $\varphi_3 = 4$; select (at random) column 3
3. build a RCL with 2 candidates: column 2 and column 1, which has $\varphi_1 = 6.25$; select (at random) column 1
4. the solution obtained is $x = \{3,1\}$ of cost $f(x) = 33$

With $\mu = 2$ the optimal solution cannot be obtained, but with $\mu = 3$ it can. Actually, appending a destructive post-processing phase to the algorithm the optimal solution is reachable even with $\mu = 2$.

Algorithm 9 GRASP Pseudocode

```
1: procedure GRASP(I)
2:     x* := ∅
3:     f* := +∞
4:     for l = 1 to ℓ do
5:         while Δ_A^+(x) ≠ ∅ do
6:             φ_i := φ_A(i,x) i ∈ Δ_A^+(x)
7:             L := Sort(Δ_A^+(x), φ)
8:             π := AssignProbabilities(L, μ)
9:             i := RandomExtract(L, π)
10:            x := x ∪ {i}
11:        end while
12:        x := Search(x)
13:        if x ∈ X and f(x) < f* then
14:            x* := x
15:            f* := f(x)
16:        end if
17:    end for
18:    return (x*, f*)
19: end procedure
```

Reactive semigreedy algorithm

A last point on semigreedy or GRASP is about tuning parameters. Of course, the number of iterations has to be tuned based on the available time and the parameter μ has to be tuned to determine the size of the RCL. An idea to tune μ is to exploit memory learning from the previous results, using a **reactive** method:

1. select m configurations of parameters $\mu_1, \cdots, \mu_2$ and set $\ell_r = \ell/m$
2. run each configuration μ_r for ℓ_r iterations
3. evaluate the mean $\bar{f}(\mu_r)$ of the results obtained with μ_r
4. update the number of iterations ℓ_r for each μ_r based on $\bar{f}(\mu_r)$

$$\ell_r = \frac{\frac{1}{\bar{f}(\mu_r)}}{\sum_{s=1}^{m} \frac{1}{\bar{f}(\mu_s)}} \ell \text{ for } r = 1, \cdots, m$$

5. repeat the whole process, going back to point (2), for R times

So, the idea is to give the same importance to different values of μ at first, then updating the importance, measured in terms of conceded iterations to each configuration, to values μ_i that yield the best mean objective function values.

7.2.4 Cost Perturbation methods and Ant Colony optimization

The last family of metaheuristic is the one of the **cost perturbation methods** in which the idea is that instead of forbidding some of the choices or modifying the probabilities in random

choices, we can try and modify the *appeal* of the available choices - that is, we can try and modify the selection criterion. Given a basic constructive heuristic A, at each step of iteration l tune the selection criterion $\varphi_A(i,x)$ with a factor $\tau_A^{[l]}(i,x)$ so that

$$\psi_A^{[l]}(i,x) = \frac{\varphi_A(i,x)}{\tau_A^{[l]}(i,x)}$$

and update $\tau_A^{[l]}(i,x)$ based on the previous solutions $x^{[1]},\cdots,x^{[l-1]}$. The elements with a better $\varphi_A(i,x)$ tend to be favoured, but $\tau_A^{[l]}(i,x)$ tunes this effect promoting **intensification** if it increases for the most frequent elements so as to obtain solutions similar to the previous ones or **diversification** if it decreases for the most frequent elements so as to obtain solutions different to the previous ones. This may be done on the basis of the history of previous solutions: if it's thought that the most frequent solutions aren't that good, enforcing diversification might be a good idea; on the contrary, if the most frequent solutions are considered to be promising, intensification may be a good choice.

The most famous and important subfamily of such methods is the **Ant Colony Optimization**, which combines cost perturbation methods with randomization in a very specific way. This method was proposed by Dorigo, Maniezzo and Colorni in 1991, drawing inspiration from the social behaviour of ants. The ants go from food to the nest at random and when an ant finds

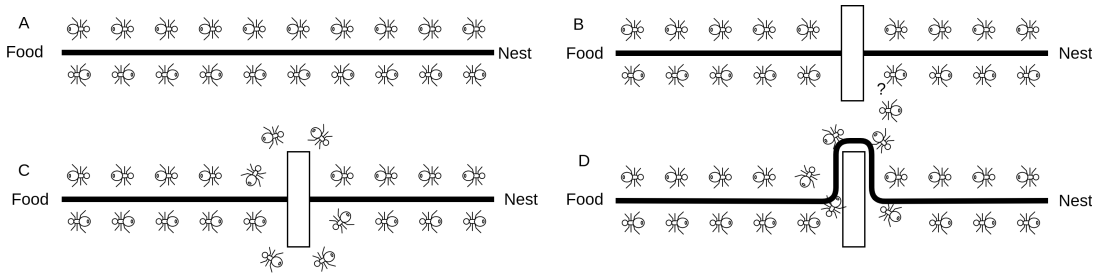


Figure 7.2: Ant behaviour.

food it goes back to the nest leavin some chemical trail on the ground and other ants follow the trail and tend to follow it again and again. Suppose that there are different paths to the food: being longer, less ants will get there and the *trail* will be weaker (in chemical terms); only the *best* path will be selected. The natural analogy is not important, and in fact it's important to get rid of it as soon as possible, keeping only the element important for optimization.

Anyway, the basic points are leaving trail, associated to the data and making choices based on the trails left on the data.

Trail

The **trail** is τ for the general cost perturbation scheme. Given the basic constructive heuristic A, each step performs a partially random choice. Differently from the semigreedy heuristic, each iteration l runs f times the heuristic A, building a **population**; all the choices of $\Delta_A^+(x)$ can be chosen (there is no RCL) and the probability $\pi_A(i,x)$ depends on the selection criterion $\varphi_A(i,x)$ and the auxiliary information $\tau_A(i,x)$, denoted as **trail** produces in previous iterations and sometimes by other agents in the same iteration.

The trail is uniformt at first ($\tau_A(i,x) = \tau_0$) and later tuned increasing or decreasing it, in order to favour promising choices or to avoid repetitive choices, respectively. For the sake of simplicity, the trail $\tau_A(i,x)$ is not associated to each arc $(x, x\cup\{i\})$, but is the same for blocks of

arcs, for example depending only on i, because the construction graph is never represented as a data structure, because it would be humongous. Typically, trails are "saved" on the elements of the ground set, meaning indirectly that the on all the arcs of the construction graph there is the same trail on all the arcs that correspond to adding the same element i to any possible x, as τ depends on i and not on x.

Random choice

The idea is that instead of selecting the best element according to the criterion $\varphi_A(i,x)$, i is extracted from $\Delta_A^+(x)$ with probability

$$\pi_A(i,x) = \frac{\tau_A(i,x)^{\mu_\tau}\eta_A(i,x)^{\mu_\eta}}{\sum_{j\in\Delta_A^+(x)}\tau_A(j,x)^{\mu_\tau}\eta_A(i,x)^{\mu_\eta}}$$

so, differently from the ranking scheme used in GRASP, the probability is given by the values of the trail τ and by a value η defined as the **visibility**

$$\eta_A(i,x) = \begin{cases} \varphi_A(i,x) \text{ for maximisation problems} \\ \frac{1}{\varphi_A(i,x)} \text{ for minimisation problems} \end{cases}$$

so that the promising choices have larger visibility. The expression of the probability is normalized by the denominator; the two parameters μ_τ and μ_η tune the weight of the two terms in the numerator: if $\mu_\eta \approx 0$ and $\mu_\tau \approx 0$ the probability distribution becomes uniform (a random walk). Setting $\mu_\eta >> \mu_\tau$ favours the data, simulting the basic constructive heuristic which makes sense when the known solutions are not very significant, meaning in a sense that not much was learnt from the previous iterations. On the contrary, setting $\mu_\eta << \mu_\tau$ favours memory, keeping close to the previous solutions which makes sense when the known solutions are very significant.

Balancing given and learned information

There is a plethora of variants of Ant Colony; in particular, the **Ant Colony System** simplifies the scheme setting $\mu_\eta = \mu_\tau = 1$

$$\pi_A(i,x) = \frac{\tau_A(i,x)\eta_A(i,x)}{\sum_{j\in\Delta_A^+(x)}\tau_A(j,x)\eta_A(i,x)}$$

and the difference between "data" and "memory" is configured using another mechanism, that is before deciding whether to use the previous expression with probability q (so, randomness on randomness) or optimizing $\varphi_A(i,x)$ (basic constructive heuristic) with probability $1-q$. Setting $q \approx 1$ favours data, while setting $q \approx 0$ favours memory.

Trail update

In order to update the trail, at each iteration ℓ h instances of the basic heuristic A are run and a subset $\tilde{X}^{[l]}$ of the solutions obtained is selected in order to favour their elements in the following iterations; then, the trail is updated according to the formula

$$\tau_A(i,x) := (1-\rho)\tau_A(i,x) + \rho \sum_{y\in\tilde{X}^{[l]}:i\in y} F_A(y)$$

where $rho \in [0,1]$ is an **oblivion parameter** used to reduce the current trail in order to leave space to the new contributions and $F_A(y)$ is a **fitness function** expressing the quality of the solution y such that $F > \tau$. The purpose of the update is to increase the trail on elements of the specific solution ($y \in \tilde{X}^{[l]}$) and decrease the trail on other elements.

The role of the oblivion parameter in the update The role of the oblivion parameter is to enhance diversification if $\rho \approx 1$, so that it cancels the current trail and keeping only the new one; this is based on the intuition that the solution obtained are not trustworthy and different solution should be explored. On the contrary, if a low oblivion is used ($\rho \approx 0$), intensification is enhanced as it preserves the current trail and completely deletes the new one, based on the intuition that the solutions obtained are trustworthy and similar solutions should be explored.

The solutions to be collected in $\tilde{X}^{[l]}$ must be inserted using a precise criterion as well: in the classical Ant System all oslutions of iteration $l-1$ were inserted, but recently more elitist methods, in which an elite of solutions is sought, which usually is reduced to a single solution, often the best solution or the best solution overall. The elitist methods find better results in shorter time but require additional mechamisms to avoid **premature convergence**, then getting stuck in the solutions previously found.

Some variants of the Ant System

- $MAX-MIN$ Ant System, which imposes on the trail a limited range of values $[\tau_{min}, \tau_{max}]$ and is experimentally tuned;
- HyperCube Ant Colony Optimization (HC-ACO) which normalizes the trail between 0 and 1;
- Ant Colony System: updates the trail on two levels:
  - the global update - already seen - modifies it at each iteration ℓ and its purpose is to intensify the search;
  - the local update modifies the trail at each application g of the basic heuristic in order to discourage identical choices in the following

    $$\tau_A(i,x) := (1-\rho)\tau_A(i,x) \quad \forall i \in x_A^{[l,g]}$$

    whose purpose is to diversify the search.

  so, overall, the idea is that each "ant", each application, reduces the trail and decreases the trail on all elements that belong to the solution found by the current application g in the current iteration l.

Convergence to the optimum

Some variants of the Ant System converge to the optimum with probability 1. The analysis is based on the construction graph, and the idea is that the trail $\tau_A(i,x)$ is a weight function defined on the arcs $(x, x \cup \{i\})$ of the construction graph. The proof doesn't assume any information from the data, meaning that the visibility is uniformly fixed to 1, that is $\eta_A(i,x) = 1$ - in other words, the basic heuristic only looks and the trail and doesn't look at the data of the problem; a note on the paper says that this restriction can be extended to consider also the data.

The idea is that at each iteration the trail at the beginning is $\tau^{[l]}$, before running the basic heuristic(s); $\gamma^{[l]}$ is the best known path on the construction graph from the empty set to a final solution at the end of an iteration: at the beginning of an iteration the couple $(\tau^{[l]}, \gamma^{[l-1]})$ is given and it is the state of a **nonhomogeneous Markov process** where the probability of each state depends only on the previous iteration, which is nonhomogeneous because the dependency on the previous iteration varies with l.

Algorithm 10 Ant Colony System Pseudocode

```
1:  procedure ANTSYSTEM(I)
2:      x* := ∅
3:      f* := +∞
4:      for l = 1 to ℓ do
5:          for g = 1 to h do
6:              x := A(l, τ_A)
7:              x := Search(x)                          ▷ Improvement heuristic
8:              if f(x) < f* then
9:                  x* := x
10:                 f* := f(x)
11:             end if
12:             τ_A := LocalUpdate(τ_A, x)
13:         end for
14:         X̃^[l] := Update(X̃^[l], x)
15:         τ_A := GlobalUpdate(τ_A, X̃^[l])
16:     end for
17:     return (x*, f*)
18: end procedure
```

The proof concludes that for $l \to +\infty$ with probability 1 at least one run follows an optimum path in F and the trail τ tends to a maximum along one of the optimal paths and to zero on other arcs.

The main point in the proof is that the trail is being updated with a variable coefficient of oblivion ρ:

$$\tau^{[l]}(i,x) := \begin{cases} (1-\rho^{[l-1]})\tau^{[l-1]}(i,x) + \rho^{[l-1]}\frac{1}{|\gamma^{[l-1]}|} & \text{if } (x, x\cup\{i\}) \in \gamma^{[l-1]} \\ (1-\rho^{[l-1]})\tau^{[l-1]}(i,x) & \text{otherwise} \end{cases}$$

where $\gamma^{[l-1]}$ is the bast path found in the graph up to iteration $l-1$ and $|\gamma^{[l-1]}$ is the number of its arc, used to normalise the trail. If the oblivion decreases slowly enough

$$p^{[l]} \geq 1 - \frac{log(l)}{log(l+1)} \text{ and } \sum_{l=0}^{+\infty} \rho^{[l]} = +\infty$$

where the sum diverges so to forget the trail on the wrong path sufficiently fast and ρ decreases sufficiently slow so not to get stuck on a path with premature convergence, then with probability 1 the state converges to (τ^*, γ^*) where γ^* is an optimal path in the construction graph and $\tau^*(i,x) = (|\gamma^*|)^{-1}$ for arcs $(x, x\cup\{i\}) \in \gamma^*$ and 0 otherwise.

The paper shows that there's a second variant with global convergence where the oblivion ρ remains constant but the trail is slowly forced to decrease to a minimum threshold

$$\tau(i,x) \geq \frac{c_l}{log(l+1)} \text{ and } \lim_{l\to+\infty} c_{[l]} \in (0,1)$$

then, with probability 1, the state converges to (τ^*, γ^*); the intuition is that the trail must *decrease fast but not too fast*. In practice, all algorithms proposed so far in the literature associate the trail to groups of arcs $(x, x\cup\{i\})$ (e.g. to a single element i) and/or use constant values for parameters ρ and τ_{min}, therefore do not guarantee convergence. Nonetheless, it's rare that the objective of a constructive metaheuristic is to let it converge and find the optimal solution in each application.

Part IV

Solution Based Heuristics: Exchange Heuristics

CHAPTER 8

Exchange Heuristics

Exchange algorithms are based on the concept of *manipulation* of the solutions, just as it was for constructive heuristics. It's important to remember that in Combinatorial Optimization problem every solution x is a subset of the ground set B. An exchange heuristic updates a current subset $x^{(t)}$ step by step, exchanging elements in and out of the solutions: some elements taken from $B \setminus x$ are inserted in the solution and some other elements in the solution are removed.

8.1 Introduction to Exchange Heuristics

The general scheme of exchange heuristic is

1. start from a feasible solution $x^0 \in X$ found somehow, often by a constructive heuristic
2. generate a family of feasible solutions by exchanging elements, for example add subsets A external to $x^{(t)}$ and drop 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 criterion $\varphi(x, A, D)$ to choose the subsets to exchange

$$(A^*, D^*) = \arg\min_{(A,D)} \varphi(x, A, D)$$

4. perform the chosen exchange to generate the new current solution

$$x^{(t+1)} := x^{(t)} \cup A^* \setminus D^*$$

5. if a termination condition holds, terminate; otherwise, go back to point (2).

As can be observed, the execution of an exchange heuristic requires a preexistent feasible solution: this is often achieved by using a constructive heuristic which takes a small time to find it; often, the final result doesn't depend very much on the initialization - anyway, this means that the starting point is not given a priori. Note that an extension of constructive heuristics contemplated a similar update of the solution using addition and removal sets: in this case, the net update is not bounded to be strictly an increment on the size of the solution. The graph that results from this mechanism won't be necessarily an acyclic graph and, in general, the exension to the solution (the "net" update) is a feasible solution as well, meaning that the selection criterion keeps track of the feasibility of the solutions. At the end of the update, if the solution is "good", meaning that some specific termination conditions hold, the algorithm

terminates, otherwise the cycle of update starts again using this new solution - called *incumbent* solution - as a starting point. It's clear that exactly as it was for constructive algorithms the possible modifications (the extensions), the selection criterion and the termination condition must be selected.

8.1.1 Neighbourhood

An exchange heuristic is defined by the pairs of exchangable subsets (A,D) in every solution x, that is the solutions generated by a single exchange starting from x and the selection criterion $\varphi(x,A,D)$. This is exactly like saying that for each current solution x there exist subsets of other feasible soutions which can be added and removed from the current solution; the **neighbourhood**

$$N : X \to 2^X$$

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 $x \in X$ and the arcs connect each solution x to those of its neighbourhood $N(x)$, moving elements into and out of x, often denoted as *moves*; so, the triple of all $\langle x^{(i)}, A^{(i)}, D^{(i)} \rangle$, the triple of solutions, the subset added and the subset removed and the search graph represent exactly the same thing. The difference between the search graph and the construction graph is that the former admits cycles and different starting points (every feasible solution), whereas the latter must be a DAG and always starts from the same point, that is $x^{(0)} = \emptyset$. Every run of the algorithm corresponds to a path in its search graph - but how does one *define* a neighbourhood and select a move?

Neighbourhood based on distance

Several neighbourhoods can be defined on the same search graph, exactly as several construction graph could be defined. In an abstract fashion, neighbourhoods can be defined in two ways, which can be always adopted; besides this general methods, there are precise methods that take into account the structure of the problem. The first abstract method is based on the concept of **distance**; first, let's remind that every solution $x \in X$ can be represented by its incidence vector, defined as

$$x_i = \begin{cases} 1 & i \in X \\ 0 & i \in B \setminus x \end{cases}$$

Given the incidence vector, it's very easy to define the distance between two solutions x and x' represented as incidence vector using **Hamming distance**[1]

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

in relation to their subsets, the Hamming distance

$$d_H(x,x') = |x \setminus x'| + |x' \setminus x|$$

Having such a definition, a typical definition of neighbourhood is the set of all solutions with a Hamming distance from x not larger than k for some integer k:

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

this is a parametric defition.

[1] What represented in the formula is actually the Manhattan (L1) distance; however, in this case, they are exactly the same.

Example: the KP The KP instance with $B = \{1,2,3,4\}$, $v = [5432]$ and $V = 10$ has 13 feasible solutions out of 16 subsets: since subsets $\{1,2,3,4\}$, $\{1,2,3\}$ and $\{1,2,4\}$ are unfeasible.

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

Solution $x = \{1,3,4\}$, in blue, has a a neighbourhood $N_{H_2}(x)$ consisting of 7 elements in red. The subsets in black do not belong to the neighbourhood, because their Hamming distance from x is greater than 2.

Neighbourhood based on operations

A second abstract way to define a neighbourhood is based on **operations**. The idea is that a family of operations $\mathcal{O}$ is defined on the solutions of the problem; the neighbourhood is then defined as

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

An *operation* is a function applied on a solution which generates another solution by adding to x an element of $B \setminus x$, removing an element from x or exchanging 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:

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

to demonstrate this, consider that the subsets of Hamming distance 1 are incidence vectors in which only one value changes, that is one can be added or one can be removed; an exchange operation is not contained here. On the other hand, two additions or two removals aren't allowed using the operations in $\mathcal{O}$, while in the subsets of Hamming distance 2 they are. As the distance-based ones, these neighbourhoods can be parametrised considering sequnces of k operations of $\mathcal{O}$ instead of a single one

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

Difference between 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. Consider the TSP instance in Figure 8.1.

Using the definition of neighbourhood given earlier, we can define the neighbourhood of a solution $x = \{3,1,4,5,2\}$

$$\begin{aligned} N_{S_1}(x) = &\{(\mathbf{1},\mathbf{3},4,5,2),(\mathbf{4},1,\mathbf{3},5,2),(\mathbf{5},1,4,\mathbf{3},2),(\mathbf{2},1,4,5,\mathbf{3}),(3,\mathbf{4},\mathbf{1},5,2),\\ &(3,\mathbf{5},4,\mathbf{1},2),(3,\mathbf{2},4,5,\mathbf{1}),(3,1,\mathbf{5},\mathbf{4},2),(3,1,\mathbf{2},5,\mathbf{4}),(3,1,4,\mathbf{2},\mathbf{5})\} \end{aligned}$$

The interesting point is: can this neighbourhood be defined, in some way, using Hamming distance? Turns out that the answer is no: this is because swapping two nodes in a solution

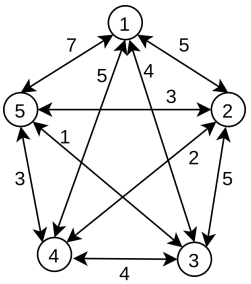


Figure 8.1: TSP Problem

that are adjacent, as in the first case, it can be noticed that of the five original arcs ($3 \to 1 \to 4 \to 5 \to 2 \to 3$) three arcs have been removed ($(3,1),(1,4),(2,3)$) and three arcs have been added ($(1,3),(3,4),(2,1)$), so the Hamming distance is $3+3=6$. Furthermore, swapping two non-adjacent as in the second solution in the neighbourhood, four arcs have been removed ($(1,4),(4,5),(2,3),(3,1)$) and, obviously, four arcs have been added, so the Hamming distance is $4+4=8$; however N_{S_1} does not coincide with neither N_{H_6} or N_{H_8}.

Relations between distance and operation-based neighbourhoods

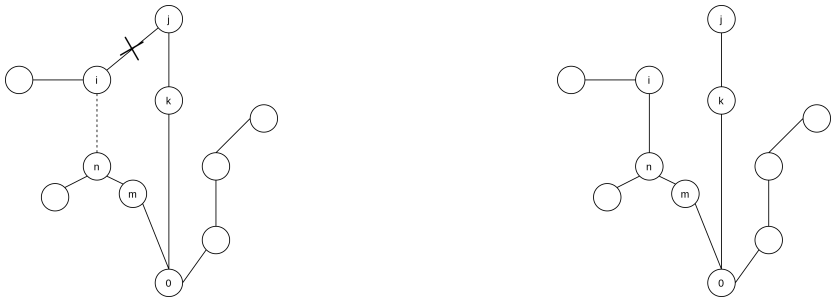
Sometimes, it can be shown that the two definitions are actually compatible: this usually happens when they concern problem with a fixed number of elements. Considering the MDP, where the number of elements in the solution must be k, it's clear that when adding j elements the same number of elements must be removed; so, considering the incidence vector of a single swap, a 1 becomes a 0 and a 0 becomes a 1, so $N_{S_1} \equiv N_{H_2}$. Something similar happens for the BPP (and the PMSP) considering the neighbourhood N_{T_1} made out of *transfers* of an object into a different container and the neighbourhood N_{H_2}, that is the solutions at Hamming distance equal to 2 defining the ground set B as the pairs $\langle object, container \rangle$. Another relation can be found in the Max-SAT problem defining the neighbourhood N_{F_1} made out of *flips* of variables by inverting their truth assignment and N_{H_1}.

This is typical but it's not a rule: sometimes operations-based neighbourhood do correspond to Hamming distance based neighbourhood and sometimes they don't.

Different neighbourhoods for the same problem

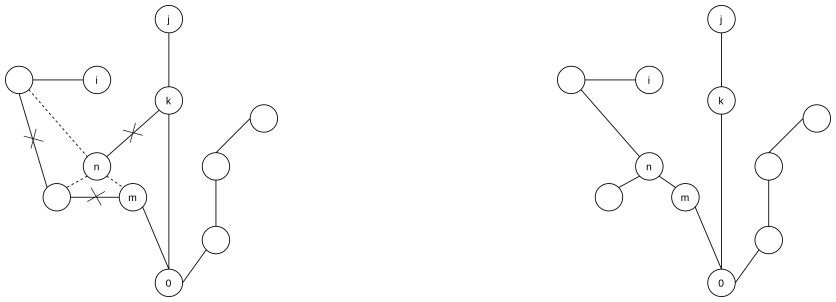
The CMST Often, different definitions of neighbourhood can be given; in particular, in the case of graph problems it is typical to describe operations made on edges and operations made on vertices (or on arcs and nodes). Let's consider the CMST. Suppose that the neighbourhood is defined with respect to edges and one can say delete edge (i,j) and add (i,n), as represented in Figure 8.2. This exchange consists in removing the said edge and adding a new edge. This corresponds to moving some vertices (in fact, the whole subtree rooted in the node i in the picture) to another subtree - but, in general, an arbitrary number of vertices can be moved with an edge update.

Conversely, defining the neighbourhood in terms of vertices, one can move the vertex n from subtree 2 to subtree 1 as represented in Figure 8.3, then recomputing the two minimum spanning subtrees and, in general, it is not known how many edges are going to be removed and how many are going to be added just moving a single vertex. This is to stress the fact that two possible ground sets were considered regarding the CMST: one based on edges and one based on assignment of vertices to subtrees. Indeed, this operation-based neighbourhood can be seen as distance based neighbourhood in a ground set corresponding to edges or vertices



(a) Before exchange (b) After exchange

Figure 8.2: Edge exchange in CMST



(a) Before exchange (b) After exchange

Figure 8.3: Vertices exchange in CMST

assignments. This is very similar to having different search spaces in constructive algorithms as different solutions will be obtained and, maybe, some are betters than others.

The PMSP Let's give another example of possible different definition of neighbourhood: for the PMSP one can define, among all, the *transfer* neighbourhood N_{T_1} based on the set T_1 of all transfers of a task on another machine, with an example represented in Figure 8.4. Previously, a transfer neighbourhood for the BPP was discussed and this one is very similar: instead of assigning a task to a certain machine it simply could be assigned to another machine. So, in a Hamming distance approach, the distance would be 2, as in a incidence vector 0 would go to 1 and a 1 would go to 0 (given, of course, that the incidence vector is based on the assignment-to-machine ground set).

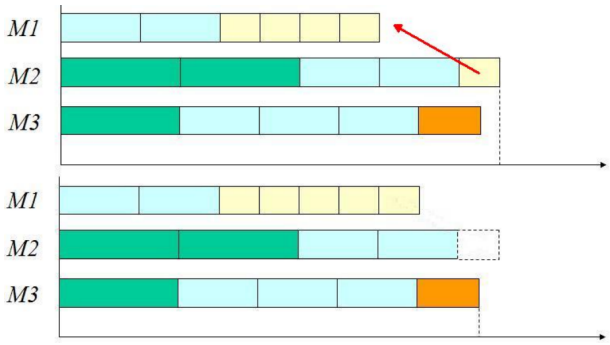


Figure 8.4: Transfer neighbourhood example

Two tasks could be moved; choosing that the two task should exchange the respective position, then a *swap*-operation based neighbourhood would be defined, so N_{S_1} is a neighbourhood

based on the set S_1 of the swaps of two tasks between two machines (one for each machine), as represented in Figure 8.5. The relationship between *swaps* and *transfers* is similar to the rela-

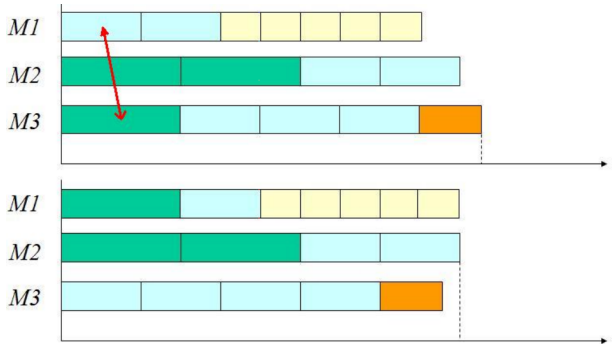


Figure 8.5: Swap neighbourhood example

tionship between Hamming distance and operations in the case of the KP discussed previously, where the operation based neighbourhood was intermediate between the neighbourhood based on hamming distance 1 and 2 and in this case a swap is a specific case of two transfer, so it is

$$N_{S_1} \subset N_{T_2}$$

8.1.2 Connectivity of the solution space

Choosing a neighbourhood is not trivial, as some properties must be satisfied. Let's start to understand which could be good properties for the neighbourhood of a problem: of course, exactly as for the construction graph, the algoritm is able to return the optimum only if there's a path to it; then for an exchange heuristic, the optimum can be reached only if every feasible solution can reach at least one optimal solution, that is there is a path from x to X^* for every $x \in X$

$$x \rightsquigarrow x^* : x^* \in X^* \quad \forall x \in X$$

this is because the general scheme of an exchange algorithm requires to start from an arbitrary feasible solution. Then, the problem of *how* to "guide" the search along that path has to be solved, but that's different. A search graph having such a property is denoted as *weakly connected* to the optimum. It's quite difficult to prove that a problem with a certain definition of neighbourhood is weakly connected, as often the (or an) optimum is unknown: often, then, a stronger condition is used, that is the condition of *strong connectivity* which is provided when a search graph admits a path from x to y for every $x, y \in X$

$$x \rightsquigarrow y \quad \forall x, y \in X$$

As done previously, one can associate to each arc a probability and the possibility to reach an optimal solution will require the existance of a path with a non-zero probability to be followed.

So, good and bad neighbourhood can be recognized based on their properties: let's give two very simple example. Consider the MDP and the swap neighbourhood N_{S_1}, there's a way to go from *any* solution to *any* solution, because two solutions can be connected with a finite number of swaps. Considering the same neighbourhood in the KP (or the SCP); in this case, the search graph is not strongly connected, as, for example a solution with j elements can't reach a solution with $i \neq j$ elements even with an arbitrary number of swaps; the search graph becomes connected also in the KP and the SCP if removals and additions are added to swaps.

Example on the connectivity of the solution space

Even if a problem with a fixed number of elements if given, such as the CMST in which the number of edges is $n-1$ in any feasible solution, there may be additional constraints that could make it difficult or impossible to move from a solution to any other. Consider the instance in Figure 8.6 with uniform unitary weights on vertices except for vertex b which has weight 2.

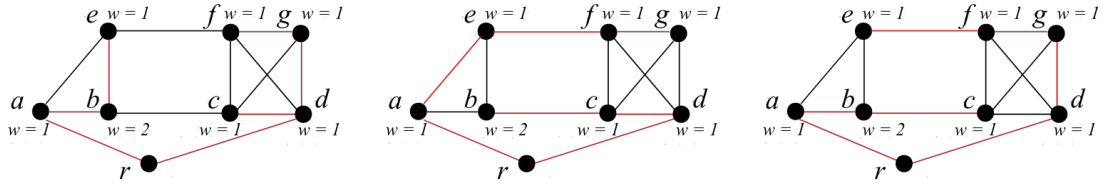


Figure 8.6: Three solutions for a CMST instance.

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

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

which are mutually reachable only exchanging at least two edges (two to remove and two to add) at a time. If the neighbourhood is a swap of a single edge, one can't move from one of these feasible solutions to any other. The lesson taught by this example is that even if a general structure given by the problem itself seems to allow connectivity, the additional complicating constraints can take away some subsets that are no longer feasible and forbid movements from a feasible solution to another one.

8.2 Structure of Exchange Heuristic algorithms

8.2.1 Moving inside the neighbourhood

Let's leave, for a moment, the concept of neighbourhood and let's consider how to move inside it - what's the selection criterion. As in the constructive heuristic, the most natural choice was to use the objective function as a selection criterion

$$\varphi(x,A,D) = f(x \cup A \setminus D)$$

The heuristic moves from $x^{(t)}$ to the best solution in $N(x^{(t)})$. In this case, one may consider the *variation* in the objective function, as done for the constructive heuristics if it allows to update the objective function and to compute it more easily. The pseudocode of the exchange heuristic is presented in Algorithm 11.

To avoid cyclic behaviour, only strictly improving solutions are accepted. This algorithm is called *steepest descent* and can be called steepest ascent or hill climbing for maximization problem. The important point in this algorithm is that since the solution changes only when $f(\tilde{x}) < f(x)$, the solution yielded is the best one, so there's no need to save the best solution.

Local and global optimality

The algorithm terminates, by definition, into a locally optimal solution, that is a solution $\bar{x} \in X$ such that

$$f(\bar{x}) \leq f(x) \quad \forall x \in N(x)$$

Algorithm 11 Exchange Heuristic Pseudocode

```
1:  procedure SteepestDescent(I)
2:      x := x^(0)
3:      Stop := false
4:      while Stop = false do
5:          x̃ := arg min_{x'∈N(x)} f(x')
6:          if f(x̃) ≥ f(x) then
7:              Stop := true
8:          else
9:              x := x̃
10:         end if
11:     end while
12:     return (x, f(x))
13: end procedure
```

A globally optimum solution is always also locally optimal, but clearly the opposite is not true

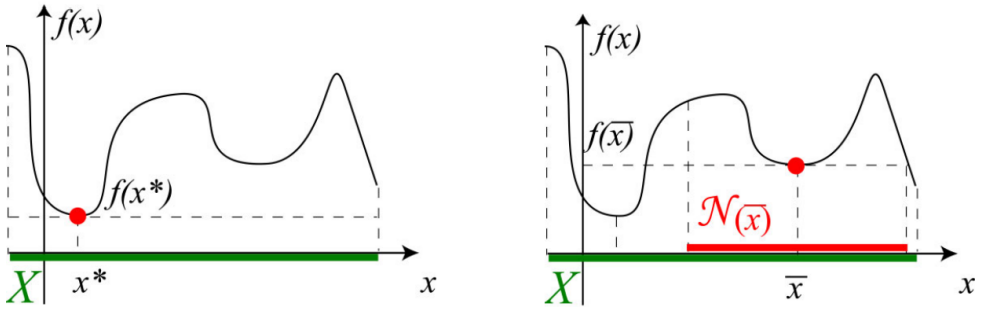


Figure 8.7: Global optimum and local optimum.

in general

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

Technically, an exchange heuristic *could* visit unfeasible solutions, but then a new whole set of problems arise. The Figure 8.7 is a continuous function, which is not the case in our analysis, but the idea is very similar.

8.2.2 Exact Neighbourhood

A neighbourhood function $N : X \to 2^X$ such that each local optimum is also a local optimum

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

is called an **exact neighbourhood** (function). The steepest descent method will "land" in $\bar{X}_N$ and consequently in the region of optimal solutions. Sadly, in general it can't be obtained: let's take a very trivial case, that is the neighbourhood of each solution coincides with the whole feasible region

$$N(x) = X \quad \forall x \in X$$

this is a useless neighbourhood, as it makes the steepest descent algorithm the exhaustive algorithm, as the selection criterion checks every feasible solution. Exact neighbourhood are extremely rare: examples of it are the neighbourhood based on edge swap for the MSTP, as any

spanning tree can be taken, any edge outside of the tree can be choosen and added to the spanning tree closing a cycle, then one edge of the cycle can be removed (it must still be a spanning tree) remaining in the set of feasible solution, obtaining another spanning tree; taking all the edges outside the solution and for each of them removing every edge that makes the solution still feasible can be done repeatedly strictly improving the solution until the optimal solution is reached - so, this is another exact algorithm besides Prim's and Kruskal's; this algorithm is less used as its complexity is stronger than those of Prim's and Kruskal's but, supposing an arbitrarily good solution is given, it can be a good algorithm. The neighbourhood based on exchange between basic and nonbasic variable used by the simplex algorithm for Linear Programming is exact as well, which can be seen as a combinatorial optimization problem whithout an additive objective function; the simplex algorithm consists in swapping basic variables with the non-basic ones and updating the objective function in such a way that the best possible swap is operated: step by step, the algorithm terminates in a local optimum which is actually a global optimum.

8.2.3 Properties of the search graph

In general, steepest descent does not give a global optimum: its effectiveness depends on the properties of the search graph and of the objective function. Some relevant properties regarding the effectiveness of an exchange heuristic are:

- the size of the search space $|X|$
- the connectivity of the search graph
- the diameter of the search graph, that is the number of arcs of the minimum path between the two farthest solutions: usually, a richer neighbourhood produces graph of smaller diameter, but other factors such as the *smallworld* effect may affect this property, where some graph is sparse, but every node can be reached with a small number of steps.

Consider, for example, the 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 and the search graph is strongly connected and has diameter less or equal to $n-2$: for example, $x = (1,5,4,2,3)$ becomes $x' = (1,2,3,4,5)$ in 3 steps:

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

as the first node is always 1 and the last one is automatically in place.

Other relevant properties, more complicated ones, are

- the density of global optima $|X^*|/|X|$ and local optima $|\bar{X}_N|/|X|$: if the local optima are numerous, it's hard to find the global ones;
- the distribution of the quality $\delta(\bar{x})$ of local optima: if local optima are good enough, it's 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's not necessary to explore the whole space. Once a local optima is reached, if the other ones are close, intensification in the search may be good; otherwise, if the density is low, diversification of the search may be helpful.

These indices would require an exhaustive exploration of the search graph to be computed, so in practice one performs a sampling (which can be a problem in and of itself) and this analyses require a very long time and can be misleading, especially if the global optima are unknown.

Example: the TSP

Typical results for the TSP on a complete symmetric graph with Euclidean costs are that the Hamming distance between two local optima is on average way smaller than n, that is the local optima concentrate in a small region of X - this means that once a local optimum is found the search should intensify. the Hamming distance between local optima on average exceeds that between local and global optima, as the global optima tend to concentrate in the middle of local optima. The Fitness-Distance Correlation diagram 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, as shown in Figure 8.8; in other words, for each local optimum of an instance the horizontal axis is its distance to the global optimum and the vertical axis is its percentage deviation from the optimum. There's a rather strong correlation between the two numbers,

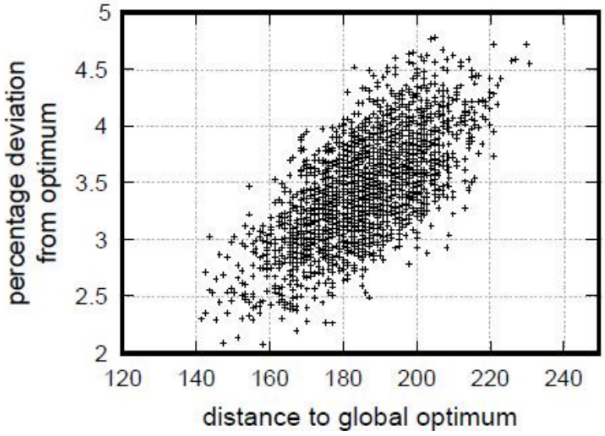


Figure 8.8: Fitness-Distance Correlation diagram for the TSP.

so moving towards local optima closer to the global optimum the percentage deviation from the optimum decreases as well (of course, the countrary holds as well). So, applying a steepest descent heuristic and obtaining a local optimum can be coupled with an action of intensification: altought it's possible to "move right" with respect to the position of the local optimum in the diagram, the selection criterion of strictly better solutions makes it probably useful to intensify the search.

Example: the QAP

Consider the Linear Assignment Problem (LAP), where some tasks and machines are given and each machine is given a single task to which a cost to the couple $\langle machine, task \rangle$ is associated (e.g. a machine can perform the task better or worse than others); the objective is to minimize the overall cost, so the objective function is additive. In the Quadratic Assignment Problem (QAP) the cost depends on *interferences* between two different assignments, so other than the linear cost, another cost depends on the fact that a task t_1 is assigned to machine m_1 and task t_2 to machine m_2 - so, it's a quite complicated set of costs to consider. In this problem the Fitness-Distance Correlation diagram, represented in Figure 8.9, which manifests that even if the solution is improved in term of deviation from the best quality (in fitness), the solution doesn't necessarily improve in distance to the global optimum: in this case, the search shouldn't be intensified but diversified.

Based on the diagram, one could argue wheter it's important to have a good solution to start with the exchange algorithm or if it's not so important; if it is known that a good solution is

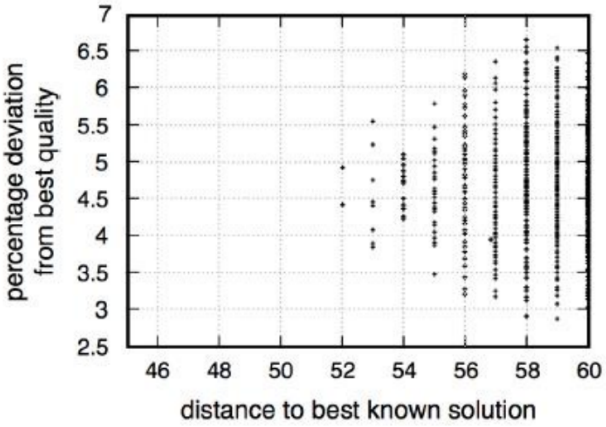


Figure 8.9: Fitness-Distance Correlation diagram fo r the QAP.

Figure 8.10: Two different landscapes.

close to the optimum, maybe it's better to spend some more time finding a good solution to initialize the problem and if, on the contrary, it's known that there's no correlation, it's better to spend less time on the initialization.

8.2.4 Descriptive Tools for Exchange Heuristic

Let's turn to some more descriptive tools to describe exchange heuristics other than te FDC.

Landscape

The **landscape** is a triplet (X, N, f) where X is the search space or the set of feasible solution, $N : X \to 2^X$ is the neighbourhood function and $f : X \to \mathbb{N}$ is the objective function - in other words, the first two give the nodes of the search space and the arcs of the search space, respectively; the third gives the weight of each arc. The effectiveness of an exchange heuristic strongly depends on its landscape: take, for example, the two landscapes pictured in Figure 8.10, where an increase in height means an increment of the cost and a decrease in height represents a decrease in cost. What is to be done is to try and get to the *lowest* nodes in the landscape: if the landscape is "smooth" and has few local optima, than the steepest descent will lead the solution to be very good. If the landscape is very rugged, it can be very hard to get to a good local optima.

There can be very different type of landscape, such as those represented in Figure 8.11; the first in the upper left corner is an exact neighbourhood, the one at its right has a good global optimum and very bad local optima, so steepest descent will perform poorly; in the lower left the landscape has pretty much equal local optima, even if they're far from the global one and the last, at the lower right, is a combination of all the previous situations, on which the steepest descent will perform very badly.

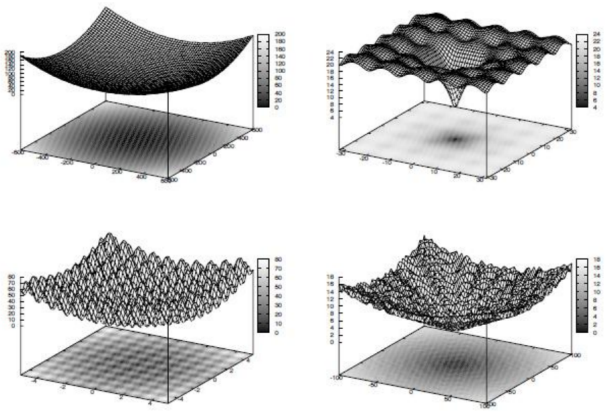


Figure 8.11: Different 3D landscapes.

Autocorrelation coefficient

The idea of the **autocorrelation coefficient** is: how can we describe any of the situation pictured in Figure 8.11? The complexity of a landscape can be empirically estimated performing a random walk on the search graph, determining the sequence of values of the objective $f^{(1)}, \cdots, f^{(t_{max})}$, computing the sample mean

$$\bar{f} = \frac{\sum_{t=1}^{t_{max}} f^{(t)}}{t_{max}}$$

and then, finally, computing the empirical autocorrelation coefficient

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

that relates the difference of the objective values in the solutions visited with the distance between these solutions along the walk. The idea is that setting $i = 0$ yields $r(0) = 1$, so that perfect correlation is achieved at 0 distance. So, setting $i = 1$ what's been measured is, given a move from one step to the following, towards values that are smaller than the mean, larger than the mean or a sequence of values either larger or smaller? In other words, are $(f^{(t)} - \bar{f})(f^{(t+1)} - \bar{f})$ both positive, both negative or they have opposite signs? Overall, in general $r(i)$ decreases as the distance i increases and if $r(i) \approx 1$ in a large range of distance the landscape is smooth: the neighbour solutions have values close to the current one, there are few local optima and the steepest descent heuristic is effective; on the other hand if $r(i)$ varies steeply the landscape is rugged: the neighbour solutions have values farm from the current one, there are many local optima and the steepest descent heuristic is ineffective. So the idea is checking whether the range of values of i in which a certain autocorrelation is strong is large or not.

Plateau

A **plateau** is a set of connected solutions that have exactly the same value for the objective function. Take, for example, the instance of PSMP represented in Figure 8.12, moving one of the tasks from machine $M1$ to machine $M3$ the objective function is not going to change: there are four different neighbourhood solutions with the neighbourhood defined by the transfers of a single task that have exactly the same objective function. The same happens transferring any of the two tasks on machine $M3$ to machine $M2$, so there's a huge number of solutions with

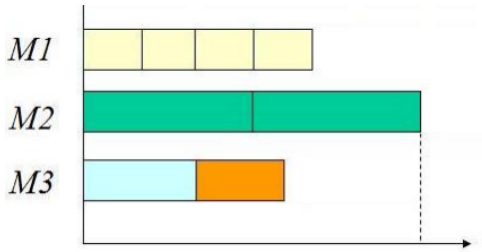


Figure 8.12: Instance of PMSP with a plateau.

the same objective function value. In this case, the selection criterion given by the objective function is not a good selection criterion: this complicates very much the problem. So, having large plateaus, on the contrary of the situation with smooth landscapes, having a *too smooth* layout can be a very big problem.

Attraction Basins

Starting from any feasible solution and following the steepest descent leads to a locally optimal solution. The set of all starting solution that end up in the same locally optimal solution is known as the **attraction basin**. The attraction basins are separated by fronteers, which have

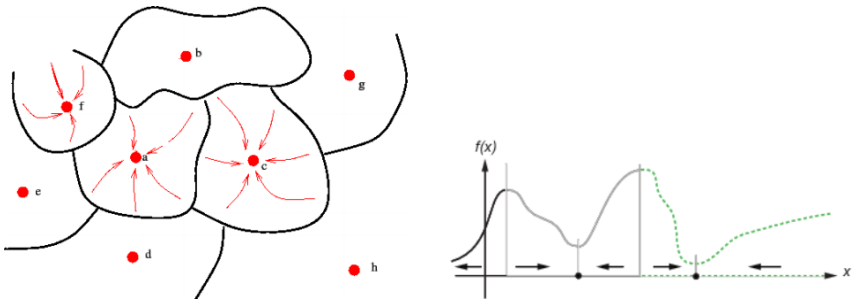


Figure 8.13: Attraction basins.

on one side a solution that goes “down” on a local optimum and on the other side, immediatly adjacent, they have a different solution that goes down on a different local optimum. The idea is to try and understand the number of attraction basins: the steepest descent heuristic is effective if the attraction basins are few and large and really ineffective if the attraction basins are many and small.

8.3 Complexity of Exchange Heuristics algorithms

Let’s consider the complexity of exchange algorithms and, in particular, how it’s possible to improve its efficiency, sometimes in a dramatic way.

The steepest descent is composed by a cycle: the problem is that the number of iterations is not fixed, espressed in terms of the size of the problem or some other parameter, but depends on the number of iterations t_{max} from $x^{(0)}$ to the local optimum found, which depends on the structure of each search graph and in particular the width of the attraction basins: this number is really hard to estimate a priori, as it also depends on the starting point $x^{(0)}$; there’s actually an active research field dedicated to the complexity of exchange heuristic or local search algorithms, as they are very commonly called. Then, the single steps can be characterized and,

done this, one can measure a posteriori, empirically, the complexity of the overall algorithm in order to combine the two informations and have some indirect information on the number of iterations that the algorithm does. For example, one can theoretically prove that each iteration is quadratic and then measure that the overall algorithm is cubic, hence suggesting that the number of iterations is probably linear; in other situations it may suggest that the number of iteration is quadratic, cubic... Then, it's important to characterize the complexity of the search for the best solution, exactly as it's done for many other typical algorithm, so by looking at how the minimization is performed, making a theoretical or practical estimation.

8.3.1 Exploration of the Neighbourhood

The exploration of the neighbourhood, line (5) in Algorithm 11, is usually done in two possible ways: the first one is *exhaustive search*, that is the most natural and simple way to do it, generating one by one all the neighbour solutions computing their objective function, searching for the best one. The complexity of this way of exploring the neighbourhood depends on the cardinality of the neighbourhood itself, so it's $\propto |N(x)|$ and on the cost of the evaluation of each solution, expressed as $\gamma_f(|B|,x)$, which in general depends on the size of the problem and, possibly, on the size of the solution. An example is the cost for computing the value of a solution for the MDP, which depends on k^2 evaluating it from scratch. However, the problem is that sometimes it's not possible to generate only neighbouring solutions starting from x, it depends on how the neighbourhood is defined (Hamming distance, operations..) and it's not always guaranteed that the generated solutions are actually feasible, so may be necessary to do is generating a superset of the neighbourhood made out of candidate solutions and for each of them evaluate its feasibility. An example of this is the KP, given the neighbourhood based on swaps: not all solutions generated from swaps are feasible, as sometimes the increase in the volume of elements in the solution may be above the capacity of the knapsack. It's not easy to understand this a priori, so what's often done is post-generation filtering after generating a superset of the feasible neighbourhood $N(x) \subset N'(x)$ by evaluating the feasibility with the function $\gamma_x(|B|,x)$ and finally, only for a feasible subset, evaluating the cost of each solution with $\gamma_f(|B|,x)$.

For some specific neighbourhood it's also possible to try and solve the auxiliary minimization problem as a combinatorial optimization problem, for which an efficient algorithm may exist. This is, of course, very nice and desirable, but requires the neighbourhood to have a specific structure.

Exhaustive visit of the neighbourhood

For the exhaustive visit of the neighbourhood, the operation

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

can be expanded as a loop, where for each $x' \in N'(x)$, the solutions in the "extended" neighbourhood which may contain unfeasible solutions, the solution with the minimum $f(x')$ is sought if and only if x' is feasible, that is $x' \in N(x)$. The complexity of the neighbourhood exploration combines three terms: the cardinality of $|N'(x)|$, that is the number of subsets visited; γ_x, that is the time to evaluate their feasibility and finally γ_f, the time to evaluate the objective function for each feasible solution.

Additive objective function It was already discussed extensively that in general it's better to update the value of the objective function instead of recomputing it from scratch: this can lead to a good save in time. In particular, additive objective functions were considered: let's elaborate on that a bit more.

If it's assumed that the objective function is a sum of terms of an auxiliary function defined on the elements of the solutions, then considering the variation of the objective function associated to the introduction of a subset A of added elements in the solution x and the removal of a subset D of deleted elements from the solution x, then the variation of the objective function is

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

This happens in many neighbourhoods and in particular in the swap neighbourhoods, some examples are swap of objects in the KP, columns in the SCP and edges in the CMSTP. This update has two fundamental properties: it takes constant time for a constant number of elements $|A| + |D|$ and $\delta f(x,A,D)$ doesn't depend on x: this last concept will be studied in depth later.

Example: the symmetric TSP Let's give an example of the update of an additive objective function: consider the TSP with the neighbourhood denoted as N_{R_2} where two arcs are broken and two different arcs are added in such a way as to build again an Hamiltonian circuit, as pictured in Figure 8.14. The neighbourhood is not based on the swap operation, as a portion of

(a) Before. (b) After the transformation.

Figure 8.14: Example of a transformation for the neighbourhood N_{R_2}.

the arcs in the solution may be reverted, as pictured in Figure 8.14 (b); so, this neighbourhood is defined on the modification of two arcs and the inversion of a certain number of arcs, which is not constant and in general is $\mathcal{O}(n)$. Assuming, however, that the TSP is symmetric, that is the objective function is symmetric, then the variation of $f(x)$ is

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

made of a constant number of terms, as two arcs, (s_i, s_{i+1}) and (s_j, s_{j+1}), are being removed, but two arcs (s_i, s_j) and (s_{i+1}, s_{j+1}) are added; all the arcs that are reversed should be removed from the objective function and added again, but each of them has the same cost, being the graphic symmetric; so, the line of reasoning doesn't work for asymmetric TSP.

Evaluating or updating the objective: the quadratic case

What if the objective function is quadratic such as the one of the MDP, as computing it costs $\Theta(n^2)$? Moving from x to $x' = x \setminus \{i\} \cup \{j\}$, that is defining the neighbourhood as N_{S_1} a single swap, the update of the value of the objective function is

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

the computation depends on $\mathcal{O}(n)$. Most of the elements of the first sum (the distances between pair of objects that belong to x or the element j but not the element i) appear in the second sum (the distances between all the elements in x), so, when computing the difference they cancel each other out. There's a general "trick" for the simmetric quadratic functions with $d_{ii} = 0$:

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

as the terms that do not appear in the second sum are those involving j: so, all the distances from point j to each other point $x\setminus\{i\}$ appear both without j in the first position and with j in the second position; since the objective function is simmetric, this two distances are the same and they can be summed; so, the first and second set of distances in the last equation (d_{jk}, d_{ij}) form the overall set of distances that belong to the first sum but not the second. In the same way, the second term (d_{ik}) represents the terms that are in x but not in $x\setminus\{i\}\cup\{j\}$, that is the terms involving the element i. So, overall, the sum can be seen as the sum of $n+n+1$ terms, which is linear and not quadratic.

But, as each of the sums have a significant meaning, that is the first sum is the total distance of point j from the solution and the second the total distance of point i from the solution, so

$$\delta f(x,i,j) = 2(D_j(x) - D_i(x) - d_{ij})$$

and if $D_\ell = \sum_{k\in x} d_{\ell k}$ is known for each $\ell \in B$, the computation takes $\mathcal{O}(1)$. So, the quadratic time is reduced to a constant time, provided that the information D_ℓ is known.

Example: the MDP Given an instance of MDP with six point, as pictured in Figure 8.15 (a), where the solution is made of 3 out of 6 (those on the left) and the complement is given by the remaining point (those on the right). Suppose that it's needed to evaluate the exchange

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

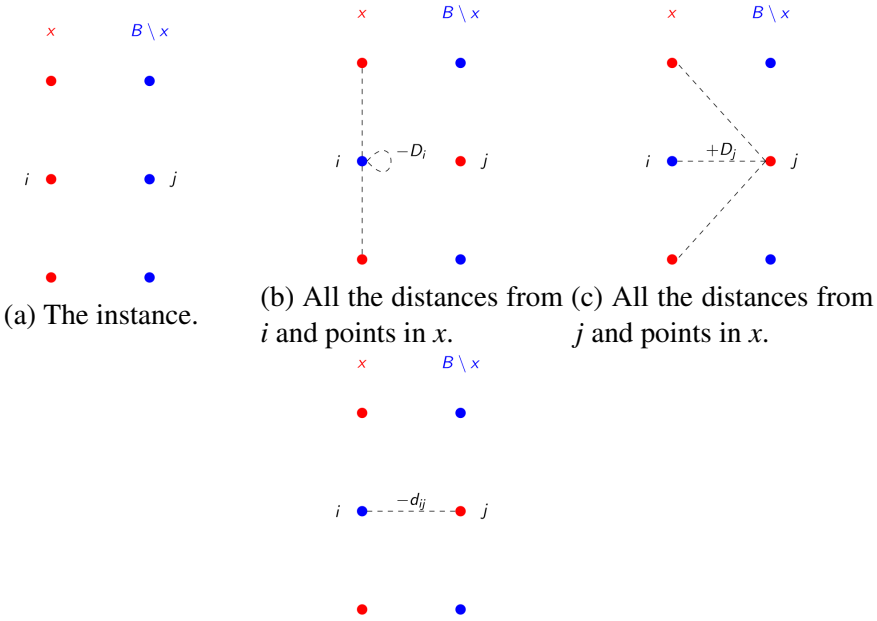
with $i \in x$ and $j \in B\setminus x$. Then, the first step is to reduce the total distances between the elements of x by removing the distances that involve the point i, represented in Figure 8.15 (b); once this is done, all the distances from the points in x to the point j are added, which are represented in Figure 8.15 (c), which generates a spurious term, that is d_{ij}, represented in Figure 8.15 (d), which should not belong to the objective function. In the end, the updated value is

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

and the cost is computed in $\mathcal{O}(1)$ time for each solution (given that D_ℓ is known). Once the function and the solutions are updated, it's necessary to update all the values D_ℓ; every element D_ℓ should be updated by subtracting its distance from i and adding its distance from j, that is

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

so the auxiliary data structure is update in $\mathcal{O}(n)$ time for each iteration. This data structure allows to reduce the computational complexity necessary for finding each solution from n^2 or n to a constant time. The number of solutions that were going to be checked is the number $n-k$, that is the number of the ground set minus those in the solution.



(a) The instance.

(b) All the distances from i and points in x.

(c) All the distances from j and points in x.

(d) The spurious distance d_{ij}.

Figure 8.15: A swap on the neighbourhood for the MDP.

Evaluating or updating the objective function: the nonlinear case

In some cases, one can update the objective function even if it's nonlinear. An example of such a feature can be seen in the case for the PMSP, where the idea is exactly the same as the one for the quadratic case, where some information is saved so that the value of the objective function can be computed in a more efficient way. The "scheme" is then to save aggregated information on the current solution $x^{(t)}$, use it to compute $f(x')$ efficiently for each $x' \in N(x^{(t)})$ and update it when moving to the following solution $x^{(t+1)}$.

Example: the PMSP Consider the transfer or swap neighbourhood: it was discussed previously that the objective funciton is going to change in a quite difficult way, as sometimes it doesn't change!

Figure 8.16: Instance of PMSP.

Consider the PMSP instance represented in Figure 8.16: moving any task from machine $M1$ to machine $M2$ doesn't change the objective function, as it refers to machine $M3$. Moving a task from machine $M3$ to another may modify the objective function in a way not directly proportional to the length of the task just moved: in general, the variation in case of a transfer of the objective function can be any number between the interval represented by the time duration of the task moved and the negative time duration ($[-d_{task}, d_{task}]$); in the case of a swap things are similar, so there's not an a priori evaluation.

Checking the overall objective function is expensive, as it requires to compute all the single partial sums and to find the maximum, so it's at least linear in the number of tasks. To do it in shorter time, one can notice that swapping two task to another machine, all the machines that are not involved in the movement won't change their overall computational time: so, what could be done is saving the completion time of each machine! In this way the new objective function is linear in the number of machine, which may be, again, not the best solution. To do even better, one can keep the indices of the two machines that have the maximum completion time: this is useful as one can compute in constant time the new completion times of the two machines interested in a swap or transfer operation, as previously said. What happens is that there are two new values: it suffices now to check if the largest of the two is larger than the overall maximum, which takes constant time. If one of the two machines is the machine with the maximum completion time, if it increases further, then its completion time will increase; if it decreases to a value less than the one of the second machine, tha latter would become first and in constant time one can retrieve the maximum machine. The point is that once the choice is done, one has to update all the retained informations: one by one the completion time of each machine (only two have changed) and the two machines with maximum completion time; this can be done using a max-heap, which takes $O(log(|M|))$.

The use of auxiliary information

Another example that's going to be presented is an example of additive objective function in which, however, the number of elements that are added or delete is not constant. Consider the neighbourhood N_{R_2} for the asymmetric TSP: the neighbouring solutions differ from x for $\mathcal{O}(n)$ arcs, the general neighbouring solutions differ from each other for $\mathcal{O}(n)$ arcs and if the pairs of arc (s_i, s_{i+1}) and (s_j, s_{j+1}) follow the lexicographic order the reverted path changes only by one arc. So, given a certain circuit starting from from a certain node pi_0 two arcs are canceled and the direction of a subpath in the solution is reverted. If the cost function is symmetric the updated objective function can be calculated in constant time. On the other hand, if the TSP is defined with an asymmetric cost function, this is no longer true! The arcs that take the place of the original ones change their cost, therefore part of the solution must be computed again, possibly $\mathcal{O}(n)$.

How to do this in constant time? The idea is to keep auxiliary information once again, but differently from before the auxiliary information kept is not based on the current solution but is based on the previous solution that was visited in the neighbourhood. So, starting from a current solution x, the neighbourhood is visited in a given, suitable order that must be chosen in a very smart way. When x_j is visited, the value of the objective function is computed using informations relative to x_{j-1}: this information is known by the visit of x_{j-1} itself! This chain of information goes up to x_1, which draws information directly from x.

(a) TSP instance before swap. (b) TSP instance after swap.

Figure 8.17: Swap for an asymmetric TSP instance.

Let's say that Figure 8.17 (a) is x_2 and 8.17 (b) is x_3. If one knows what "happened" between the two, one can easily know the variation of the objective function relative to x_3. In order to achieve so, one must know that with respect to the original solution x two arcs have been added, (π_i, π_{j+1}) and (π_{i+1}, π_{j+2}) and two arcs, (π_i, π_{i+1}) and (π_j, π_{j+1}), have been removed

and three arcs have been reversed (those between π_{i+1} and π_{j+1}). To compute the variation of the objective function due to this reversal one can use the variation of the objective function with respect to the inversion of the arcs computed in x'', which is already known. So, the variation of $f(x)$ between two generic neighbouring solutions is

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

where the fifth term is the cost of the reversed path and the sixth is the cost of the direct path. The first four terms change for each solution in the neighbourhood, but can be checked in constant time; the last two term can be updated in constant time by adding the "previous one", meaning the one calculated for the solution analysed before this one, plus the additional arc that is in the direct arc or in the reversed path

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

The only problem is: is it acceptable to explore the neighbourhood in a predefined order? The answer depends on the specific algorithm that's being used and the neighbourhood being considered. In the case in which there are several equivalent solution, always the first one is being accepted, inserting a bias in the exploration; further, if one doesn't want to explore completely the neighbourhood surely can't adopt this trick, which requires a systematic exploration of the neighbourhood.

8.3.2 Feasibility

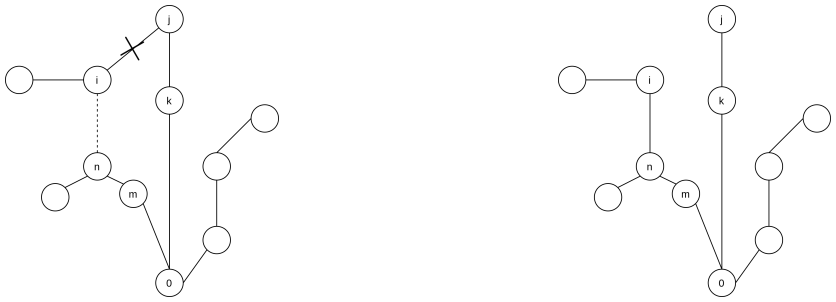
The fact that sometimes it is not possible to generate a strictly feasible solution set as a neighbourhood: if it's possible it's obviously a good idea, such as in the MDP with the swap neighbourhood and the MSTP; in some cases such as for the CMSTP, BPP or KP a strictly feasible neighbourhood is not guaranteed. So, it's possible that the operations that define the neighbourhood generate not only feasible solutions but also unfeasible subsets:

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

In this situation, it's necessary to first check the feasibility of the generated subsets by estimation of some function; for example, in the KP one has to estimate the total volume of a solution to check if it's higher than the knapsack volume, in the SCP one has to check how many columns cover each row and so on. The point is that even in such situation it's better to update these functions instead of recomputing them from scratch and the techniques that are used are exactly the same as those used to make easier the evaluation of the objective function presented above. Take, as an example, the update in constant time of the total volume of a subset in the KP, which can be updated by adding the volume of the added elements subtrating the volume of the removed ones (constant only if the number of added and removed element is constant).

Example: the CMSTP

Consider the swap neighbourhood N_{S_1} (add one edge, delete another) for the CMSTP. There's the concept of *capacity* in this problem, so one could increment and decrement the volume occupied by each subtree and then check in constant time if these volumes violate the overall capacity constraint or not. This is valid if the neighbourhood is defined on the swap of vertices, but not for edges.



(a) Before exchange (b) After exchange

Figure 8.18: Edge exchange in CMST

In this case, the number of vertex that are moved following a edge swap is not constan, which is not known a priori. Hence, it's not easy to understand how much volume is moved from a subtree to another following a swap: of course one could get this information by visiting each moved subtree, but this would take, in the worst case, linear time. The idea to achieve this in constant time is the following. Let's consider the cut of the edge (i, j) and the addition of the edge (i, n) as pictured in Figure 8.18 (a). The idea is that if one knows the total volume of the subtree appended to an edge, whenever an edge is cut one of the two vertices is the father and the other is a son on the path that goes from the subtree up to the root of the whole tree. Distinguishing this two types of nodes on each edge is possible, and when one decides to cut the edge one can try to understand what is the total volume of the vertices that are still appended to the son, as taking the subtree and instead of adding it in any general way to the remaining subtree it is added with an edge that is incident to the son, in other words, taking as a reference the previous Figure, attaching the "dangling" subtree exactly from the orphane node i and not from the unnamed node at its left, hence restricting the neighbourhood to swaps where an edge is cut and another edge that's incident to the same vertex that was incident on the cut edge, the resulting weight could be *saved somewhere*, and if one knows the total volume from the node one can check if it can be feasibly added to the total volume of the subtree in which the new subtree is to be inserted, in turn checking if the overall total capacity of the tree is respected.

One can save in a vector in which each component corresponds to a vertex the total weight of the subtree that's appended to the subtree in the current solution. Then, checking feasibility is easy as an edge is cut, father and son are recognized and since the volume of the subtree appended to the son is known one can try and add the dangling subtree to other subtrees; then, for each possible addition, one must know the volume of the subtree where the dangling one is added, which is an information that must be stored somewhere as well, so that one can check if the addition of the volume of the dangling subtree fits in the capacity of the tree, which takes constant time. Of course, when in the end one chooses which solution to take, all auxiliary informations must be update: for each node the subtree to which it belongs and for each node the total weight that's appended, which takes $\mathcal{O}(n)$ time, as only a subtree must be updated and this can be done with a visit. To give a practical example, once that the edge (i, n) in the Figure 8.18 is added one visits the first subtree rooted in m and marking the weights and subtree for each node.

A general scheme of sophisticated exploration

It's possible to explore a neighbourhood in a sophisticated fashion in order to save time, which is fundamental, as the saved time can be used to perform more search and yield a better result. In order to achieve that, additional data structure are usually needed, which must be initialized

and updated: these structure can be global or local, where the distintion is made as the global structure keeps information about the current solution, which are the ones seen first as the vector of total distances from each point from the solution in the MDP or the total processing time in each machine for the PMSP, or the total volume occupied by each subtree in the CMSTP where the local structure refer to the previously visited solution in the neighbourhood, which concerns for example the total cost of the reversal during the visit of the N_{R_2} neighbourhood for the asymmetric TSP. So, global structures change for each move performed whereas local structures change for each feasible solution visited.

Algorithm 12 Sophisticated Exchange Heuristic Pseudocode

```
1:  procedure STEEPESTDESCENT(I)
2:      x := x^(0)
3:      GD := InitializeGD()
4:      Stop := false
5:      while Stop = false do
6:          x̃ := 0
7:          δ̃ := 0
8:          LD := InitializeLD()
9:          for ∀x' ∈ N(x) do
10:             if f(x') < f(x̃) then
11:                 x := x̃
12:             end if
13:             LD := UpdateLD(LD, x')
14:         end for
15:         if f(x̃) ≥ f(x) then
16:             Stop := true
17:         else
18:             x := x̃
19:             GD := UpdateGD(GD, x̃)
20:         end if
21:     end while
22:     return (x, f(x))
23: end procedure
```

Partial saving of the neighbourhood

There's a nice and sophisticated technique used to make the exploration of the neighbourhood much faster, that doesn't always work but works in several problems, based on the idea that sometimes the solutions of a problem is actually composed of independent parts (for example in the BPP there are objects partitioned into containers, in VRP costumers are divided for each vehicle, in the CMSTP the vertices are divided into subtrees, ...) which has an implication on the effects of the moves: in general terms, performing an operation $o \in \mathcal{O}$ on a solution $x \in X$, the variation

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

doesn't depend on x or depends only on a part of x and in particular it happens that there exists a large substet $\tilde{\mathcal{O}} \subset \mathcal{O}$ such that

$$\delta f(x',o) = \delta f(x,o) \;\; \forall x' = o(x), \;\; \forall o \in \tilde{\mathcal{O}}$$

The variation in the objective funciton due to the application of an operation $o(x)$, doesn't always depend on the whole solution x, so it can be applied again on x', making it advantageous to

1. compute $\delta f(x,o) \ \ \forall o \in \mathcal{O}$ and save the values
2. perform the best operation o^*, generating the new solution x'
3. retrieve $\delta f(x',o) o \in \tilde{\mathcal{O}}$ as they are still valid values and recompute $\delta f(x',o) o \in \mathcal{O} \setminus \tilde{\mathcal{O}}$ and save it
4. go back to (2)

Example: the CMSTP Consider a given CMSTP instance, where one can perform the red exchange, that consists in deleting the edge (i,j) and replacing it with the edge (i,n) but also the blue exchange can be applied, as pictured in Figure 8.19.

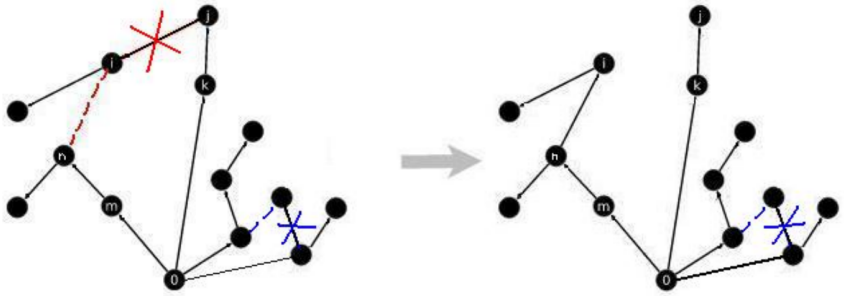


Figure 8.19: A move in the N_{S_1} neighbourhood for the CMSTP.

Given the two possible exchanges, maybe one is better than the other! Let's say that the red exchange decreases the objective function from 100 to 95 and the blue exchange decreases it to 99; so, the effect of the first exchange is -5 and the effect of the second is -1. Obviously, it's better to chose the first exchange, which is going to be chosen, obtaining the solution on the right of the picture. Exploring the neighbourhood, looking for all the possible exchanges, one can easily see that the blue exchange is stil appliable, as it did not involve the subtrees involved in the previuos operation and exaclty as before its effect is a decrease of -1 in the objective function, making it 94. This exchange is also feasible as it was before, because nothing changed in the distribution of vertices in these two subtrees. Therefore, one can keep the set of feasible exchanges witht he associated values δf, perform the best exchange $(\tilde{i},\tilde{j})$ (to be intended as two arbitrary edges) and remove from the set the exchanges involving one of the two branches involved in the best one and then recompute the exchanges that involve any such branch and add them to the set.

For example, this technique can't be used for the MDP, as in it every exchange interacts with every other exchange because the objective function is given by the sum of the total distance of every point with every other point in the solution, so every point in every solution interferes with the value of the objective function.

8.3.3 Tradeoff between efficiency and effectiveness

There's a tradeoff between efficiency and effectiveness, as the complexity of the heuristic depends on the number of iterations performed (t_{max}) which is hard to predict a priori, but then each iteration depends on how many solutions are visited in the neighbourhood or a superset

of it and how the feasibility and cost of every subset is computed, which can be determined with the usual, standard techniques to compute the complexity of an algorithm, which can be improved as seen above.

It's rather clear that the first two elements, the number of iterations and the cardinality of the visited neighbourhood, conflict with each other concerning the choice of the neighbourhood; in other words, choosing a very small neighbourhood (so the cardinality of it is decreased), means that probably a lot of iterations will be done, as in a small neighbourhood there can't be many good solutions. Taking a large neighbourhood, on the contrary, means that there may be very good solutions and the number of iterations is decreased; however, the overall speed depends on both factors, so one has to find a neighbourhood large enough that it contains many good solutions but small enough to be explored quickly. The tradeoff is particularly complex as one doesn't necessarily reach the same local optimum with different neighbourhoods, quite the contrary in the case of very different neighbourhood sizes, as discussed previously. In the end, enlarging the size of the effectiveness of the algorithm improve at unknown speed as large neighbourhood have better local optima, but efficiency quickly worsen, as large neighbourhood are slow to explore.

Fine tuning of the neighbourhood

It is also possible to define a neighbourhood N and tune its size, as one is not bound to *choose* a neighbourhood. Given a neighbourhood, one can modify it by small modifications; referring to the case of the additive objective function as it's easier to have an intuition on it, if one just considers that some elements that are out of the solution have a low cost, one concludes that they're promising; if elements inside the solution have a high cost, one may conclude that these must be removed from the solution. So, why not consider the exchanges between element of low cost outside with element of high cost inside? It depends, but it could be a good idea (in particular for additive objective function) not to explore the whole neighbourhood N but only to define a subneighbourhood consisting of exchanges between costly element inside the solution with cheap elements outside the solution. This is an idea to reduce the neighbourhood: the fine tuning goes in the direction discussed previously. One is not bound to do this on the basis of the data but also on the basis of the results, in a sense it is not memory mechanism based on several runs of the whole algorithm but based on results of the algorithm itself in the previous iterations. For example, visiting an n^3 neighbourhood with thousands and thousands of solutions but maybe the seconds solution is already an improvement with respect to the given one: does it make sense to go on and find the best solution overall in the neighbourhood? This is called **first-best** strategy, opposed to the **global-best** strategy; the first-best strategy stops as soon as the first improvement is found

$$\text{if } f(\tilde{x}) < f(x) \text{ then } x := \tilde{x}, \text{break};$$

So, the idea is that probably one doesn't improve much for each iteration, but a lot of time is saved: whether or not this is a good idea depends; if the problem has a smooth landscape, what happens is that probably all solutions are rather similar to each other, so the best won't be much better, so it may make sense to use a first-best strategy; if the landscape is ruggedm the best solution of the neighbourhood could be much better and it may be better to use a global-best strategy.

There are lots of tools to tune the size and the structure of the neighbourhood and it's particularly important to make the exploration of the neighbourhood as efficiencient as possible, because if it can be done in the smallest time possible a lot of time is going to be saved.

8.4 Very Large Scale neighbourhood search

After introducing the idea of search in the neighbourhood, moving from feasible solution to feasible solution, based on the idea that for each solution only a subset of feasible solution called neighbourhood can be reached by the use of exchanges and after seeing how to define and explore neighbourhoods, the tradeoff between quality and efficiency in the definition and visit of the neighbourhood, the next concept discussed is *enlarging* the neighbourhood as much as to have an explonential or high-order polynomial number of elements in the neighbourhood.

If one defined a neighbourhood that's explonential or high-order polynomial with respect to the size of the ground set then one can find very good local optima, in turn requiring high computational time if one explores the solutions in the neighbourhood exhaustively. A solution to this problem is finding another way of visiting the solutions, using a low-order polynomial time, combining a very large neighbourhood with exponential size with a low visit of polynomial cost. This can be done in two ways:

1. define the neighbourhood in such a way that one can avoid visiting exhaustively, but apply a suitable algorithm to visit efficiently. After all, the exploration of a neighbourhood, is a combinatorial optimization problem, as one's trying to find the best solution in a finite set: this problem is exactly the problem that's being solved with the additional constraint on the contents of the solution.
2. expore the neighbourhood heuristically and return a promising neighbouring solution instead of the best one.

8.4.1 Efficient visit of exponential neighbourhoods

Four examples of this technique will be presented. The first will be based on the idea of building an auxiliary problem in which there's a matrix and a SPP instance has to be solved: this method is called *Dynasearch*. Then, another technique considered involves making cyclic exchanges between components of the solution, which will be modeled as a search for negative cost circuit in an auxiliary graph. The two last examples involve shortest path problems in an auxiliary graph: sometimes easy shortest path problems, as in the case of *order and split* and sometimes an $\mathcal{NP}$-hard problem with an additional cost as in the case of *ejection chains*.

So, all this cases have in common the fact that starting from a given problem one has to transform the problem of finding the optimal solution in the neighbourhood in an auxiliary subproblem on a matrix, graph or generically on a combinatorial structure and then solve the problem on such a structure.

Dynasearch

Previously, a technique to accelerate the exploration of a neighbourhood by exploring it not completely was presented. The idea was that if a solution of a problem is given and such solutions are divided into components, sometimes an operation $o \in \mathcal{O}$ that works on some components of the solution is completely independent of operations $o' \in \mathcal{O}$ that work on different independent components; the example shown was the one of the CMSTP.

In Dynasearch, the idea is that given a similar situation where the variation $\delta(x,o)$ of the objective function implied by an elementary move $o \in \mathcal{O}$ depends only on a part of the solutions, the order with which different **independent** moves are performed is irrelevant:

$$f(o'(o(x))) = f(o(o'(x)))$$

And if $f(\cdot)$ is additive, the effect of the two exchanges is simply summed. In general, provided that they are independent, why not applying more than just two moves? The idea is to find the *best combination* of moves that can be applied to a certain solution to obtain a better one. Another example of this behaviour is portrayed in the TSP in which two exchanges can be performed provided that they operate on disjoint segments of a circuit: an example of this is Figure 8.20, where a solution for the TSP starts in vertex u_1 and after following a path to u_n it goes back to the first ($u_{n+1} \equiv u_1$). Suppose that two different exchanges can be applied:

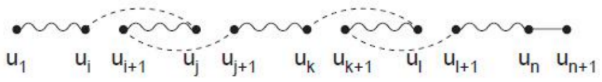


Figure 8.20: TSP instance with two optional exchanges.

break the arc (u_i, u_{i+1}) and (u_j, u_{j+1}), adding two arcs reversing the intermediate subpath, and break the arc (u_k, u_{k+1}) and (u_l, u_{l+1}) as the second exchange. It's clear that performing the first exchange leaves the second exchange feasible and as well for the reverse. The effect of the two exchanges is given by the sum of the costs two added arcs minus the sum of the costs of the deleted arcs and, assuming the TSP instance to be simmetric, the reversal of the subpaths does not change their costs; so the final effect of performing one of the two is perfectly independent from the other, so both can be applied together. Consider the exchange (u_i, u_{i+1}) with (u_j, u_{j+1}) and the exchange (u_j, u_{j+1}) with (u_k, u_{k+1}). This two clearly aren't independent and they are incompatible as performing one of them the other wouldn't "be able" to remove one of the two arcs in the exchanges as it was already removed. Another case of incompatibility is performing two exchanges where the second acts on the subpath reversed by the first exchange.

Move search How to compute the subsets of moves that have the best total effect and that are reciprocally compatible? Luckily, it's not that difficult: the idea is to build a set of moves with mutually independent effects on the feasibility and the objective value of the solution and avoid to include two moves that are incompatible.

A matrix, called *improvement matrix*, can be built where the rows represent the components of the solution, where *components* depend on the problem: for a CMSTP they can be subtrees, for the VRP they can be circuits, for the TSP they can be circuit segments and in general are the "parts" that are effected by the moves. The columns represent the elementary moves and each column has a value equal to the objective improvement $-\delta f$. If the move j impacts on component i then $a_{ij} = 1$, otherwise $a_{ij} = 0$.

The problem is formulated as determining the optimum packing of the columns, that is the subset of nonconflicting columns of maximum value; the Set Packing Problem is in general $\mathcal{NP}$-hard, but on special matrices it is polynomial (e.g. for the TSP) and if each move modifies at most two components the rows can be seen as nodes, the columns can be seen as edges and each packing of columns becomes a matching, which is a polynomial problem. If the SPP is not polynomial, one may solve it if it's "small" or an heuristic solution may be sufficient.

Again, the idea is that instead of just performing one move and saving part of the neighbourhood for the future, the whole neighbourhood is used, making a lot of moves together, finding a much better solution: the problem is that at the next step one has to rexplore the whole neighbourhood, building the matrix from scratch again. There's also another effect: making just one move saves part of the remaining moves for the future, but one has to recompute from scratch all the moves on the part of the solution that have been modified, which means that new moves will arise: so, the single move may not be that good, but the following moves can be chosen in a larger set.

Cyclic Exchanges

A second possibility to explore in polynomial time exponential size neighbourhood is the use of **cyclic exchanges**. Once again, the solutions can be partitioned into components (vertices devided into subtrees, edges into subtrees, nodes into circuits, objects into containers and so on) and the ground set is made of the cartesian product of the objects and components and a single solution is partitioned in components $S^{(l)}$ composed of pairs (i, S_i).

Quite often, these problems have a feasibility that depends on each single component, so in order to cheack if a certain solution is feasible one checks the feasibility of each component: for example, in the BPP one may check if each bin's volume is respected and the same for the CMSTP, the VRP and so on. Also, often the objective function is additive with respect to the components:

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

It is natural to define for these problems the set of operations $\mathcal{T}_k$ which includes the transfer of k elements from their component to another and to derive from $\mathcal{T}_k$ the neighbourhood $N_{\mathcal{T}_k}$. The single transfer $\mathcal{T}_1$ has the characteristic that quite often its subsets are unfeasible, as if the additional constraints are tight it's really difficult to move an object from a partition to the other without breaking some constraing. On the other hand, considering a neighbourhood with a large k is really large and computing the objective function becomes difficult and the overall complexity of the exploration is large. A good situation would be having a subset of $N_{\mathcal{T}_k}$ that is large but efficient to explore.

Improvement graph The solution to this problem is to build an auxiliary graph solving an auxiliary subproblem on it. Such a graph is known as the **improvement graph**, which is a quite refined structure, which allows to describe sequences of transfers:

- each node i of the graph corresponds to one of the elements of the current solution x, in other words, considering for example the CMSTP, for each vertex there's a node in the improvement graph.
- the arcs (i, j) connect two nodes, but the meaning of the arc is rather complex:
  - the arc corresponds to the transfer of element i from its current component S_i to the current component S_j of element j
  - the arc correspondds to the deletion of element j from component S_j

  so, an arc going from node i to node j represents the transfer, in which the element i is removed from its current component S_i and it's inserted in another component to replace the element j, which is deleted from its current component. This is not a feasible move, as j must be inserted somewhere! A single arc, then, doesn't exactly represent an elementary move - however, for now this point will be ignored and explored later.
- the cost of an arc, c_{ij}, corresponds to the variation of the contribution of S_j to the objective

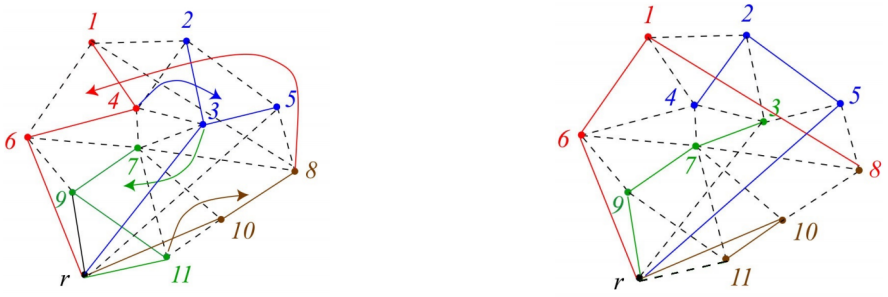
  $$c_{ij} = f(S_j \cup \{i\} \setminus \{j\}) - f(S_j)$$

  with $c_{ij} = +\infty$ if it is unfeasible to transfer i deleting j.

The problem is that following a single transfer, the deleted element j must be inserted somewhere else, otherwise the constraint of partitioning is violated: j is then inserted in another

component, therefore another arc is needed (j,k), which will then delete the element k which will have to be inserted somewhere; so, in the end, it's necessary to create a circuit, so that the first arc is (i,j) and the last is (q,i). The cost of the circuit corresponds to the cost of the sequence, but only if each node belongs to a different component.

Example: the CMSTP Let's discuss an example based on the Capacitated Minimum Spanning Tree problem as pictured in Figure 8.21. The CMSTP instance is an undirected graph with



(a) A solution with possible exchanges.

(b) After the exchanges.

Figure 8.21: Moves on the solution for the CMSTP.

cost on the edges and weights on the vertices (which aren't represented in the picture) and a solution is given, represented by the continuous lines in the picture. The solution is divided intro four subtrees, each of different color; consider the composite move described by the circuit $(4,3),(3,11),(11,8),(8,4)$: vertex 4 moves into the blue subtree to replace 3, vertex 3 moves into the green subtree to replace vertex 11, vertex 11 moves into the brown subtree to replace vertex 8 and, finally, vertex 8 moves into the red subtree to replace vertex 4.

Is this solution feasible? From the point of view of the partition it is; from the point of view of the weights it depends on whether or not the single arcs are feasible, so if it's feasible to replace 3 by 4 and so on - if the two weights are the same it's certainly feasible, if they're different ona has to consider the original weight of the subtree S_j, adding the new weight w_i and removing the weight w_j; if the total weight is still feasible, then the composite move is feasible concerning that single subtree. If, instead, the move is unfeasible, the arc must be forbidden. The total cost is given by the variation of the cost of the blue subtree, that is the cost of $(4,3)$, plus the variation of the cost of the green subtree, that is the cost of $(3,11)$, and so on. In order to evaluate the variation in cost, for example considering the red subtree, one must compute the "before" and "after" spanning trees and give the variation of the cost. Of course, if the cost can be updated with some smart procedure it's better.

Why is it required for each subtree to appear exactly once in these exchanges? The reason is that visiting two nodes that belong to the same component it is no longer guaranteed that feasibility and cost just depend on the single arcs, as it is possible that the two moves are feasible one by one but unfeasible applied together.

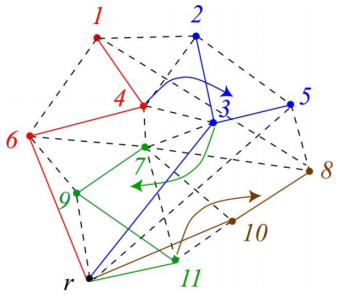
Search for the minimum cost circuit Solving the problem of finding the minimum cost circuit is not easy, as it's - of course - $\mathcal{NP}$-hard, but a lot of nice things can be achieved. First of all, the constraint of visiting only once each component allows a rather efficient dynamic programming algorithm: if the components are r, then the circuit has at most r arcs.

A second, complex property, is that since a sequence of numbers with negative sum always admits a cyclic permutation with all negative partial sums, one can delete all the partial paths of

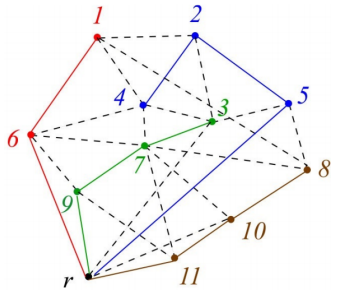
cost greater or equal to 0; in other words, given numbers such as $+1$, $+2$, $+3$ and -10 which sum to a negative number, at least one cyclic permutation such as $2+3-10+1$, $3-10+1+2$ or $-10+1+2+3$ have all negative partial sums: in this case, it is $-10+1+2+3$. This means that it isn't necessary to generate all the circuits starting from all the points, but as soon as one gets a positive partial sum the circuit is dropped as either it isn't part of a negative sum or it is a part of a negative circuit but it can be found in another way.

Of course, as a third possibility, one could use heuristic algorithms; some heuristics apply the Floyd-Warshall algorithm, which finds all the point to point shortest path in a graph but fails if there's a negative circuit, which is exactly what's needed! Some other possibility is to compute the minimum average cost circuit, computed as the total cost divided by the number of arcs. However, these violate the constring on the components, but provide an underestimate (or, if Floyd-Warshall succeed, the proof that no negative circuit exists) or a negative relaxed circuit that can be feasible by chance or can be modified to obtain a feasible one.

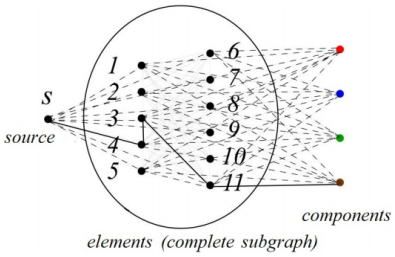
Noncyclic Exchange Chains Cyclic exchange have the properties to keep the same number of elements in each component. It was already seen that swapping elements keeping the same number of elements in each component makes it impossible to find all the possible solutions and one may not find the optimal solution if it has different cardinality.



(a) CMSTP solution and possible exchanges.



(b) After the exchanges.



(c) The extended noncylic exchange improvement graph.

Figure 8.22: CMSTP instance and its noncylic improvement graph.

In order to change the cardinality of each component, one must find a chain of transfers such that it's non-cyclic, starting by removing an element, putting that element in another component and maybe making a chain of transfers that keeps the same number of elements and then, finally, avoid putting the last element in the original component but in another. In an improvement graph this is represented by increasing the graph; one adds to the improvement graph what follows:

- a source node, which is fictitious and doesn't represent any node of the problem
- a fictitious 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

Figure 8.22 (c) shows the complete graph in which every node is connected to every other node, apart from infeasibility derived from the capacity with the new source node and the components nodes, one for each subtree, with added arcs. Consider now the exchange

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

which represents the fact that node 4 is taken out of the red subtree; the second arc represent the removal of node 3 from the blue subtree and the insertion of 4 in it; the third represents the removal of 11 from the green subtree and the last arc represents the insertion of 11 in the brown subtree. The situation is represented by a path going from $s \rightsquigarrow S_4$ in the improvement graph and, again, each component is visited exactly once.

Order-first split second

Let's describe another case in which the shortest path problem can be solved in order to find the optimum in an exponential neighbourhood, but this time as an exact solution of an easy shortest path problem. These methods are known as **order-first** split second methods, which concern once again partition problems, or problems in which the solutions partition objects into subsets. The idea is the following: first, one takes the elements and they're are inserted, in order, in a given permutation that is a starting permutation. Then, the elements must be partitioned into components respecting the additional constraint that the elements of the same components must be adjacent (consecutive, subsequent) in the permutation. In other words, the permutation is *cut* at given points.

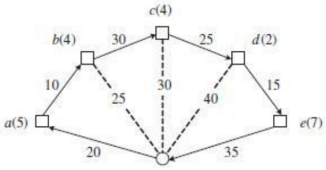
Of course, the resulting solution will depend on the starting permutation, as one can't put together things that aren't adjacent in it. Well, the methods is based on the change of the permutation! One can think of a two-level method in which in an upper level, in an outer loop, one produces permutations based on some idea and then at a lower level the partition for the current permutation is optimized.

The problem of this method is not the computation of the optimal partition, but there's no *nice* way to select a permutation and the point is that changing the permutation sometimes the same solution is obtained, so there are many more permutations than solutions, making the operation inefficient.

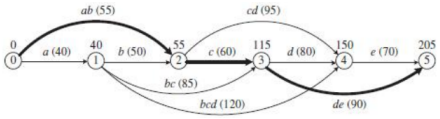
The auxiliary graph To solve the problem of optimally partitioning the permutation one has to build a graph, that will be used in a specific way. Given a permutation $(s_1,\cdots,s_n)$ of the elements, each node s_i corresponds to an element s_i, plus a fictitious node s_0. Each arc (s_i,s_j) with $i<j$ corresponds to a potential component $S_\ell=(s_{i+1},\cdots,s_j)$ 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 $f(S_\ell)$ and if the component is unfesible, the arc simply doesn't exist.

If one can find a path going from the source node s_0 to the last node, one obtains a solution that is a partition of the elements. The cost of the path coincides with the cost of the partition and since the graph is acyclic finding the optimum path costs $\mathcal{O}(m)$ where $m \leq n(n-1)/2$ is the number of arcs.

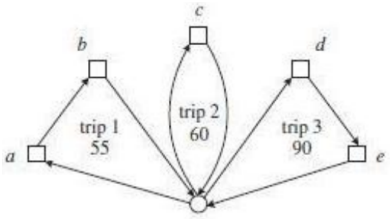
Example: the VRP In this example based on the Vehicle Routing problem, the graph is based on a single depot and five customer nodes. Each customer has a weight associated to it, each arc has a cost and the capacity $W=10$. The problem is solved by finding a number of vehicle that service all these customers with a minimum total cost such that each vehicle doesn't load more than the capacity W.



(a) VRP instance.



(b) The order first split second graph.



(c) The optimal path.

Figure 8.23: VRP instance and its OFSS solution.

The five nodes can be put in order building the auxiliary graph represented in Figure 8.23 (b), where each node represents an element and there's a fictitious node that is the origin. The arcs go only *forward*, from nodes that are "before" in the sequence to nodes that are "after". Considering the arc $(1,2)$ means considering the component that includes only the element 2; the arc $(1,3)$ means considering the component that include the elements 2 and 3 and so on. The cost of arc $(0,2)$, that represents a vehicle serving nodes 1 and 2 corresponds to the minimum cost of a path from the depot to nodes 1, 2 and back; theoretically one sholud solve a TSP problem and since it's very small one can solve it exactly - otherwise, obviously one must use an heuristic. The minimum cost of the path is $20+10+25=55$, that is the cost of the arc $(0,2)$. There's no arc $(0,3)$: this is because such an arc would represent a vehicle serving elements 1, 2 and 3, so the total weight is $5+4+4$ which exceeds the capacity, so it's not feasible.

The fact that the minimum path in the graph can be found in polynomial time is very good, as the neighbourhood, the number of possible partitions of nodes, is exponential. The only problem is that changing a partition one could get the same solution.

8.4.2 Heuristic visit of extended neighbourhoods

So far were discussed cases of problems in which, at least theoretically, with the cost of having an $\mathcal{NP}$-hard problem, one could find the best solution of the neighbourhood. Now, let's turn to cases in which one has to decide since the beginning that the extended neighbourhood will be visited with the use of a heuristic, as the size of the neighbourhood is not even perfectly defined. These methods are the Variable Depth Search and the so called Iterated Greedy or Destroy-and-Repare method.

Variable Depth Search

The **variable depth search** method is based once again on the idea of parametrisation of a given neighbourhood: so, given a set of operations $\mathcal{O}$ one can apply a sequence of k such operations and the result, if it's feasible, it's a neighbouring solution, so

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

the parametrisation here occurs in the number k, as if it's small one obtains a small number of solution which is probably not that nice; on the other hand, increasing k may yield very good

solutions but at a very high computational cost. The idea is to build a composite move as a sequence of elementary moves, which aren't required to be independent of each other, on the contrary they're bounded to work "in a given direction" so that in the end one can explore the final result of this method. The basic idea is that starting from a solution the algorithm visits its

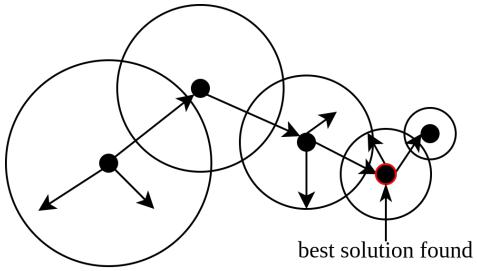


Figure 8.24: Schematic of VDS search.

neighbourhood; instead of computing the value of the objective function of each of the subsets in the neighbourhood, the algorithm starts from each of those solution and a whole local search run is applied to it. In a sense, this could be seen as a sort of rollout or lookahead algorithm, in which instead of adding an element and then running the constructive heuristic down to the end and using the final result as a selection criterion to understand if that element was good or not, this technique does something similar, running a local search algorithm until it yields a solution and the final result is used to understand if it's good to select a certain subset in the neighbourhood instead of another one. In other words, the algorithm defines a composite move as a sequence of elementary moves:

- consider each neighbour solution obtained with an elementary move
- make a sequence of moves optimizing each elementary step, forbidding backward moves and also allowing worsening moves
- terminate when the solution becomes worse than the starting one, so the length k of the sequence is not fixed a priori, but there's a fixed maximum
- return the best solution found along the sequence.

The difference between this algorithm and a local search is that in this case the search doesn't start from point x', as it would be extremely inefficient; the rollout algorithm increased the complexity of the basic constructive heuristic multiplying it by n^2 as for each of the n steps it *tried* $\mathcal{O}(n)$ times the basic heuristic: this is clearly quite inefficient. Using this technique for exchange heuristic would be much worse as the number of possible neighbouring solution is not, in general, $\mathcal{O}(n)$ but can be much larger and the number of steps of the local search needed to reach a local optima is not bounded by $\mathcal{O}(n)$ but it's bounded by t_{max}, the number of iterations, which is unknown and each single iteration has a complexty which is almost surely quite heavy; the problem with the general idea is that it's probably impractical.

In order to make it easier, one chooses a starting point and doesn't explore neighbourhood, instead start from a candidate solution in the neighbourhood of the starting one, explores a neighbourhood $\hat{N}$ that is smaller (usually strictly smaller) than the origianl neighbourhood N, so that this exploration is fast(er); finds the new solution and instead of stopping when the new solution isn't better than the current one go on until the new solution is worse than the original solution. So, referring to Figure 8.24, in each step one can improve the cost, then worsen it provided that it never gets back to the original cost; the reason for this is that in this way one

can go farther away. If at the first step one finds a solution that's worse than the first solution the exploration stops.

Algorithm 13 Variable Depth Search Pseudocode

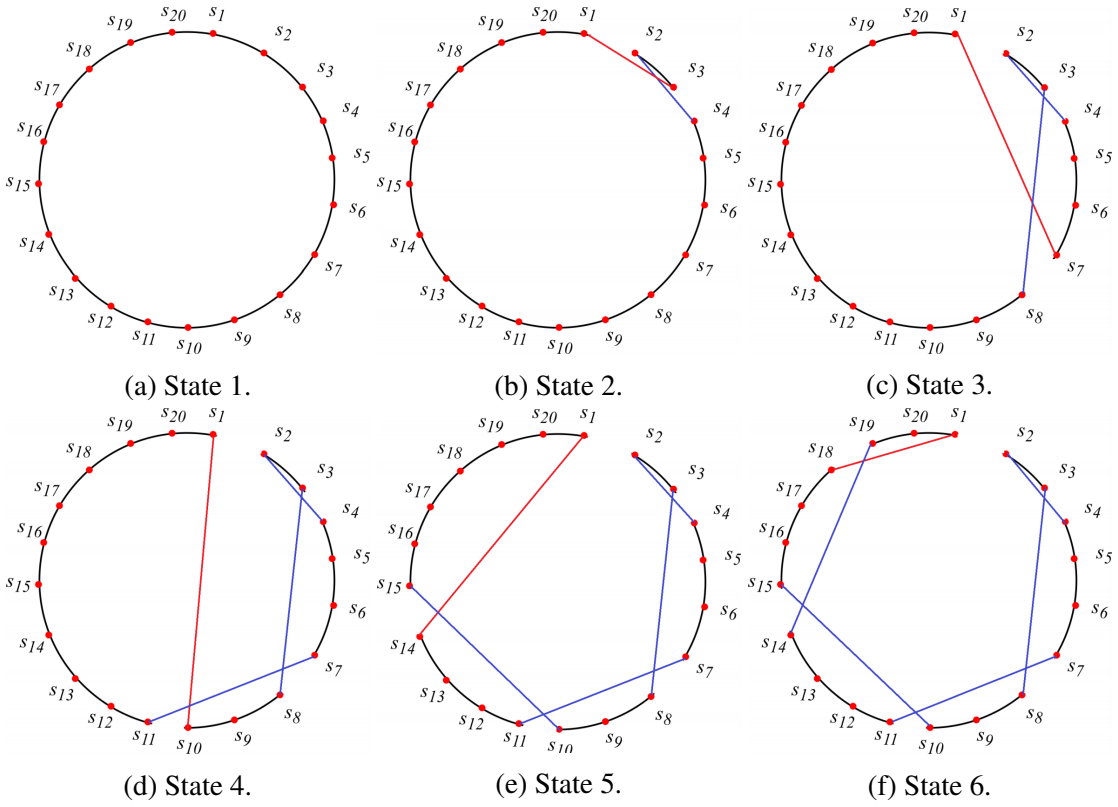
```
1:  procedure VDS(x')
2:      y := x'
3:      y* := x'
4:      Stop := false
5:      while Stop = false do
6:          ỹ := arg min_{y∈N̂(x)} f(y')
7:          if f(ỹ) ≥ f(x^(t)) then
8:              Stop := true
9:          else
10:             y := ỹ
11:         end if
12:         if f(ỹ) < f(y*) then
13:             y* := ỹ
14:         end if
15:     end while
16:     Compute f(y*)
17: end procedure
```

Why stop as soon as a solution worse than the original one is found? The reason for this is that, once again, there are nice properties concerning the sequences of solution, already mentioned referring about the exchanges. So, concluding, with respect to the standard steepest descent exploration VDS finds a local optimum for each solution of the neighbourhood, performing a sort of one-step lookahead; admits worsenings along the sequence of elementary moves, but never with respect to the starting solution and makes moves that increase the distance from the starting point to avoid cyclic behaviours by gradually restricting the neighbourhood.

Since the mechanism can become computationally heavy, the elementary moves often use a strictly reduced neighbourhood $\hat{N} \subset N$ as mentioned above, which is explored with the first best strategy, as well as the starting neighbourhood N.

Lin-Kernighan's Algorithm for the symmetric TSP An example application of VSP is the **Lin-Kernighan's Algorithm** for the symmetric TSP. It's still the best performing heuristic algorithm on the symmetric TSP - actually a variation of it which includes refinements of the neighbourhood exploiting some additional information. The basic neighbourhood N_{R_k} consists in deleting k arcs from the solution x adding k different arcs in such a way that the new solution is still an Hamiltonian circuit. Lin-Kernighan algorithm is a VDS with sequences of 2-opt exchanges, that is delete 2 arcs and add 2 arcs. If one desires to make k exchanges, each k-opt exchange is equivalent to a sequence of $(k-1)$ 2-opt exchanges, each deleting one of the two arcs added by the previous one: here one can find the realization of the concept of building complex moves by the use of many simple moves.

For each solution $x' \in N_{R_2}(x)$, 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 the exchange (i', j') obtaining x'', then evaluate the exchanges that delete $(s_{i'}, s_{j'})$ and each arc of $x \cap x''$ and so on. If the best solution among $x', x'', \cdots$ is bettern than x accept it.



(a) State 1. (b) State 2. (c) State 3.

(d) State 4. (e) State 5. (f) State 6.

Figure 8.25: Chain of moves in Lin-Kernighan's algoritm for the symmetric TSP.

Starting from the situation pictured in Figure 8.25 (a), which represents an Hamiltonian circuit, the algorithm explores the N_{R_2} neighbourhood, meaning that it considers all possible pairs of non-adjacent arcs, which are $20 * 18/2$, a large number. Suppose that the algorithm chooses the exchange $(1,3)$, so the arcs (s_1,s_2) (s_3,s_4) are removed. This exchange has the result pictured in Figure 8.25 (b), so the path between $(s_2,\cdots,s_3)$ is reversed and the arcs (s_1,s_3), (s_2,s_4) are added. Now, the local search must start from this point: this local search won't consider all possible R_2 exchanges but only the two exchanges where (s_1,s_3) and one of the other edges are removed. So the algorithm explores, one by one, each possible solution: let's say that the best one is $(1,7)$, which removes (s_1,s_3) and (s_7,s_8), adds (s_1,s_7) and (s_3,s_8) and inverts the path $(s_3,\cdots,s_7)$ (Figure 8.25 (c)); this 3 exchange is obtained by making two 2 exchanges which are particularly related.

The algorithm goes on and it's forced to remove the arc (s_1,s_7), as this is the rule that limits the neighbourhood; however, any other arc other than those added by the procedure (the blue ones in the pictures) can be removed. Let's say the best exchange is $(1,10)$, so (s_1,s_7), (s_10,s_11) are removed and (s_1,s_10), (s_7,s_11) are added and $(s_7,\cdots,s_10)$ are inverted (Figure 8.25 (d)). Let's suppose, then, that the algorithm choose as exchange $(1,14)$ (Figure 8.25 (e)), resulting in the situation pictured in Figure 8.25 (f), where the best exchange that removes (s_1,s_18) yields an objective function value that is worse than that of the original solution x: the algorithm stops and returns the best solution found in its run.

Implementation details In order to avoid destroying the already performed moves, the second arc deleted each time must belong to the starting solution, as mentioned above. This implies an upper bound on the length of the sequence. It's provable that stoppping the sequence as soon as the exchanges no longer improve the starting solution does not impair the result: the overall

variation of the objective function is the sum of the variations due to the single exchanges

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

where each 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 $o_1, \cdots, o_k$ such that it's advantageous at each step.

Iterated greedy methods (destroy and repair)

The **iterated greedy methods**, also called destroy and repair or ruin and recreate, is based on the idea that general exchange heuristics consist in adding a subset of elements and removing a subset of elements, so when the algorithm passes from x to x' some new elements are introduced and some old elements are removed.

So, generally, visiting the neighbourhood exhaustively, one chooses A and D of small dimension, otherwise a lot of exploration must be made. The point is that if there are small variations, for example one element inside and one element outisde, the result could be easily unfeasible: an example was the transfer neighbourhood in the BPP. Enlargin the neighbourhood to exchanges of more elements can be inefficient and in many problems A and D can have different cardinalities, since the cardinality of the solutions is nonuniform (e.g. KP, SCP...).

The alternative to enlarging it is deleting a subset $D \subset x$ of cardinality less or equal to k and completing it with a constructive heuristic and adding a set $A \subset B \setminus x$ of cardinality less or equal to k and reducing x with a destructive heuristic.

Destroy-and-repair methods The complexity of an exhaustive exploration is $\mathcal{O}(n^{|A|}n^{|D|}\gamma(n))$, where

- the number of possible subsets to add is $\mathcal{O}(n^{|A|})$
- the number of possible subsets to remove is $\mathcal{O}(n^{|D|})$
- $\gamma(n)$ is the complexity of evaluating feasibility and the objective function

and, by contrast, deleting every subset D and running a constructive heuristic costs $\mathcal{O}(n^{|D|}T_{ch}(n))$ (where T_{ch} stands for the time complexity of the constructive heuristic); adding any subset A and running a destructive heuristic only costs $\mathcal{O}(n^{|A|}T_{dh}(n))$. Furthermore, in practice the complexity is further reduced limiting D or A to be made of elements of high or low cost and applying the first-best strategy instead of the global-best strategy.

CHAPTER 9

Exchange Metaheuristics

9.1 Introduction to Exchange Metaheuristics

After discussing the basic scheme of exchange heuristic, the *steepest descent* algorithm, whose elements were the starting solution, the definition of neighbourhood and the acceptance or termination condition. To extend this scheme, which find only a local optima, there are two possible ways, without abandoning completely the scheme of exchange heuristic: the first possibility is to repeat the search, starting from another point applying again the scheme of steepest descent - how to avoid following the same path? The second possibility is to extend the search, that is instead of restarting it, the algorithm goes on by accepting new solution that is in the local optima's neighbourhood, which requires to allow worsening of the objective function, and there's a strong risk that the algorithm "falls" back into the first local optimum.

9.1.1 Overcoming local optima

Previously, to build constructive metaheuristics the only possibility was to repeat the search. Exchange metaheuristics can also *extend* it, which isn't possible for constructive metaheuristics as increasing a set, which is the basic idea of any constructive heuristic, at a certain point can't be done anymore as it reaches is maximum size.

Nontheless, the mechanisms used to perform metaheuristics are again randomization and memory, which can be combined and applied to the basic elements of the algorithm, that is the starting solution $x^{(0)}$ (multi-start, ILS, VNS), the neighbourhood (VND) and the selection criterion $\varphi(x, A, D)$ (DLS or GLS, SA, TS).

Termination condition

Before discussing the techniques, it's necessary to address some issues regarding the fact that repeating or prolonging the search removes the intrinsic termination condition - this is something previously seen during the discussion of constructive metaheuristics. There are two possible ways to resolve this problem and the first is to use an "absolute" termination condition, which depends on something that's fixed a priori at the beginning of the search:

- a given total number of exploration of the neighbourhood (i.e., loops in the basic scheme) or a given total number of repetitions of the local search, that is how many times the basic overall scheme is applied, how many different starting solutions are given and taken to their respective local optimum

- a given total execution time, typically used when the time is limited; the problem with this is that running several times the same algorithm with the same parameters, the result will vary - even if the algorithm is "randomized", as it's in reality deterministic as well for a fixed seed; given the time to operate 1000 operations, the real machine may perform one time 1002, one time 999 and so on, yielding different results
- a given value of the objective
- a given improvement of the objective with respect to the starting solution

or a "relative" termination condition, which is *triggered* by some event that happens during the search, so it's not perfectly controlled by the user of the algorithm:

- a given number of exploration of the neighbourhood or repetitions after the last improvement of f^*: for example, after thousands of iterations without improvement the algorithm one may decide that no better solutions will be found or there aren't any in sight, so the computation will take too long
- a given execution time after the last improvement
- a given minimum value of the ratio between improvement of the objective and number of exploration or execution time

It's important to remember that fair comparisons require absolute consitions.

9.2 Repeat the search

How to generate different starting solutions in order to obtain different results when repeating the search? There are three possible main strategies: the first is to generate them at random, the second is applying different constructive heuristics (assuming that the initial solution is found by the use of a constructive heuristic) and the third is modifying solutions generated by the exchange algorithm, that is modifying the neighbourhoods.

9.2.1 Random generation

The first solution, the use of random generation, has some advantages: first of all, it's quite simple conceptually, which is an advangate in general; it's usually a quick procedure for problems in which it is easy to guarantee feasibility, which strongly depends on the single problem. Third, the user has control on the probability distribution and some elements could be favoured with respect to other ones: for example, the generation may favour low-cost elements, which is something similar to constructive heuristic with a random component, or elements that during the past search were frequent, intensifying the search, or those that weren't frequent at all, diversifying the search. Last, in infinite time random generation guarantees asymptotic convergence to the optimum.

Of course, there are some disadvantages as well: the starting solution which is generated is typically of very scarse quality, even though this doesn't necessarily means that the final solutions will be bad as well. But to be able to reach a good solution from a very bad one, a long time is often required, so t_{max} tends to be higher; this of course depends on the complexity of the exchange algorithm. Another disadvantage is that the feasibility decision may be $\mathcal{NP}$-complete, such as in the case for TSP in non complete graphs.

9.2.2 Using different constructive heuristics: multi-start methods

The multi-start methods were partly introduced as constructive metaheuristics, where different constructive heuristics were applied, yielding the best result obtained by all the different applications. Now, many solutions could be generated by constructive heuristics and then each of them can be improved with the use of exchange heuristics. The advantage is that one can choose how to build the starting solution, but there are several disadvantages: the first is that the user doesn't constrol what kind of solution are yielded; if similar ideas are applied in the generation of solution, probably similar solution will be generated, but this is not known a priori. A second big problem is that the algorithm cannot go on forever, so it's impossible to proceed indefinitely as the number of repetitions is fixed; in other words, using a fixed number of constructive heuristics automatically generates a number of runs of the exchange algorithm and in the end it will stop, so there is an automatic termination condition. This may be bad if one wants to test the dependency of the algorithm on the computational time, because for times longer than the maximum total time of all the heuristics the algorithm isn't going to improve. This also requires a high design effort, as several different algorithms must be designed and, furthermore, there is no guarantee of convergence, not even in infinite time.

What is the difference between a metaheuristic in which there is a randomized or memory based constructive phase and an exchange heuristic, or a metaheuristics in which there is a exchange heuristic initialized by some randomized process? The answer is that the difference is very small, so what was presented in the constructive metaheuristics section was a basic scheme in which only the constructive phase was considered and in this section only the exchange phase will be considered, just mentioning the fact that there can be a very refined constructive phase to initialize the search; the methods can be hybrid, and their "taxonomy" lies on the how refined is each phase: an algorithm with an highly refined constructive phase will be considered a constructive metaheuristic and so on.

Influence of the starting solution on the algorithm

Should one choose a random algorithm to generate the solution or a constructive heuristic to initialize the search? In general, if there are very good exchange heuristics, then it may take some more steps but in the end the starting solution has a short lived influence, so a random or heuristic generation of $x^{(0)}$ are very similar. On the other hand, if the exchange heuristic is not that good, the starting solution strongly influences the search and a good heuristic to generate $x^{(0)}$ is useful.

9.2.3 Exploit previous solutions

The third and last possibility is to exploit the information gathered by previously visited solutions, saving *reference solutions* such as the best local optimum found so far and possibly other local optima, generating new starting solutions modifying the reference ones.

By these choices, the user gains a good deal of control on the result, that is on the starting solution for the following phase: the user can control the starting point and how much it is modified, so the search can be intensified or diversified with respect to it. Typically, if the modification is not too large and the reference solutions are good, the starting solution is very good! The method is simple conceptually, as the user doesn't have to invent new construction methods but only has to invent a modification of the starting solution: such modifications should be easy as the problem should be already studied, since an exchange heuristic should already be present. Also, the implementation is simple, as the modification can be performed with the operations

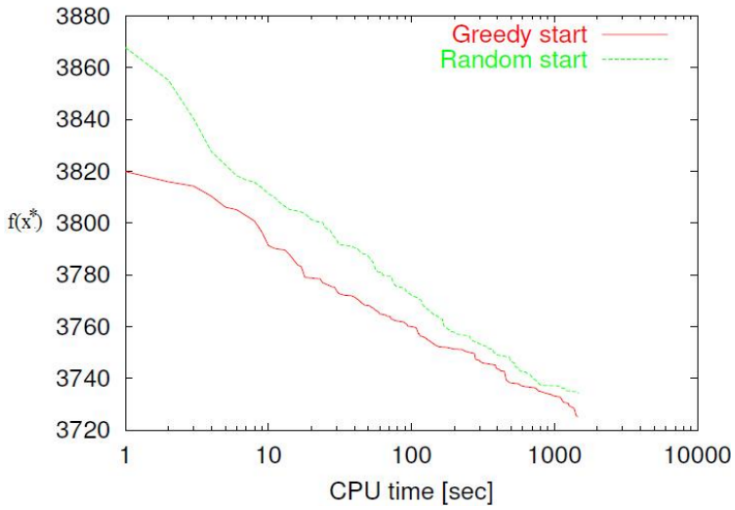


Figure 9.1: Greedy vs Random start for a bad exchange heuristic.

that define the neighbourhood. Finally, this method guarantees asymptotic convergence to the optimum in infinite time under suitable conditions, that is under the assumption that the user can control the distance of the new solution from a given one.

Two methods are here proposed: the *iterated local search* and *variable neighbourhood search*. These two methods are just *one* method with minor differences between them, however there's a huge deal of literature regarding them and the authors of ILS make a lot of effort to distinguish their method from the other ones.

Iterated Local Search

This methods iteratively applies local search, that is the classical steepest descent exchange heuristic, by generating new starting solutions every time - so, this metaheuristic, includes in itself a "normal" heuristic in its core. Then, the algorithm uses a *perturbation procedure* to generate starting solution from the reference solution(s). Finally, an acceptance condition is needed in order to decide whether to change the reference solution x or to reject it, going back to the original reference solution. Of course, also a termination condition is needed, but that's been already explained.

The Algorithm 14 starts, as in basic exchange heuristics, from a feasible solution $x^{(0)}$ then improving it with steepest descent: this would be all for the basic exchange algorithm, but in this case the result is saved and for a given number of iteration, assuming an absolute termination condition based on the number of repetitions of the basic scheme which can be replaced with time-based condition and so on, a *perturbation* is operated on the current solution x obtaining a new solution x', which is in turn improved again, finding an new local optimum which, in Algorithm 14, is still referred to as x' which is then compared to the overall optimum, in orded to decide if it's accepted or not. If it's accepted, the current reference solution becomes x' and then the two objective values, the one of the newly found local optima x' and the one of the overall best solution found until now, are compared to see if x' is also an improvement on the overall best solution.

So, the idea is that the exchange heuristic quckly explores an attraction basin, terminating into a local optimum; the perturbation procedure moves the starting solution to another attraction basin and the acceptance condition evaluates if the new local optimum is a promising starting point for the following perturbation, as represented in Figure 9.2.

Algorithm 14 Iterated Local Search Pseudocode

```
1:  procedure ILS(I, x^(0))
2:      x := SteepestDescent(x^(0))
3:      x* := x;
4:      for l := 1 to ℓ do
5:          x' := Perturbate(x)
6:          x' := SteepestDescent(x')
7:          if Accept(x', x*) then
8:              x := x'
9:          end if
10:         if f(x') < f(x*) then
11:             x* := x'
12:         end if
13:     end for
14:     return (x*, f(x*))
15: end procedure
```

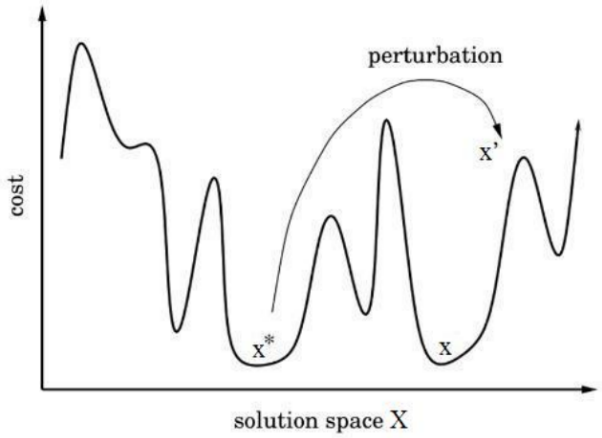


Figure 9.2: Schematic representation of the Iterated Local Search algorithm.

Example: ILS for the TSP A classical application of ILS to the TSP, used here to give some insight on the perturbation procedure, uses as exchange heuristic the steepest descent with neighbourhood N_{R_2} or N_{R_3} and as a perturbation procedure a *double-bridge* move that is a particular kind of 4-exchange, so it's in a "different neighbourhood" with respect to N_{R_2} or N_{R_3}. The double-bridge neighbourhood consists in fixing 4 edges and replace each one with another edge with a given rule, which is pictorially represented in Figure 9.3. It doesn't matter if it's

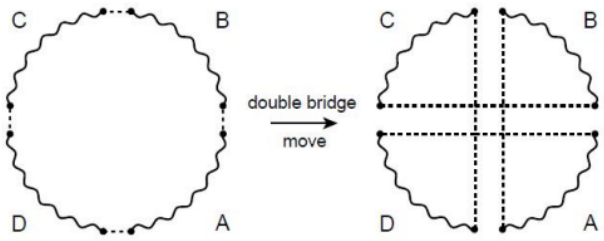


Figure 9.3: Representation of a double-bridge move.

better or worse: due to the nature of the move it's certainly feasible and it's generally out of

the original neighbourhood. The computation goes on with the original neighbourhood until it concludes in a local optimum: it could be the same as before in unlucky cases, it could be worse or it could be better. In this case the acceptance condition is that of an improvement on the best known solution, so

$$f(x') < f(x^*)$$

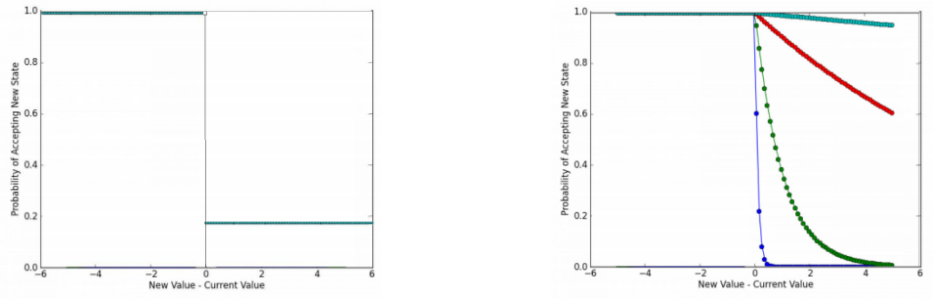
Perturbation procedure Let $\mathcal{O}$ be the operation set that defines the neighbourhood $N_{\mathcal{O}}$. The perturbation procedure performs a random operation o with $o \in \mathcal{O}' \not\subseteq \mathcal{O}$ to avoid that the exchange heuristic drives the solution x' back to the starting local optimum x. Two typical definitions of $\mathcal{O}'$ are:

- sequences of $k > 1$ operations of $\mathcal{O}$ if generating a random sequence is cheap
- conceptually different operations, for example vertex exchanges instead of arc exchanges

The main difficulty of ILS is in tuning the perturbation: if it's too strong it turns the search into a random restart, but if it's too weak it guides the search back to the starting optimum, wasting time and possibly losing the asymptotic convergence. Ideally, one would like to enter any basin and get out of any basin.

Acceptance condition In general, when the algorithms finds something that's better than the best known solution, it is wise to accept it. But if the new local optimum is worse than the given one, should the algorithm accept it or not? It depends on how the user thinks the balance between intensification and diversification should be tuned: accepting only improving solutions favours intensification, while accepting any solution favours diversification. In between these two options, there are lot of situations, which can be defined on the basis of $\delta f = f(x') - f(x)$, so that if $\delta f < 0$ the algorithm always accepts x' and otherwise the algorithms accepts x' with a certain probability $\pi(\delta f)$, which is a nonincreasing function. This is somewhat similar to the GRASP technique.

There are, of course, many schemes of distribution, but a simple one is better, such as a fixed probability: $\pi(\delta f) = \bar{\pi}(\delta f) \in (0;1)$ for each $\delta f \geq 0$, which is represented in Figure 9.4 (a); another case is that of monotonically decreasing probability with $\pi(0) = 1$ and $\lim_{\delta f \to +\infty} pi(\delta f) = 0$; there are different functions that are monotonically decreasing: exponential, linear and so on, parametrized by the slope, as represented in Figure 9.4 (b).



(a) Uniform probability. (b) Monotonically decreasing probability.

Figure 9.4: Different schemes of acceptance condition probabilities.

A third possibility is to use memory, accepting x' more easily if many iterations have passed by since the last improvement of x^*, in order to diversify the search.

ILS is defined on a very simple idea, perturbating each initial solution, on a sophisticated management of the acceptance condition of new local optima and has a problem on the definition of the perturbation technique.

Variable Neighbourhood Search

In a certain sense, VNS is complementary to ILS: it has a very simple acceptance condition, as it only intensifies, accepting only improving new local optima, but has a very refined mechanism to perturbate the solutions. VNS was proposed by Hansen and Mladenović in 1997 and other than having a strict acceptance condition, uses an **adaptive perturbation mechanism** instead of a fixed one. So, instead of having a single neighbourhood to generate new solutions, there is a hierarchy of neighbourhoods, that is a family of parametrically defined neighbourhoods with an increasing size k

$$N_1 \subset N_2 \subset \cdots \subset N_k \subset \cdots \subset N_{k_{max}}$$

Typically one uses the parametrised neighbourhood N_{H_k}, based on the Hamming distance between subsets or $N_{\mathcal{O}_k}$, based on the sequences of operations from a basic set $\mathcal{O}$, always extracting $x^{(0)}$ randomly from a neighbourhood of the hierarchy. This technique is called *variable neighbourhood* because the neighbourhood used to extract $x^{(0)}$ varias based on the results of the exchange heuristic: if a better solution is found, the algorithm uses the smalles neighbourhood to generate a starting solution very close to x^*, intensifying the search; if a worse solution is found, the algorithm uses a slightly larger neighbourhood to generate a starting solution slightly farther from x^*, diversifying the search.

The method has three parameters:

- k_{min} identifies the smallest neighbourhood to generate new solutions
- k_{max} identifies the largest neighbourhood to generate new solutions
- δk identifies the increase of k between two subsequent attempts

The exchange heuristic adopts the the smallest neighbourhood to be efficient, in general $N_k, k \leq k_{min}$

In the Algorithm 15, the *perturbation* phase is intrinsic in the extraction of the solution from the neighbourhoods.

Parameter tuning The value k_{min}, the minimum neighbourhood in which new starting solutions are generated, shouldn't be too small, as it would make it harder to exit current attraction basins, but it must not be too large, as to avoid "skipping" an attraction basin. In general one sets $k_{min} = 1$ and increases it if experimentally profitable. The value k_{max} must be large enough to reach any useful attraction basin and small enough to avoid reaching useless regions of the solution space. The reasonable value of δk must be large enough to reach k_{max} in a reasonable time but small enough to allow each reasonable value of k and typically $\delta k = 1$.

Skewed VNS Indeed, the creators noticed that diversification could be improved and proposed a variant known as *skewed VNS*, which replaces the basic acceptance criterion

$$f(x') < f(x^*) + \alpha \cdot d_H(x', x^*)$$

where $d_H(x', x^*)$ is the Hamming distance between x' and x^* and $\alpha > 0$ is a suitable parameter, which is really hard to tune, accepting then a worse solution provided that it is at a certain

Algorithm 15 Variable Neighbourhood Search Pseudocode

```
1:  procedure ILS(I, x^(0))
2:      x := SteepestDescent(x^(0))
3:      x* := x;
4:      k := k_min;
5:      for l := 1 to ℓ do
6:          x' := ExtractNeighbour(x*, k)          ▷ this is called "shaking" in VNS
7:          x' := SteepestDescent(x')
8:          if f(x') < f(x*) then
9:              x* := x'
10:             k := k_min
11:         else
12:             k := k + δk
13:         end if
14:         if k > k_max then
15:             k = k_min
16:         end if
17:     end for
18:     return (x*, f(x*))
19: end procedure
```

distance from the original solution: if $\alpha \approx 0$ tends to accept only improving solutions and if $\alpha >> 0$ tends to accept any solution. Of course, the random strategies for the ILS can also be adopted.

9.3 Extending the local search

After considering the idea of repeating the search, let's turn to the idea of prolonging the search and in particular doing so based on the basics selection step, modifying its elements. The basic selection step of steepest descent consists in evaluating for each solution in the neighbourhood the objective function choosing the solution which minimises it. This can be achieved in two different ways:

- using the *Variable Neighbourhood Descent* (VND) which **changes the neighbourhood** N; guarantees an evolution with no cycles (that is, the objective strictly improves) and terminates when all neighbourhoods have been exploited
- using the *Dynamic Local Search* (DLS) which **changes the objective function** f such that $\tilde{x}$ is better than x for the new objective function and possibly worse for the old one; this method can be "trapped" in loops as the objective changes over time, so it can proceed indefinitely.

9.3.1 Variable Neighbourhood Descent

As the name suggests, there is a similarity between **Variable Neighbourhood Descent** and the *Variable Neighbourhood Search* algorithm. In fact, VND was proposed by Hansen and Mladenović in the same papers where VNS was proposed and it's based on a similar idea: having

several different neighbourhoods. The big difference is that instead of using the neighbourhoods to restart the search, it's used to "go on" with the classical search.

Algorithm 16 Variable Neighbourhood Descent Pseudocode

```
1:  procedure VND(I, x^(0))
2:      flag_k := false ∀k;
3:      x̄ := x^(0);
4:      x* := x^(0);
5:      k := 1;
6:      while ∃k : flag_l = false do
7:          x̄ := SteepestDescent(x̄, k)
8:          if f(x̄) < f(x*) then
9:              x* := x̄
10:             flag_k' := false ∀k' ≠ k
11:         else
12:             flag_k := true
13:         end if
14:         k := Update(k)
15:     end while
16:     return (x*, f(x*))
17: end procedure
```

Given a sequence of neighbourhoods $N_1, \cdots, N_{k_max}$ (not necessarily a hierarchy), the algorithm starts with the first neighbourhood applying the classical steepest descent algorithm, finding a local optima with respect to the current neighbourhood. At this point, the current solution is probably a local optimum for many neighbourhoods but not for all of them, so the algorithms checks if the current solution is improved, and if it doesn't is the solution $\bar{x}$ is locally optimal also for the current neighbourhood, and this is signaled using a flag $flag_k$ relative to the neighbourhood k. If, instead, the solution is improved, the algorithm checks if this improvement holds also for every other neighbourhood; the algorithm stops when the current optimum is a local optima for every neighbourhood.

The fundamental differences between the VND and VNS are that in the VND at each step the current solution is the best known one, the neighbourhoods are explored, instead of used to extract random solutions (therefore they must bo not-so-large) and the chain of subsectioning is not necessary. Also, when a local optimum for each N_k has been reached the algorithm terminates, so VND is deterministic.

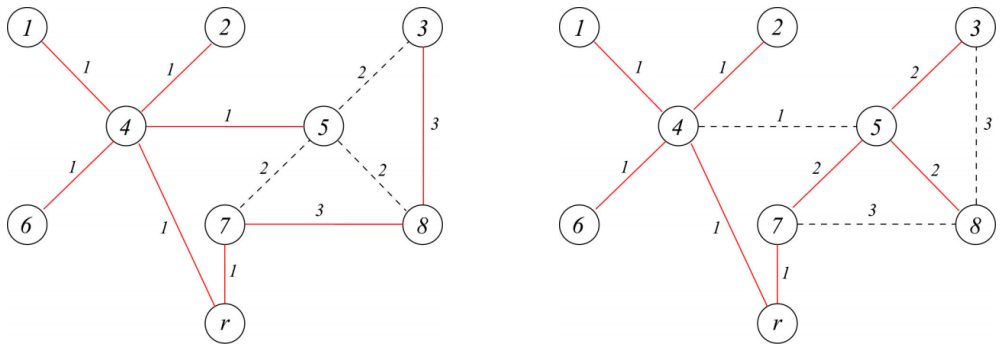
Neighbourhood scan strategies in the VND

There are two main classes of VND methods:

- methods with heterogeneous neighbourhood, based on the exploitation of the potential of topologically different neighbourhood, for example instead of exchanging vertices between subtrees of the CMST exchange edges; consequently, k periodically scans the values from 1 to k_{max} as the two neighbourhoods are intrinsically different and there's no good reason to think that a local optimum in one is a local optimum in the other
- methods with hierarchical neighbourhood, so $N_1 \subset \cdots \subset N_{k_{max}}$ which fully exploit the small and fast neighbourhoods, resorting to the large and slow ones only to get out of

the local optima. So, typically the scheme is a bit different from Algorithm 16, as the steepest descent will be operated on a solution for N_{S_1} and as soon as it reaches a local optima the algorithm will consider N_{S_2}, but it's not required to apply the steepest descent to termination, as is could be slow, but only some moves to guarantee that the algorithm gets far away from the original local optimum for N_{S_1}; once reached a distant solution the algorithm starts again with steepest descent for N_{S_1}. Another property of this method is that when the algorithm can't find an improvement in a neighbourhood N_k it must increase k and if it gets to the maximum possible value of k there's no sense in going back to the smaller ones, so instead of considering whether there's a $flag_k = false$, the algorithm "knows" that if a neighbourhood has been exploited so there's $flag_j = true$, then all $flag_u = true$ for $u < j$, as having a local optimum for certain values of N_k, that solution is also locally optimal for previous dimensions.

Example: the CMSTP Consider the instance of CMSTP where $n = 9$ vertices with uniform weights $w_v = 1$, capacity $W = 5$ and the cost reported in Figure 9.5, where missing edges have $c_e >> 3$. Consider the neighbourhood N_{S_1} for the first solution, Figure 9.5 (a): no edge in the



(a) Sollution 1. (b) Solution 2.

Figure 9.5: Two different solutions for the CMSTP

right subtree can be deleted because the left subtree has no residual capacity and deleting any edge in the left subtree increases the total cost, so the solution is a local optimum for N_{S_1}.

Consider, now, the neighbourhood N_{T_1} of transfers of single vertex from a subtree to another. In this case, moving vertex 5 to the right subtree yields the solution represented in Figure 9.5 (b), which improves the objective function.

9.3.2 Dynamic Local Search

The second family of methods is **Dynamic Local Search** or *Guided Local Search* Its approach is, in a sense, complementary to VND, as the object to be modified is not the neighbourhood but the objective function, which is typical of cases in which the objective function is not very informative, because it is *flat*, that is, it has wide plateaus. The idea is to introduce an additional penaly function $w : X \to \mathbb{N}$ defined on the solutions and usually with integer values and build an auxiliary function $\tilde{f}(f(x), w(x))$ which combines, somehow, the original objective value $f(x)$ and the penalty function $w(x)$, applying then the steepest descent algorithm to the combined function $\tilde{f}$; so, the steepest descent algorithm may take some step in a direction where f worse, as long as the combination with $w(x)$ (so $\tilde{f}$) makes it better. Furthermore, at each iteration the penalty w is updated based on the results obtained; anyhow, if these updates get stuck in a cycle, one may obtain always the same solutions: this situation is obviously to be avoided.

Algorithm 17 Dynamic Local Search Pseudocode

```
1:  procedure DLS(I, x^(0))
2:      x := SteepestDescent(x^(0))
3:      w := StartingPenalty(I)
4:      x* := x;
5:      while Stop() = false do
6:          (x̄, x_f) := SteepestDescent(x̄, f, w)
7:          if f(x_f) < f(x*) then
8:              x* := x_f
9:          end if
10:         w := UpdatePenalty(w, x̄, x*)
11:     end while
12:     return (x*, f(x*))
13: end procedure
```

The general idea, represented in Algorithm 17, is quite abstract as it can be developed and adapted in many different ways to different problems. The steepest descent algorithm optimises $\tilde{f}$ and returns two solutions: a final solution $\bar{x}$, locally optimal with respect to $\tilde{f}$ to update w and a solution x_f that is the best known with respect to f.

Variants

Let's discuss a bit more the concept of penalty, exploring some examples; two typical examples are **additive penalty**, where

$$\tilde{f}(x) = f(x) + \sum_{i \in x} w_i$$

having defined a penalty for each element of the ground set, which will be updated at each iteration, and a **multiplicative penalty**, where

$$\tilde{f}(x) = \sum_j w_j \phi_j(x)$$

which requires that $f(x) = \sum_j \phi(x)$.

The penalty can be modified with simple updates or with complex rules such as random updates as "noisy" perturbation of the costs, aiming to avoid cycles of operations and memory-based updates, favouring the most frequent elements for intensification or the less frequent ones for diversification.

The update can be performed at each single neighbourhood exploration, that is at each iteration, or when a local optimum for $\tilde{f}$ is reached or, finally, when the best known solution x^* is unchanged for a certain amount of time.

Additive Example: DLS for the MCP The first example is Dynamic Local Search for the Maximum Clique Problem, so given an instance of an undirected graph such as the one in Figure 9.6, the problem is to find a subsets of vertices that are reciprocally connected. For example, in Figure 9.6, vertices C and E are reciprocally connected, so they form a clique; the problem requires a clique of maximum cardinality. This simple case is quite complicated: what could be the right neighbourhood for this problem? The natural definition of neighbourhood is N_{A_1}, that is the neighbourhood built on vertex addition; most of the time, the newly added vertex is not

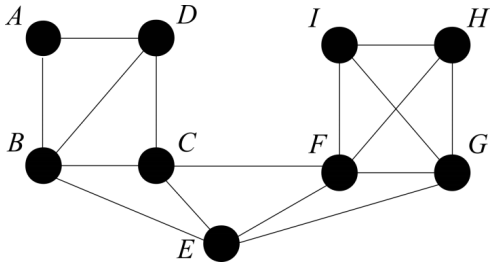


Figure 9.6: MCP instance.

adjacent to all the original ones and therefore is not a feasible new solution: it's quite difficult to find feasible elements, and even if there are feasible additions, in this case the objective function is flat (i.e. adding C or F to a solution $x = \{E\}$ is the same), which is typical of this neighbourhood. The wide plateaus are typical also of the swap neighbourhood N_{S_1}: consider $\{B,C,D\}$; C can be thrown away and A be added or D thrown away and E added: in both this two cases the change in the objective function is the same. How to move using neighbourhoods such as these ones?

Finding new neighbourhoods is not very easy, so one may think of changing the objective function, associating to each vertex i a penalty w_i initially equal to 0 and then the exchange heuristic will be applied to minimise the total penaly within the neighbourhood. In general, one should optimize the combined function, but in this case the objective function is going to be completely flat!

The penalty is updated when the exploration of N_{S_1} terminates: the penalty of the current clique vertices increases by 1 and after a given number of exploration, where all the nonzero penalties decrease by 1. The rationale of the method consists in aiming to expel the internal vertices for diversification and, in particular, the oldest internal vertices, as a form of memory usage.

Suppose that a starting solution is $x^{(0)} = \{B,C,D\}$, with $w = [011100000]$. Then, the algorithm performs these steps:

1. $w(\{B,C,E\}) = w(\{A,B,D\}) = 2$ but $\{A,B,D\}$ wins lexicographically, so $x^{(1)} = \{A,B,D\}$ with $w = [121200000]$.

2. $x^{(2)} = \{B,C,D\}$ with $w = [131300000]$ is the only neighbour.

3. $w(\{B,C,E\}) = 5 < 7 = w(\{A,B,D\})$, so $x^{(3)} = \{B,C,E\}$ with $w = [143310000]$

4. $w(\{C,E,F\}) = 4 < 10 = w(\{B,C,D\})$, so $x^{(4)} = \{C,E,F\}$ with $w = [144321000]$

5. $w(\{E,F,G\}) = 3 < 11 = w(\{B,C,E\})$, so $x^{(5)} = \{E,F,G\}$ with $w = [143310000]$

6. $w(\{F,G,H\}) = w(\{F,G,I\}) = 3 < 9 = w(\{C,E,F\})$, so $x^{(6)} = \{F,G,H\}$ with $w = [144333210]$

and now the neighbourhood N_{A_1} is not empty: $x^{(7)} = \{F,G,H,I\}$.

Multiplicative Example: DLS for the MAX-SAT Given m logical disjunction depending on n logical variables, find a truth assignment satisfying the maximum number of formulae. The neighbourhood N_{F_1} called *1-flip* is generated by complementing a variable. The DLS algorithm associates to each logical formula a penalty w_j initially equal to 1; in this case, the objective function is not only flat but it's also non-linear, as the problem works on a ground set B that

represents the truth assignments, but the effect of a precise assignment on the objective function is complicated because any number of formulae could be satisfied. So, in this case,

$$f(x) = \sum_j \phi_j(x)$$

where $\varphi_j(x)$ is 1 if the logical component j is satisfied and 0 if it's not.

The exchange heuristic maximizes the weight of statisfied formulae, thus modifying their number with the multiplicative penalty and then the penalty is updated

- when a local optimum is reached, increasing the weight of unsatisfied formulae to favour them
$$w_j := \alpha_{us} w_j \quad \forall j \in U(x)$$
with $\alpha_{us} > 1$, so that these will be the first to be satisfied, in some way, in the next iteration;
- after a certain number of updates or with a certain probability, reducing the penalty towards 1
$$w_j := (1-\rho)w_k + \rho \quad \forall j \in C$$
with $\rho \in (0,1)$.

The rationale of the method consists in aiming to satisfy the currently unsatisfied formulae as a form of diversification and, in particular, those which hav been unsatisfied for a longer time and more recently, as a form of memory usage. The parameters tune intensification and diversification: small values of α_{us} and ρ preserve the current penalty (intensification) and large values of α_{us} and ρ cancel the current penalty (diversification).

9.4 Modifying the minimization operation

After modifying the basic exchange heuristic algorithm by means of restarts, modifying the neighbourhood and/or modifying the objective function, the last possibility is to modify the last remaining element of the basic scheme, that is the minimization operation: in other words, while keeping the same neighbourhood defined for the basic exchange heuristic, objective function and initial solution, the algorithms that are going to be proposed modify the mechanism that determines the incumbent solution: instead of determining the minimum of the neighbourhood according to the objective function $f(x)$, other solutions may be accepted, possibly nonimproving (in local optima); the main problem is the risk of cyclically visiting the same solutions.

The two main strategies that allow to control this risk are

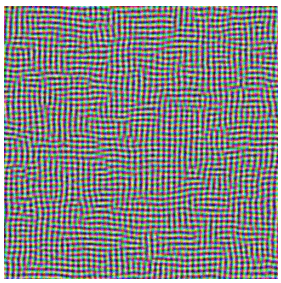
- Simulated Annealing (SA) which uses **randomization** to make repetitions unlikely;
- Tabu Search (TS) which uses **memory** to forbid repetitions.

9.4.1 Simulated Annealing

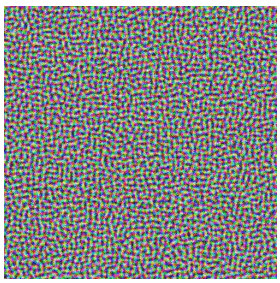
The **Simulated Annealing** algorithm has a long history and it was derived from an algorithm proposed by Metropolis in 1953, not to optimize problems but to simulate the behaviour of metals in the *annealing* process, which consists in taking a block of metal and heating it up to a temperature close to its fusion allowing the particles of metal not only to oscillate but also to move nearly freely in the volume of the metal and possibly redistribute nearly at random.

Then, the metal is cooled very slowly so that the energy of the metal decreases but converges to thermal equilibrium, so that there are no parts which have temperatures different from other parts. This means that in every situation the probability of every single particle of the metal of going in *any* direction should be, more or less, the same.

The physical, technological aim of the process is to guarantee that the particles reaching the thermal equilibrium distribute in a very even and regular way, obtaining a very regular defectless *crystal lattice*, that corresponds to the base state, that is the *minimum energy configuration* also obtaining a material with useful physical properties.



(a) Slow annealing.



(b) Fast annealing.

Figure 9.7: Different result of annealing procedures.

Quite soon, scholars realized that there were similarities between *continuous* optimization problems and simulation problems and then the similarity was "taken" to combinatorial optimization. The idea is that as the physical reality has different states, the problems have different solutions and each state of the physical reality is characterized by a value of the energy function, which "corresponds" to the objective function; the base state, that corresponds to the most desirable state, corresponds to the globally optimal solution. State transition, in physics, corresponds to the fact that particles moves and the state of the metal is not fixed but changes in time, therefore there's a transition from a given state to another state, which correspond to moves in combinatorial optimization given by local search. Finally, the overall process is ruled, in physics, by a numerical coefficient that is the temperature, which corresponds to a numerical parameter in the resolution to the combinatorial optimization problem.

The details of the simulation

According to thermodynamics, at the thermal equilibrium the probability of observing each state i depends on its energy E_i

$$\pi'_T(i) = \frac{e^{\frac{E_i}{k*T}}}{\sum_{j \in S} e^{\frac{-E_j}{k*T}}}$$

where S is the state set, T is the temperature and k is Boltzmann's constant. The numerator shows that when the energy of the state is large, its probability is close to zero, which means that having states of very large energy is very small; on the contrary, if the energy is very small the probability increases and, assuming that the energy is always postive, it's always smaller than 1 and overall at thermal equilibrium states with small energy are more likely than states with large energy.

Being Boltzmann's constant k, in fact, constant, the only "tunable" parameter is the temperature T, which plays an interesting role: if the temperature is very large, the exponent of the numerator goes down to zero, which means that the numerator is 1 and the normalization factor

is exactly the number of states, in other words all states are equiprobable. If the temperature is small, the role of the energy becomes very important.

The algorithm proposed by Metropolis generates a random sequence of states, each of which has an energy value which is given from the investigation of the problem and, starting from a state chosen in some way, the algorithm iteratively perturbs the state generating another state at random, which will have another energy value. The algorithm then chooses if it's better to move from the initial state to the perturbated one or not with probability

$$\pi_T(i,j) = \begin{cases} 1 & E_j < E_i \\ \exp(\frac{E_i - E_j}{k*T}) = \frac{\pi'(j)}{\pi'(i)} & E_j \geq E_i \end{cases}$$

the transition is deterministic if improving and probabilistic is worsening: the transition is accepted if the difference is very large (so the new state is very worsening) and with a possibly quite large probability if the transition is not so worsening (so the difference is very small). Simulated Annealing applies exactly the same principle.

The algorithm

Algorithm 18 Simulated Annealing Pseudocode

```
1:  procedure SA(I, x^(0))
2:      x := x^(0);
3:      x* := x^(0);
4:      T := T^[0];
5:      while Stop() = false do
6:          x' := RandomExtract(N, x)                ▷ random uniform extraction
7:          if f(x') < f(x*) or U[0,1] ≤ exp((f(x) − f(x'))/T) then
8:              x := x'
9:          end if
10:         if f(x') < f(x*) then
11:             x* := x'
12:         end if
13:         T := Update(T)
14:     end while
15:     return (x*, f(x*))
16: end procedure
```

where $U[0,1]$ represents the random uniform extraction of a number between $[0,1]$. As the neighbourhood is used to generate a solution and not fully explored, it is possible to worsen even if improving soultions exist; of course, this may have an impact on the quality of the obtained solution and, if improvements in the neighbourhood aren't "seen", the algorithm may perform many more steps to reach good solutions, so there's a tradeoff, as usual, between the efficiency of the overall process and the quality of the results. Since the exploration doesn't explore completely the neighbourhood, methods which depend on the order of exploration can't be applied. In order to make the computation faster, a table of possible values $\exp(\delta f/T)$ may be precomputed.

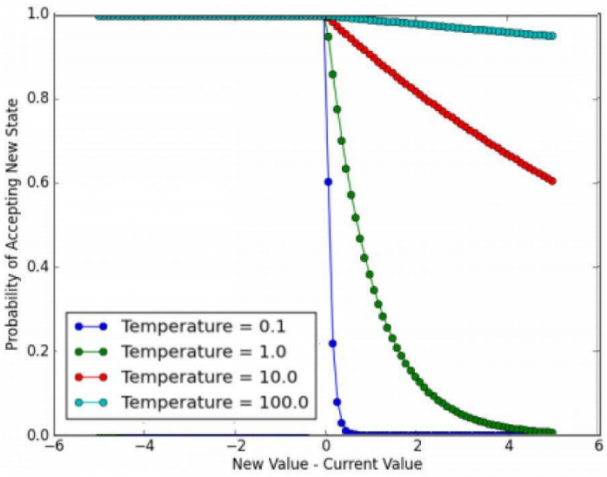


Figure 9.8: Probability distribution for different temperature values.

The acceptance criterion The profile of the probability of accepting is very similar to the one of Iterated Local Search, and actually the idea is exactly the same. In Figure 9.8 one can see that if the solution is improving it's always accepted ($\pi_T(x,x') = 1$) and if it's not improving the choice depends on the temperature T: if it's larger than 0 the choice is that of diversification, because nearly all solutions are accepted and in the extreme case it's a random walk; if $T \approx 0$ nearly all solutions are rejected and in the extreme case it's a steepest descent.

Asymptotic convergence to the optimum

There's a fundamental result that proves the asymptotic convergence of the SA to the optimum. While of low practical importance, as it's asymptotic convergence requires infinite time and combinatorial optimization problem have finite-time exact algorithms, it provides an insight on the tuning of the temperature. The idea is that since the algorithm performs random steps, the current solution x is a random variable and not deterministic. Its *state probability* $\pi'(x)$ in a given step t combines on all possible predecessors $x^{(t-1)}$: the state probability $\pi'(x^{(t-1)})$ of the predecessor, the probability to choose the move from $x^{(t-1)}$ to x, which is uniform as x is chosen at random and finally the probability to accept the move, that is

$$\pi_T(x^{(t-1)},x) = \begin{cases} 1 & f(x) < f(x^{(t-1)}) \\ \exp\frac{(f(x^{(t-1)})-f(x)}{T} & f(x) \geq f(x^{(t-1)}) \end{cases}$$

As it depends only on the previous step, the solution is a **Markov chain** and for a fixed temperature T the transition probabilities are stationary, so it's a homogeneous Markov chain. If the search space for neighbourhood N is connected, the probability to reach is state is greater than 0, and so it's called an irreducible Markov chain. Under these assumptions, the state probability converges to a stationary distribution independent from the starting state: this has the immediate practical consequence that for any starting solution, the probability of being in a given current solution after a long time tends to assume a stationary distribution that doesn't change in time and doesn't depend on the starting solution. The stationary distribution favours "good" solutions with the same law imposed by thermodynamics on physical systems at thermal equilibruim:

$$\pi_T(x) = \frac{\exp(\frac{-f(x)}{T})}{\sum_{x \in X} \exp(\frac{-f(x)}{T})} \quad \forall x \in X$$

where X is the feasible region and T the temperature parameter. The distribution converges to a limit distribution as $T \to 0$:

$$\pi(x) = \lim_{T \to 0} \pi_T(x) = \begin{cases} \frac{1}{|X|} & x \in X^* \\ 0 & x \in X \setminus X^* \end{cases}$$

which corresponds to a certain convergence to a globally optimal solution. This results, however, holds at the equilibrium, that is in infinite time: this has a big impact in practice as one can't choose an arbitrarily small T and other than the numerical problems with the representation of real numbers on a computer, very low values of T imply that a high probability to visit a gloval optimum, but also a slow convergence to the optimum (many exchanges are rejected) and the result may be that, in a finite time, the result obtained with low T is far from optimal.

What's is done is to use a dynamic profile of temperature in the SA algorithms, simulating the annealing process itself. Starting from a high temperature value, which increase diversification at the starting point, decreasing it iteratively. The starting value $T^{[0]}$ should be high enough to allow to reach any solution quickly and small enough to discourage visiting very bad solutions. A classical tuning for $T^{[0]}$ is to sample the first neighbourhood $N(x^{(0)})$ and set $T^{[0]}$ such as to accept a fraction α of the sampled solutions.

Updating the temperature The tempreature must be kept fixed for a long time and then decreased, iteratively, going through subsequent phases $r = 0, \cdots, m$ where eachc phase applies a constant value $T^{[r]}$ for $\ell^{[r]}$ iterations and $T^{[r]}$ decreases exponentially from phase to phase

$$T^{[r]} := \alpha T^{[r-1]} = \alpha^r T^{[0]}, \;\; \alpha \in (0,1)$$

and $\ell^{[r]}$ increases from phase to phase with values related to the diameter of the search graph, increasing the length of each phase; low temperature imply slow convergence to the optimum, which means that more iterations must be used - usually, the increase is linear.

The point is that since T is now variable, the Markov chain x is no longer homogeneous, so the convergence property might not hold. It has been proved that if the temperature increases slow "enough", the convergence is still guaranteed. Good parameters to tune the decrease depend on the instance, namely on $f(\tilde{x}) - f(x^*)$, where $f(\tilde{x})$ is the second-best value of f, but the best parameter values aren't known a priori.

Adaptive SA variants tune the temperature T based on the results: T is set to a value such that a given fraction of $N(x)$ is accepted and is increased if the solution has not improved for a certain time, otherwise is decreased.

9.4.2 Tabu Search

The **Tabu Search** (TS) has been proposed by Glover in 1986 and it's a very common and powerful heuristic. The idea is to keep exactly the same neighbourhood, exactly the same function and to keep the minimization criterion as well (that is, choosing the best solution in the neighbourhood), but to avoid going back to solutions that were visited previosly. This is done to avoid "attractions" due to local optimas found in previous rounds, which means tuning the neighbourhood in a sense, but tuning it based on the memory of solutions and not on costs or the results of the objective function.

This is a simple idea, but how to manage the tabu efficienctly and effectively?

Exchange heuristics with tabu

The problem with efficiency is that the algorithm must manage the set of altready visited solutions. Applying an exchange heuristic with a tabu means that the feasibility of every subset in the neighbourhood is still evaluated: if the subset is feasible, the objective function is evaluated as well and if its value is good (the best so far) the algorithm evaluates the *tabu status* and it's possible that even if the solution is very good it's tabu, so it can't be chosen. Only in the end the algorithm selects the best feasible and not-tabu one. So, there are three conditions to check instead of two.

An elementary way to implement the evaluation of the tabu is saving the visited solutions in a suitable structure called **tabu list** and checking each explored solution making a query on the tabu list.

This elementary evaluation of the tabu is, however, very inefficient. The comparison of the solutions at step t requires $\mathcal{O}(t)$, which is reducible with hash tables or search trees; the number of solutions visited grows indefinitely over time and the memory occupation grows indefinitely over time.

The *Cancellation Sequence Method* and the *Reverse Elimination Method* tackle these problems, exploiting the fact that, in general, the solutions visited from a chain with small variations and few solutions visited are neighbourhood of the current one. These techniques are so sophisticated that would require a lot of time to be explained. What's exploited is that the solutions in the tabu list are similar to each other, in fact they're neighbours, so there's no need to actually save completely the solutions but to save the *variations* on them, that is the move that was performed to obtain it. This also has the nice consequence that moves cancel each other, for example consider the MDP. A solution contains point 1 and the exterior has points 2, 3. exchanging 1 and 2 yields a situation where 2 is in the solution and 1 is outside. Changing then 2 and 3 is the same as changing 1 and 3 at first.

There are other reasons for which it's not nice, usually, to adopt an explicit definition of the tabu list. The first was inefficiency, the second is the effect on the quality of the result, as sometimes forbidding the solutions can disconnect the search graph, creating impassable "iron curtains" that block the search. Moreover, it can slow down the exit from attraction basis, creating a "gradual filling" effect that slows down the search: in effect, going down to a local optimum and applying the general rule of minimization with the except of visited solutions means exiting the basin as slow as possible, getting the second best solution, the third best solution and so on. It would be much better to keep into account the fact that the local optimum was reached form a certain direction and the search must go straight away in the same direction, whitout never coming back: the worsening is stronger, but it would make it easier to reach the basin of attraction and go to another one. This two general problems give a suggestion on how to modify the basic tabu search scheme: the first problem asserts that the prohibition should not be permanent, as it could "cut" the graph. The second problem affirms that one should not only prohibit already visited solutions but also solutions that are in the same direction, so solutions similar to the visited ones should be prohibited. The two phenomena suggest apparently opposite remedies, how to combine them?

Example A very degenerate example is provided by the following problem. Let the ground set $B = \{1, \cdots, n\}$, so that it includes the first n natural numbers; all subsets are feasible, that is $X = 2^B$ and the objective combines a nearly uniform additive term $\phi_i = 1 + \varepsilon i$ with $0 < \varepsilon << 1$

and, if and only if $x = x^*$, a strong negative term

$$f(x) = \begin{cases} \sum_{i \in x}(1+\varepsilon i) & x \neq x^* \\ \sum_{i \in x}(1+\varepsilon i) - n - 1 & x = x^* \end{cases}$$

Using the neighbourhood of all solutions at Hamming distance ≤ 1

$$N_{H_1}(x) = \{x' \in 2^B : d_H(x,x') \leq 1\}$$

the problam has a local optimum $\bar{x} = \emptyset$ with $f(\bar{x}) = 0$ whose attraction basin includes the solutions x with $d_H(x,x^*) > 1$ and a global optimum x^* with $f(x^*) = n(n+1)\varepsilon/2 - 1 < 0$, whose attraction basin includes the solutions x with $d_H(x,x^*) \leq 1$.

Starting from $x^0 = \emptyset = \bar{x}$ and forbidding all the solutions visited, the algorithms visits methodically most of 2^B, with f and $d(x,\bar{x})$ going up and down; for $4 \leq n \leq 14$ the search graph is disconnected and the search is stuck, but all solutions are at least explored and, finally, for $n \geq 15$ the search is stuch and some unvisited solutions are not explored, possibly missing the optimum.

t	f	x	$d(x,\bar{x})$
1	0	0000	0
2	$1+\varepsilon$	1000	1
3	$2+3\varepsilon$	1100	2
4	$1+2\varepsilon$	0100	1
5	$2+5\varepsilon$	0110	2
6	$1+3\varepsilon$	0010	1
7	$2+4\varepsilon$	1010	2
8	$3+6\varepsilon$	1110	3
9	$4+10\varepsilon$	1111	4
10	$3+9\varepsilon$	0111	3
11	$2+7\varepsilon$	0011	2
12	$1+4\varepsilon$	0001	1
13	$2+5\varepsilon$	1001	2
14	$3+7\varepsilon$	1101	3
15	$2+6\varepsilon$	0101	2

Table 9.1: Execution of tabu for $n = 4$.

Table 9.1 is a numerical eample for tha case in which $n = 4$. At iteration $t = 1$, the algorithm chooses the incumbent solution to be 1000, which is the one with the minimum cost among those in the neighbourhood. From the latter, the algorithm would choose as best solution $x = \{0000\}$, but it's prohibited, so $x = \{1100\}$ is the chosen one. Then, items are removed and added as shown in table until the algorithm reaches a saturation point at $t = 15$, where no move can be done and not all solutions have been examinated, as $2^4 = 16 \neq 15$, so the graph is disconnected, but at a certain point it can be visited: at step $t = 7$ the missing combination (1011) is in the neighbourhood.

The objective function profile, shown in Figure 9.9, configmr the limitations of the method. The solution x repeatdly gets far from $x^{(0)} = \bar{x}$ and close to it: it visits nearly the whole attraction basin of $\bar{x}$ and, in the end, it does not get out of it, but gets stuck in a solution whose neighbourhood is fully tabu. Removing the oldest tabu, the exploration goes around and the risk of looping gets back.

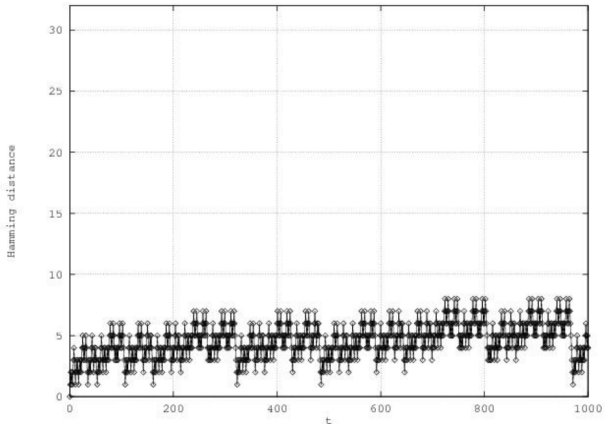


Figure 9.9: Objective function value for the previous example.

Attribute-based tabu

The idea of **attribute-based tabu** is to forbid solutions that share attributes with the ones that have already been visited. An *attribute* is a completely abstract concept and a set of attributes A must be defined; the algorithm will manage a subset of attributes $\bar{A}$ which will be empty at first. Each solution y has a subsets of attributes A_y that characterize it and, if a solution has an attribute which *is tabu*, then it's forbidden

$$y \text{ is tabu} \iff A_y \cap \bar{A} \neq \emptyset$$

after performing a move from the current solution x to x', the algorithms adds to $\bar{A}$ the attributes possessed by x and not by x':

$$\bar{A} := \bar{A} \cup (A_x \setminus A_{x'})$$

so that x becomes tabu. This allows to avoid solutions similar to the visited ones and get more quickly far away from the visited local optima; however, this increases the prohibition and the problem of disconnecting the search graph is not solved. To solve it, the prohibition must be made not permanent.

Temporary tabu and aspiration criterium The second point is, in fact, giving a limited value to the tenure of the tabu, that is a limited length L to the prohibition and old elements will be thrown away. Each element has a tabu length and when its length has expired, the item is removed from $\bar{A}$ and solutions wich have no more attributes in it can be chosen again. Going back to old solutions raises the probability of having loop in the execution, but going back to the old solutions with a different $\bar{A}$ will, one hopes, different solutions. Tuning the tabu tenure is fundamental for the effectiveness of the method.

Since the tabu could forbid a global optimum similar to a known solution, one may introduce an **aspiration criterion**: a tabu solution better than the best known one is anyway accepted; this, of course, doesn't bring any new looping risk. There are looser aspiration criteria, but they are not commonly used. Finally, if all neighbouring solutions are tabu, the algorithm may accept the one with the oldest tabu and this can be interpretes as another aspiration criterion.

From the Algorithm 19 it seems that managing a tabu list is in fact managing a list of attributes by introducing and removing elements, but that's not how it actually works.

Tabu attributes The concept of "attribute" is intentionally generic and the simpler ones are the inclusion of an element in the solution, so $A_x = x$: when the move from x to x' expels an

Algorithm 19 Tabu Search Pseudocode

```
1:  procedure TABUSEARCH(I, x^(0), L)
2:      x := x^(0);
3:      x* := x^(0);
4:      Ā := ∅;
5:      while Stop() = false do
6:          f' := +∞
7:          for y ∈ N(x) do
8:              if f(y) < f' then
9:                  if Tabu(y, Ā) = false or f(y) < f(x*) then x' := y f' := y
10:                 end if
11:             end if
12:         end for
13:         Ā := Update(Ā, x', L)
14:         if f(x') < f(x*) then
15:             x* := x'
16:         end if
17:     end while
18:     return (x*, f(x*))
19: end procedure
```

element i from the solution, the tabu forbids the reinsertion of i in the solution; x has the attribute "presence of i" and x' hasn't got it - it enters $\bar{A}$ and every solution including i becomes tabu. A second, easy, possibility for an attribute is to be the exclusion of an element from the solution, that is $A_x = B \setminus x$: when the move from x to x' inserts an element i into the solution, the tabu forbids the removal of i from the solution; x has the attribute "absence of i" and x' hasn't got it - the attribute "absence of i" enters A and every solution devoid of i becomes a tabu.

Different attributes sets can be combined, each with its tenure and list: for example, replacing i with j, forbidding the removal of j for L^{in} steps and to insert i for L^{out} steps, with $L^{in} \neq L^{out}$.

Other, less frequent attributes are the value of the objective function and the value of an auxiliary function. The first forbids solutions of a given value, previously assumed by the objective.

Complex attributes can be obtained combining simple attributes: the coexistence in the solution of two elements or their separation or, if a move replaces element i with element j, the tabu can forbid the removal of j to include i but allow the simple removal of j and the simple inclusion of i.

Efficient evaluation of the tabu status

The evaluation of the tabu status must be efficient and must avoid scanning the whole solution. In this direction, the attributes may be associated to moves and not to solutions; in other words, if the attribute is the inclusion of an element in the solution, to check if an element is in the solution or not one must not consider the solution that is obtained but the move that's being operated. In order to make the check constant, a data structure is needed such that it associates to each attribute an information on its tabu status, that is when it became a tabu.

If the tabu is on insertions, at iteration t the algorithm evaluates all moves forbidding to add

$i \in B \setminus x$ when $t \leq T_i^{in} + L^{in}$ and updates the structure, setting $T_i^{in} := t$ for each i removed, that is $i \in x \setminus x'$.

If the tabu is on deletions, at iteration t the algorithm evaluates all moves forbidding to delete $i \in x$ when $t \leq T_i^{out} + L^{out}$ and updates the structure, setting $T_i^{out} := t$ for each i added, that is $i \in x' \setminus x$.

A single vector is enough for both, given that either $i \in x$ or $i \in B \setminus x$. More sophisticated attributes require more complex structures.

Example: the TSP Consider the neighbourhood N_{R_2} generated by 2-opt exchanges and use as attributes both the presence and the absence of arcs in the solution, hence adapting the tabu on both the insertions of elements that are currently out and removal of arcs that are currently in the solution.

So, instead of a vector, a matrix can be used - this is also because arcs are naturally defined by a node-by-node matrix, that is the adjacency matrix of the graph - and each elements T_{ij} of the matrix is set, initially, to $-\infty$ for each arc $(i, j) \in A$, so that each element is non-tabu. are not verified. At each step t, the algorithm explores $n(n-1)/2$ pairs of removable arcs and the corresponding pairs of arcs which would replace them and the move (i, j), which replaces (s_i, s_{i+1}) and (s_j, s_{j+1}) with (s_i, s_j) and (s_{i+1}, s_{j+1}) is tabu if one of the following conditions holds:

$$t \leq T_{s_i,s_{i+1}} + L^{out}$$
$$t \leq T_{s_j,s_{j+1}} + L^{out}$$
$$t \leq T_{s_i,s_j} + L^{in}$$
$$t \leq T_{s_{j+1},s_{i+1}} + L^{in}$$

and at first all moves are legal. Once the move (i^*, j^*) is selected, the algorithm updates the auxiliary structures, setting

$$T_{s_{j^*},s_{i^*+1}} := t$$
$$T_{s_{j^*},s_{j^*+1}} := t$$
$$T_{s_{i^*},s_{j^*}} := t$$
$$T_{s_{j^*+1},s_{i^*+1}} := t$$

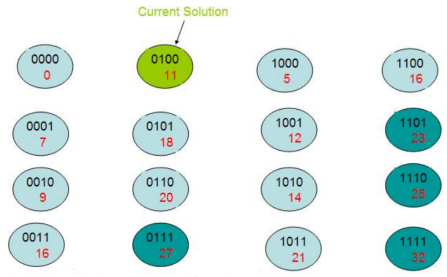
An interesting remark is that L^{in} and L^{out} must be tuned and it is better to set $L^{out} << L^{in}$ when the solution is small with respect to the ground set, which is often the case.

Example: the Max-SAT Consider the neighbourhood N_{F_1} which includes the solutions obtained complementing the value of a variable and all n solutions are feasible. Since $|x| = |B \setminus x|$ for each $x \in X$, as the elements of the ground set are the truth assignment for each variable and each variable must have exactly one (true or false), the classical combined insertion and deletion tabu can be used with the same tenure value for both and it is sufficient to forbid the change of value of a variable and the attribute is variable.

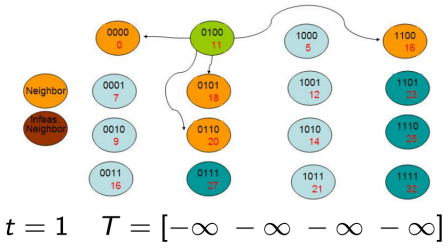
The algorithm proceeds as follows:

- at first set $T_i = -\infty$ for each variable $i = 1, \cdots, n$
- at each step t explore the n solutions obtained complementing each variable
- the move i, which assigns $x_i := \bar{x_i}$, is tabu at step t if $t \leq T_i + L$ and at first all moves are non-tabu
- perform move i^* and set $T_{i^*} := t$

Example: the KP Consider the neighbourhood N_{H_1} which includes all solutions at Hamming distance ≤ 1, so it includes only additions and deletions of single elements. For the sake of simplicity let's use the variable t as an attribute (indicating the last move performed), with $L = 3$; in other words, a vector T saves the iteration at which the last move was performed on each $i \in B$.

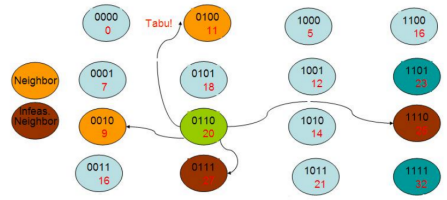


(a) Initial state.



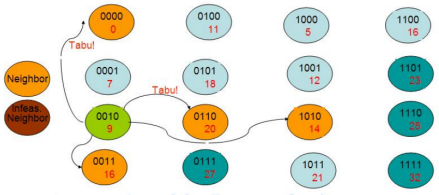
$t = 1 \quad T = [-\infty \ -\infty \ -\infty \ -\infty]$

(b) Neighbouring solutions.



$t = 2 \quad T = [-\infty \ -\infty \quad 1 \ -\infty]$

(c) Second iteration.



$t = 3 \quad T = [-\infty \quad 2 \quad 1 \ -\infty]$

(d) Third iteration.

Figure 9.10: Example of tabu on KP.

Suppose to start in the state represented in Figure 9.10 (a) and there are 4 neighbouring solutions, which are represented in Figure 9.10 (b): each one of them is feasible and non-tabu, and the best one to chose is 0110, which is chosen and $T[2] = 1$ is set to 1. Let's say that the dark brown neighbours are unfeasible (and all dark blue are as well): at step $t = 2$, the neighbouring solutions are half feasible and half unfeasible; the feasible solution 0100, which is the starting one, is tabu so it can't be chosen and the algorithm is forced to chose 0010, even though it's worsening. At the third iterations, all neighbouring solutions are feasible but two are taboo, in particular one is the last visited solution and the other is tabu as $t = 3 \leq T_3 + L = 1 + 3 = 4$

Tuning the tabu tenure

The value of the tabu tenure L is a crucial parameter: if it's too large, it could conceal the global optimum by forbidding moves and disconnecting the graph. If the tabu tenure is too small, it can hold the exploration back in useless regions and, in the worst case, it can produce cyclic behaviours, which can be detected by plotting the values of the solutions and find a periodic behaviour.

Some general rules can be given about its value: in general, the value of the tenure grows as the instance grows, but it doesn't grow linearly with the size and Glover suggests that $L \in \mathcal{O}(\sqrt{|A|})$ where A is the number of attributes; usually, if the size of the instance doesn't change very much it's possible to keep a constant value of the tenure without big problems. Another trick to break the loops is extracting the value of the tenure at random in a given range, so going back to the origin of the loop it's unlikely to use the same value for the tenure for all repetitions.

Another common mechanism that's often adopted is to make the tabu tenure vary, taking into account improvements, meaning that the search is reaching a local optimum, making L decrease

favouring intensification, or when the curretn solution worsen, making L increase favouring diversification.

Tabu search variants

Other adaptive strategies work in the long term: the reactive Tabu Search varian uses efficient structures to save the solutions visited, which has the objective to detect cyclic behaviours; once a cycle is detected, the range $[L_{min}, L_{max}]$ is moved upwards. A very similar idea is the so called frequency-based Tabu Search, in which instead of saving the solution the algorithm saves the frequency of the attributes, so there's a vector that instead of saving the last iteration in which an attributes changed its status saves the frequency with which that attribute was occurring in the solution and, if the frequency is too large, one can decide to diversify and make that move tabu or even work on the objective function as in the DLS. Of course, the search could also intensify, favouring the moves introducing the frequent attribute.

Another idea, that looks more like VNS is the concept of exploring Tabu Search, in which if the search is not "going well", that is solutions haven't improved in a long time, the search can reinitialized from a solution that was explored previously but not chosen (i.e. the "second-best solutions") - a pool is choosen on some basis (best ones, most diverse ones and so on) and then the search is restarted from one of these solutions.

A final idea is an extension of the fine neighbourhood tuning techniques, where the idea, based on the concept of first working in a reduced neighbourhood and then enlarging the neighbourhood has times go by so to slow down the search but making it more precise and careful.

Part V

Solution Based Metaheuristics: Recombination Metaheuristics

CHAPTER 10

Recombination Metaheuristics

The last class of heuristics to be examined are the **recombination metaheuristics**. They are called metaheuristics because they often adopt memory or random steps as methods to proceed and there's no basic scheme that is later on extend with additional devices.

10.1 Introduction to recombination metaheuristics

Recombination heuristics are based on the idea of **handling many solution at a time**; both constructive and exchange heuristics manage one solution at a time, except for *Ant System*. In recombination heuristics there is a set of solutions which is generally called *population of individuals*, due to the natural-inspired analogy that drives many of these algorithms and the main point is that the "individuals" are going to be recombined, mixing solutions generating new ones. Of course, recombination metaheuristics also adopt elements taken from the other heuristics, sometimes renamed. In the following, two main families of recombination heuristics will be used: the first category of recombination metaheuristics has a rather deterministic structure and adopt random steps only as a last resort and it's not intrinsic in their definition (so they could be classified as recombination heuristics) and they mainly derive from the field of heuristics algorithm of scholars interested in optimization. The main examples of this family are *Scatter search* and *Path relinking*.

The second family is based on a strong use of randomization, which sometimes derives from a biological metaphor which genetic or evolutionistic mechanisms or, in general, the behaviour of living things: in this case, the randomicity tends to simulate the behaviour of these agents. Examples are *genetic algorithms*, *memetic algorithms* and *evolutionistic strategies*.

10.1.1 The general scheme

The general scheme of recombination heuristics is based on the idea that different solutions can be partly optimal, where "partly" means that there are some components of these solutions that are exactly identical to the corrisponding components in the optimal solution. A *component* is something very vague in general: it could be a circuit in the VRP, a truth assignment for a subset of variables in a MAX-SAT and so on; a component is anything that's part of the solution which can have a good or bad quality.

Instead of having a single solution and trying to correct the "bad" components, having many solutions from which components could be extracted and combined in a final solution, one may be able to identify the good parts from different solutions and obtain, possibly, the optimal solution, which would be much more efficient than just modify each single solution pushing it

towards the optimal one. So, the general scheme of recombination heuristics consists in building a starting population of solutions which, exactly as for the exchange heuristics, can be obtained randomly, using contrustive meta/heuristics and, in this case, also using exchange heuristics. Then, as long as the solutions doesn't meet a certain parametric condition, the algorithm goes on iteration after iteration, generating new populations of solution. Each subsequent population is called **generation**. For each generation the basic scheme extracts a collection of subsets of individuals of the populations which are made out of single individuals or groups and to each individual is applied some modification (exchange operations) which implies the production of new individuals. Another possibility, which is a characteristic of recombination heuristics, is the application of recombination operations to the subsets of more than one individual producing new individuals (often, these techniques are the ones that give the name to the overall algorithm). Once all the generated individuals are collected, the algorithm chooses, in some way, if each one is worthy of being included in the next generation or not, having a *selection rule*: in some heuristics, several copies of the same solutions may be generated, whereas in other cases this is absolutely forbidden; usally, deterministic recombination heuristics avoid generating multiple copies, as it is a waste of space and time, whereas the most simulation-oriented algorithms allow this mechanisms.

10.2 Scatter Search

Scatter Search (SS) was proposed by Glover, the creator of Tabu Search and many other optimization methods, in 1977 and the general idea is, as in all recombination heuristics, starting from a initial population of solutions which are then improved with an exchange procedure, which is not typical of all recombination heuristics. Scatter search and, in general, the family of deterministic recombination heuristics tend to work on very good, which are then improved with an exchange procedure, yielding a population of locally optimal solutions. Then, a **reference set** $R = B \cup D$ is built, where B includes the best known solutions and D includes the "farthest" solutions from B and each other (this requires a distance definition and an example is the Hamming distance). At this point, in order to build the reference set, the algorithm takes each pair of solutions $(x,y) \in B \times (B \cup D)$ (at the first step they're extracted from the population) which are recombined in some way, generating a new solution z, which is improved with the exchange procedure, obtaining z', which is tested on whether or not it can enter the subset of best solutions: if $z' \notin B$ and B contains a worse solution, that solution is replaced with z';. In other words, one just tries to put z' in B possibly displacing the worst solution in the set. If z' is worse than all of them, z' is discarded. If z' is discarded, one tries to insert it in the most diverse subset D, meaning that the algorithm tests the distance of z' from all the solutions in B and from all the solutions in D except the one that has the smallest total distance from B and from D (so to make it comparable); if the distance of z' is larger than that of the element of the smallest distance appartaining to D, the latter is replaced with z'. No duplicates must figure in either B or D.

However, not *every* pair of solutions $(x,y) \in B \times (B \cup D)$ has been considered, because one of the two solutions is in the best solutions and the other is either in the best solutions subset or in the "farthest" solutions subset: this means that pairs of $D \times D$ aren't considered. The purpose of recombination is to build new starting points for the exchange heuristics and are admitted starting points that are combinations of best with best $B \times B$ as it can intensify the search, while $B \times D$ solutions can diversify the search without losing too much, producing something that's "partly" good and "partly" diverse. After the reference set has been updated, the algorithm goes back and restarts and for each pair of solutions in the new reference set it riapplies the

recombination, the update of the best subset and the update of the diverse subset: going on and on there's a termination condition which is when the reference subset R isn't modified, so all recombination haven't made it into the best or diverse subsets, meaning that the reference set isn't changed at all which in turn means that the following recombinations will give exactly the same thing, unless some randomization is involved, which isn't present in the basic SS scheme.

10.2.1 The algorithm

Algorithm 20 Scatter Search Pseudocode

```
1:  procedure SCATTERSEARCH(I, P)
2:      B := ∅
3:      D := ∅
4:      repeat
5:          Stop = true
6:          for each x ∈ P do
7:              z := SteepestDescent(I, x)
8:              y_B := arg max_{y∈B} f(y)
9:              y_D := arg min_{y∈D} d_H(y, B ∪ D \ {y})
10:             if f(z) < f(y_B) then
11:                 B := B \ {y_B} ∪ {z}
12:                 Stop := false
13:                 if f(z) < f(x*) then
14:                     x* := z
15:                 end if
16:             else
17:                 if f(z, B ∪ D \ {y_D}) > d(y_D, B ∪ D \ {y_D}) then
18:                     D := D \ {y_D} ∪ {z}
19:                     Stop := false
20:                 end if
21:             end if
22:         end forP := ∅
23:         for each (x, y) ∈ B × (B ∪ D) do
24:             P := P ∪ Recombine(x, y, I)
25:         end for
26:     until Stop = True
27:     return (x*, f(x*))
28: end procedure
```

Let's go over the steps in Algorithm 20. The algorithm starts with a given population P and at first, for each solution in the current population, steepest descent is applied so to find a local optimal solution z. Then, one needs to know which is the worst solution in the best subset B, that is the solution with maximum value of the objective function (for minimization problems) and one needs to know the less diverse solution in the diverse subset D, which is the solution that has minimum distance from the overall reference set (usually the distance is the Hamming distance, but other definitions can be used). Having obtained the worst solution in the best set and the worst solution in the diverse set, the algorithm takes the current local optima that is being tested checking if it's better than the worst among the best (that is, in B): it's the case, the

new solution z replaces y_B in the best subset, and the algorithm must not stop, as the reference set has changed. Otherwise, if z doesn't make it into the best subset, the algorithm checks it against the diverse subset and, if its distance $d(z, B \cup D \setminus y_D)$ is larger than that of the solution with the smallest distance y_D, it can be inserted in D and again the algorithm mustn't stop. In the end, the recombination of all the pairs $(x, y) \in B \times (B \cup D)$ are inserted into the population.

10.2.2 Recombination Procedure

The last thing to define regarding Scatter Search is the recombination procedure, as it depends, in practice, on the problem itself; however, it's usually based on the manipulation of the two solutions x and y considered as subsets (which is pretty much always the case, as these are solution-based problems). The idea is that if both x and y include an element, it's probable that such element is good, as it's included in both solution that are the base of the reasoning. Defining z as the new, recombined, solution, the first elements included in it are

$$z := x \cap y$$

then, the solution z is augmented adding elements from $x \setminus y$ or $y \setminus x$ which can be added at random, with a greedy selection criterion, in alternation from the two source (one from x then one from y and so on) or, finally, freely from the two sources.

It's also possible that z isn't feasible, so starting from a certain feasible point a constructive heuristic could be used, adding external elements from $B \setminus (x \cup y)$; if no feasible solutions are found in this way either, an auxiliary exchange heuristic can be applied to lead z back to feasibility: this possibility is called **repair procedure**.

Examples

MDP Given two solutions, which are subsets of k elements, the recombined solution z is initially defined as

$$z := x \cap y$$

and is augmented with $k - |z|$ random of greedy points from x and y and no repair procedure is required.

Max-SAT Given two solutions, which are subsets of n truth assignments, the recombined solution z is initially defined as

$$z := x \cap y$$

and is augmented with $n - |z|$ random or greedy truth assignments from x and y and no repair procedure is required.

KP Given two solutions, which are subsets of objects, the recombined solution z is initially defined as

$$z := x \cap y$$

and is augmented choosing randomly or greedily objects from x and y such that the capacity constraint is respected, and no repair procedure is required, but the solution could be augmented.

SCP Given two solutions, which are subset of columns, the recombined solution z is initially defined as

$$z := x \cap y$$

and is augmented choosing randomly or greedily objects from x and y avoiding redundant ones. External columns from $B \setminus (x \cup y)$ could be added as it's not guaranteed to have feasible solutions and, finally, some redundant columns could be removed with a destructive phase.

10.3 Path Relinking

The **Path Relinking** procedure, proposed again by Glover in 1989, is generally used as a final intensification procedure more than as a standalone method. The idea is to take pairs of solutions and try to combine them, but the combination is intrinsically different.

Given an exchange heuristic and a neighbourhood N, a set of *interesting* solutions is generated which is again called **reference set** R, which is based on the best solutions generated on the auxiliary heuristic (using, then, *elite solutions*). For each pair of solutions x and y in the reference set R, a path from x to y is built in the search space using the neighbourhood N applying to $z^{(0)} = x$ the auxiliary heuristic, but choosing at each step the solution closest to the destination y:

$$z^{(k+1)} := \arg \min_{z \in N(z^{(k)})} d(z, y)$$

where d is a suitable metric function on the solutions and in case of equal distance, the objective function f is optimized. Obtained a second solution $z^{(1)}$ the same is done, exactly as in local search, that is exploring the neighbourhood searching the closest one to the final solution. If the search graph is connected, there must be a path $x \rightsquigarrow y$; there are case in which this path can't be followed by decreasing the Hamming distance, but those are strange and *pathological* cases, so in let's say that in general that path exists and can be followed. Along the followed path, the best solution

$$z^*_{xy} := \arg\min_k f(z^{(k)})$$

is saved and if $z^*_{xy} \notin R$ and is better than those of R, it's added to R.

10.3.1 The algorithm

The general scheme of Path Relinking is the following in Algorithm 21. Given a population, until it's not empty the algorithm checks all the pairs of solution in the population starting from a solution x and saving the best solution along the path. Until the algorithm gets to the end of the path, that is y, the algorithm finds in the neighbourhood of the current solution z the one that has the smallest distance from y: in general, many will be found and inserted into a set Z. Among these, the best one is found and at the point the algorithm moves in that direction, until $z = y$. At every step the algorithm checks if a better solution is found, so to be able to return the best solution along the path. If the best solution along the path is not in the reference set, that is it's not already known, it's improved using steepest descent and it's inserted in the new candidate set, which replaces the population.

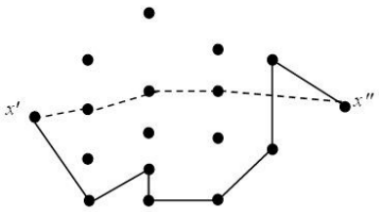
Example

The paths explored in this way intensify the search, because they connect good solutions (as shown in Figure 10.1 (a)) and, in general, diversify the search, because they are different from

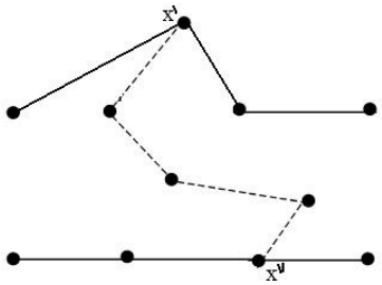
Algorithm 21 Path Relinking Pseudocode

```
1:  procedure PATH RELINKING(I, P)
2:      while P ≠ ∅ do
3:          R := ∅
4:          for each x ∈ P, y ∈ P \ {x} do          ▷ Recombine to build new population
5:              z := x
6:              z* := x
7:              while z ≠ y do          ▷ Build a path from x to y Z := arg min_{z'∈N(z)} d(z', y)
    z̃ := arg min_{z'∈Z} f(z')
8:                  if f(z̃) < f(z*) then
9:                      z* := z̃
10:                 end if
11:                 z := z̃
12:             end while
13:             if z* ∉ P then
14:                 z* := SteepestDescent(I, z*)          ▷ Improve the best solution from the path
15:                 R := R ∪ {z*}
16:             end if
17:         end for
18:         P := Rk                                          ▷ Update the population
19:     end while
20:     return (x*, f(x*))
21: end procedure
```

the path followed by the exchange heuristics, especially if the extremes are far away, as shown in Figure 10.1 (b).



(a) Intensification of the search.



(b) Diversification of the search.

Figure 10.1: Execution of Path Relinking.

Since the distance of $z^{(k)}$ from y is decreasing, one can explore worsening solutions without the risk of cyclic behaviour and unfeasible subsets without the rist of not getting back to feasibility: these are not useful by themselves, but are useful in order to open the way to improvements.

Variants of Path Relinking

There are several variants of Path Relinking, which are very similar to each other:

- **backward path relinking**: instead of building the path forward from x to y, the path

is built from y to x. Of course, checking all pairs like done in Algorithm 21, both the forward and backward path relinking are executed, so it's the same.

- **back-and-forward path relinking**: the idea is to build both paths, exactly as in Algorithm 21.
- **mixed path relinking**: build a path with alternative steps from each extreme, updating the destination. So, starting from x the algorithm makes a step in the direction of y; then, a step from y to x and the next step will try to go from the solution obtained by x to the solution obtained from y and so on.
- **truncated path relinking**: instead of building a whole path, the exploration stops after a while because there may be some reason to think that in the middle, far away from x and far away from y, there won't be promising solutions - which depends on the knowledge of the problem.
- **external path relinking**: instead of going from x to y, y is used as a reference but the algorithm goes in the *opposite* direction, so instead of minimizing the distance from y it's maximized. The purpose of this is to take into account that in some problems the good local optima tend to be far away from each other.

10.4 Genetic Algorithms

Genetic algorithms appartain to a family of methods that make heavy use of randomization. A feature that may distinguish this family from the others is the presence of an **encoding** of the solution, so genetic algorithms are *encoding-based* algorithms.

10.4.1 Encoding-based algorithms

Many recombination heuristics are based on the idea of taking solutions from a defined solution space and transforming them with an **encoding operation** in a more compact representation. A *representation* can be anything, for example strings of bit as shown in Figure 10.2. The

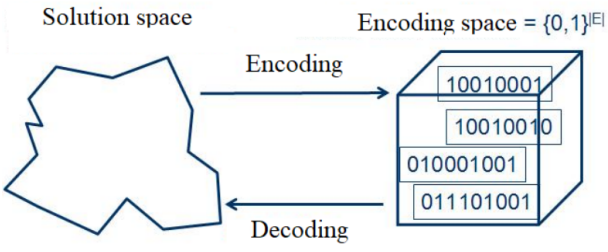


Figure 10.2: Representation of solution encoding.

operators that manipulate these solutions actually manipulate their encodings: exchanges, recombinations and so on operate not directly on the solutions but on the encoded solutions. This means that after the operators have been applied, the representation has to be **decoded**, in order to retrieve the solution that the encoding represents and understanding, for example, the cost of the solution and its feasibility. However, the distinction between solutions and encodings is not so clear: thinking about any practical implementation of an algorithm, the first thing to do

is to build a data structure that represents the solutions, which can be considered as encodings themselves.

Originally, the purpose of the encoding-decoding process was to get a level of **abstraction**, meaning that instead of solving a given problem, one transforms the problem into another which is, hopefully, more general and can be attacked by other techniques; in other words the idea is not to spend time understanding the problem itself but transforming it in a different form on which general operators could be applied, such as managing strings of bits or symbols of different kind or other mathematical structures that aren't related to the original problem. It was a really ambitious idea and it'll be shown that, unfortunately, it doesn't really work.

10.4.2 The algorithm

The **genetic algorithm** was originally proposed by Holland in 1975; this version is the most famous, but there is a plethora of similar methods. It's based on the idea of having a starting population of solutions, which is typical of all recombination heuristics.

Algorithm 22 Genetic Algorithm Pseudocode

1: **procedure** GENETICALGORITHM($I, X^{(0)}$)
2: $x^* := \arg\min_{x \in X^{(0)}} f(x)$
3: **for** $g = 1$ to n_g **do**
4: $X^{(g)} := Selection(X^{(g-1)})$
5: $X^{(g)} := Crossover(X^{(g)})$
6: $x_c := \arg\min_{x \in X^{(g)}} f(x)$
7: **if** $f(x_c) < f(x^*)$ **then**
8: $x^* := x_c$
9: **end if**
10: $X^{(g)} := Mutation(X^{(g)})$
11: $x_m := \arg\min_{x \in X^{(g)}} f(x)$
12: **if** $f(x_m) < f(x^*)$ **then**
13: $x^* := x_m$
14: **end if**
15: **end for**
16: **return** $(x^*, f(x^*))$
17: **end procedure**

The algorithm requires to build a population $X^{(0)}$ followed by three repeated phases:

1. **selection**: generate a new population starting from the current one
2. **crossover**: recombine subsets of two or more individuals
3. **mutation**: modify the individuals

as shown in Algorithm 22. Each of the phases can be implemented in many different ways, which characterize the algorithm. The other elements that characterise the algorithm are the starting population $X^{(0)}$ which is obtained randomly, by constructive heuristics or metaheuristics or by some exchange heuristics or metaheuristics; it must be defined in order to design the algorithm, but there's nothing special concerning genetic algorithms regarding it; the encoding, which are the units handled by the phases, so one may think of $X^{(i)}$ as a set of encoded solutions.

10.4.3 Encoding

The performance of a genetic algorithm depends heavily on the used solution encoding. The following properties should be satisfied (with decreasing importance):

1. each solution should have a unique encoding, different from the other ones (one-to-many or at least one-to-one); otherwise, there would be unreachable solutions
2. each encoding should correspond to a feasible solution; otherwise, the population would include useless individuals
3. each solution should correspond to the same number of encodings; otherwise, some solutions would be unduly favoured, since mutations and crossover opearators are stochastic operators, probability theory plays a role which is based on the number of cases
4. the encoding and decoding operations should be efficient; otherwise, the algorithm would be inefficient
5. small modifications to the encoding should induce small modifications to the solutions (**locality**); if one wants to be able to tune the operators so that the amounts of intensification and diversification can be imposed it's necessary that one can decide if the operators of mutation or crossover should be "strong" or "weak".

These conditions depend very much on the constraints of the problem. Farewell, abstraction!

Features of a good encoding

Delving into the second property (that is not including useless individuals), let's discuss the fact that once the mutation and crossover operations are applied, it's quite typical that the produced subsets are unfeasible.

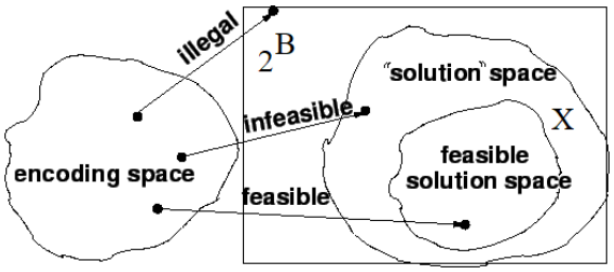


Figure 10.3: Representation of the map between encoding space and power set of the ground set.

The feasible set X is part of the unfeasible set which, in turn, is part of the power set of the ground set 2^B, which means that all solutions "outside" X are unfeasible, as shown in Figure 10.3. Why is a subset unfeasible? Of course, it depends on the problem; however, in general, in a problem is possible to distinguish between constraints that are of a *quantitative* nature and constraints that are of a *structural* nature. For example, in the KP, a solution is unfeasible if the volume exceeds the capacity of the knapsack (quantitative); in the BPP, assigning a single object to multiple bins is a structural infraction. The unfeasible solution space, $2^B \setminus X$, contains subsets that are, strictly speaking, not solution; however these subsets can be obtained by relaxing some constraint - typically the quantitative constraint - obtaining what's often called the *solution space*, which includes X and something similar: an example is, in the BPP, assigning each

object to a bin is considered a (unfeasible) solution even if some of the objects in a bin exceed its capacity. This makes sense as sometimes it's possible to build encodings that guarantee structural (though not quantitative) feasibility.

Encodings: the incidence vector

The most natural kind of encoding that can be proposed is the use of the incidence vector: after all, the solutions of combinatorial optimization problems are subsets and they can be described as binary incidence vector $\xi \in \mathbb{B}^{|B|}$ defined as

$$\begin{cases} \xi_i = 1 & i \in x \\ \xi_i = 0 & i \notin x \end{cases}$$

A generic binary vector corresponds

- in the KP to a set of objects: its weight could be excessive
- in the SCP to a set of columns: it could leave uncovered rows
- in the PMSP and in the BPP a set of assignments of tasks (objects) to machines (containers): it could make zero or more assignments for an elemen; in the BPP, it could violate the capacity of some container
- in the TSP to a set of arcs: it could not form a Hamiltonian circuit
- in the CMSTP (VRP) to a set of edges (arcs): it could not form a tree (set of cycles), or exceed the capacity of the subtrees (circuits).

Encodings: symbol strings

If the ground set is partitioned into components

$$B = \bigcup_{c \in C} B_c \text{ with } B_c \cap B_{c'} = \emptyset \ \ \forall c \neq c'$$

that correspond to objects, tasks, Boolean variables, vertices, nodes, eccetera and if the feasible solutions contain one element of each component

$$|x \cap B_c| = 1 \ \ \forall c$$

one can define for each $c \in C$ an alphabet of symbols describing component B_c, which have the purpose to describe the possible elements of each B_c: for example, in the BPP there are many possible containers for each object i and a symbol is given to each container. In general, each B_c has a different alphabet. Once the alphabet is built, a string of symbols ξ is built such as $\xi \in B_1 \times \cdots \times B_{|b|}$, that is it's composed of symbols such that

$$\xi_c = \alpha \implies x \cap B_c = \{(c, \alpha)\}$$

in other words, the nth symbol appartains to the alphabet describing B_n. A solution for the BPP can be described by putting, in sequence, the names of the containers: the first symbol will describe the assignment of the first object to some bin, the second symbol will describe the assignment of the second object to some bin and so on.

- Max-SAT: a string of n Boolean values, one for each logical variable
- PMSP: a string of machine labels, one for each task
- BPP/CMSTP: a string of container/subtree labels, one for each object/vertex:
  - the structural constraint on object assignment is enforced (if the graph is complete, for the CMSTP)
  - the quantitative constraint on capacity is neglected
- VRP: a string of vehicle labels, one for each node (but decoding the circuit for each vehicle is an $\mathcal{NP}$-complete problem)
- the solutions of the TSP, KP and SCP are not partitions

Encodings: permutations of a set

Another possible encoding for solutions is given by permutations of a given set: for example, in the TSP, solutions are Hamiltonian circuits, that issubsets of arcs, but they can also be seen as permutations of nodes, so this is a "natural encoding" of the solutions. However, permutations can be used also for other problems: for the KPP, for example, a given permutation of objects can be seen as a serie of objects to insert into the knapsack, stopping when hitting the capacity constraint, deciding whether to stop at the first violation or to skip the violating item - this may have some impact on the number of encodings that correspond to the same solution.

Any time a problem admits a partition as a solution, permutations can be used; in fact a method previously explored called **order-first split-second**, in which by build an auxiliary DAG, one could transform a permutation of elements into a partition by solving a shortest-path problem, explained in Subsection 8.4.1 about the efficient visit of exponential neighbourhoods. The problem with this representations is that there's a bias: some solutions have many encodings and some solutions have less encodings.

It's also possible to use permutations for solutions that haven't really got a partition structure or a permutation structure but a constructive algorithm is admitted such that at each step is chooses an element and chooses *how* to add it to the solution, if many ways exist. What one could do is permute the elements and apply the constructive algorithm not using the selection criterion to determine which elements must be taken next but determining it through permutation. Take the SCP for example: one has an adaptive greedy algorithm based on the ratio of the cost devided by the number of rows that the column covers; the selection criterion could be ignored and the columns can be permuted taking the first column, then the second, the third and so on, until one gets the final solution. Of course, some variations such as not considering columns that are redundant can be introduced. Sometimes the choice is now obvious as one's not permuting the ground set but something different, for example one could use, for the BPP, a permutation of objects and not assignments and a rule defining where to add the object must be introduced, which could be done based on the selection criterion. In the case of the TSP a number of greedy algorithms were seen where one chooses the node to add and *then* where to add it - this is exactly the same. Adopting this variant, then there are heuristic rules which, obviously, could be wrong so some solutions may not have any encoding; in a sense, this is better as it probably gives a better grasp of the objective function and good solutions will be reached earlier, but not all solutions may be represented.

10.4.4 Selection

After seeing a survey of some possible encoding, let's consider the first of the three phases of the method. The general idea is that given a current population $X^{(g-1)}$ which must be transformed in a new population $X^{(g)}$ which, at each iteration g, is built by extracting $n_p = |X^{(g)}$ individuals from the current population $X^{(g-1)}$:

$$X^{(g)} := Selection(X^{(g-1)})$$

The extraction follows two fundamental criteria:

1. an individual can be extracted more than once

2. better individuals are extracted with higher probability

so, selection is based on a **random mechanism**, as probability is involved. Every element of the old population $X^{(g-1)}$ has a probability of being extracted, with the criterion (2):

$$\varphi(\xi) > \varphi(\xi') \implies \pi_\xi \leq \pi_{\xi'}$$

the quality of a solutions is called **fitness** $\varphi(\xi)$ and for a maximization problem, commonly

$$\varphi(\xi) = f(x(\xi))$$

while for minimization problems

$$\varphi(\xi) = UB - f(x(\xi))$$

where $UB \geq f^*$ is a suitable upper bound on the optimum. The notation $x(\xi)$ represents the decoding process. There are three main mechanisms proposed to map fitness to probability.

Proportional selection

The original scheme proposed by Holland in 1975 assumed a probability *proportional to the fitness*:

$$\pi_\xi = \frac{\varphi(\xi)}{\sum_{\xi \in X^{(g-1)}} \varphi(\xi)}$$

this is named **roulette-wheel selection** (or spinning wheel selection) as in order to implement this algorithm one must compute the fitness for every individual in the population and then build the partial sums of the fitness

$$\Gamma_i = \big(\sum_{k=1}^{i-1} \pi_{\xi_i}, \sum_{k=1}^{i} \pi_{\xi_i} \big]$$

in $\mathcal{O}(n_p)$ time and, after extracting a random number $r \in U(0,1]$, choose an individual i^* such that $r \in \Gamma_{i^*}$ in $\mathcal{O}(\log n_p)$ time; in other words, r "lands" in an interval, which is selected. In the example of Figure 10.4, having $r = 0.78$ and 4 different possibilities, by binary search one chooses the interval containg r, that is A_4.

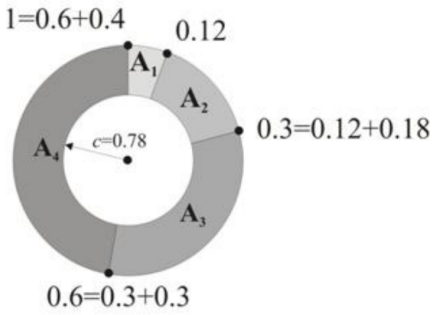


Figure 10.4: Roulette wheel selection.

Rank selection

There are two problems with proportional selection: the first is stagnation. In fact, in the long term all individuals tend to have a good fitness, and therefore similar selection probabilities. The second problem, typical of the first phase of the phase of the search is when there are some bad individuals and a lot of very bad individuals and the difference between the two is very large, the selection quickly generates a bad population.

A proposal devised to avoid these two problems is to guarantee that two probabilities aren't too similar but at the same time not too different for each other, creating a scheme of decreasing probability. Such a scheme is proposed in GRASP, called **rank selection** method, which consists in sorting the individuals by nondecreasing fitness

$$X^{(g)} = \{\xi_1, \cdots, \xi_{n_p}\} \text{ with } \varphi(\xi_1) \leq \cdots \leq \varphi(\xi_{n_p})$$

assigning to each individual a probability that is linearly increasing based on its position

$$\pi_{\xi_j} = \frac{k}{\sum_{k=1}^{n} k} = \frac{2k}{n_p(n_p-1)}$$

which can be computed in $\mathcal{O}(n_p)$ time, as there's a way to avoid full sorting and performing a linear scan to find the kth invidivual for each k. This is still slower than proportional selection, but it has advantages.

Tournament selection

An efficient compromise consists in extracting n_p random subsets $\bar{X}_1, \cdots, \bar{X}_{n_p}$ of size α selecting the best individual from each subset

$$\xi_k := \arg\max_{\xi \in \bar{X}_k} \varphi(\xi), k = 1, \cdots, n_p$$

which is done in time $\mathcal{O}(n_p\alpha)$. The parameter α tunes the strength of the selection: if $\alpha \approx n_p$ it favours the best individuals, while if $\alpha \approx 2$ it leaves chances to the bad individuals.
All selection procedures admit an **elitist variant**, which includes in the new population the best individual of the current one. Besides this, the best individual found so far is always kept.

10.4.5 Crossover

The crossover operator derives from a biological methaphor: the chromosomes of living being, when they reproduce, typically have this strange, anomalous behaviour in which two sequences

of DNA "exchange" part of their material, crossing each other and in the end two different individuals that possess part of the genetic code of each of the two parents individuals. This has been easily simulated once an encoding is given by applying similar mechanisms. The crossover operator in general combines $k \geq 2$ individuals generating other k individuals and, commonly, $k = 2$. The basic idea of crossover (**simple crossover**) is extracting a random position of the encoded solution, usually with uniform probability, splitting the encoding in two parts, as shown in Figure 10.5 (a). In that picture, the two *parents* are split in the position between the fifth and

(a) Simple crossover (b) Double crossover

Figure 10.5: Crossover.

sixth symbol and the two *children* encoding are built taking the first part of the black individual and the second part of the red individual and merging them and doing respectively for the second child.

This behaviour can be generalized introducing multiple splits, for example in two positions (**double crossover**), where the encoding is split in three parts and the extreme parts of the encodings of the two parents individuals are exchanged, generating the two children, as shown in Figure 10.5 (b).

Bias

Generalizing, one obtains the **α-points crossover**, where α points are extracted at random with uniform probability and the encodings are split in $\alpha + 1$ positions, exchanging the odd parts of the encoding of the two individuals. For small values of α this implies a positional bias: symbols colse in the encoding thend to remain close. Suppose to have a KP where solutions are represented by an incidence vector, (1 if the object is inside the knapsack, 0 if it's outside). Suppose now that two parents individuals are given where two objects are present in one individual and not present in the other. Generated children can have both object of the first, none of the two or one of the two: these last children are possible only if one of the α positions falls between the two objects. Let's consider the following cases. It's possible that the two objects are adjacent: in this case it's very unlikely that the two objects will be separated, so in the children they will be both present or both missing. Think about a situation where in an individual which includes three objects: the first, the second and the last; in this situation, having $\alpha = 1$ will almost certainly separate the first and the last, while the first and the second won't be separated with the same frequency or probability: this is clearly a bias that has no reasonable motivation.

To cancel the bias, one can adopt the **uniform crossover**, which is more modern and less attached to the biological analogy. In the uniform crossover one must first build a random vector m which is made of bits uniformly extracted from $\{0,1\}$, so $m \in U(\mathbb{B}^n)$ which is called **mask**. If $m_i = 1$, then the symbols in position i of the two individuals must be exchanged; if $m_i = 0$ the symbols in position i remain unchanged. Figure 10.6 represents this possibility.

In Figure 10.6, there's a 1 in the first position of the mask, so the black one and red one are exchanged; in this case, both parents have a 1 in the first position. In the second position of the mask there's a 0, so no exchange occurs and the two children have the symbol in the second position of their parents (right with right, left with left) and so on. This destroys the positional

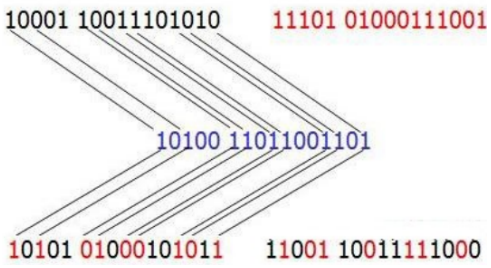


Figure 10.6: Uniform crossover.

bias because the fact of keeping two elements inside in the same position or a different position doesn't depend on where the two elements are.

Crossover versus Scatter Search and Path Relinking

The crossover operator resembles the recombination phase of SS and PR: the main differences are that

1. it recombines the symbols of the encodings, instead of
   - recombining the solution (SS)
   - performing a chein of exchanges on the solutions (PR)
2. it operates on the whole population, instead of only a reference set R
3. it operates on random pairs of individuals, instead of methodically scanning all pairs of solutions of R
4. it generates a pair of new individuals, instead of
   - generating a single intermediate solution (SS)
   - visiting the intermediate solutions and choosing one (PR)
5. the new individuals enter hte population, instead of becoming candidates for the reference set

Of course, based on which encoding is used and how is the crossover implemented, these differeces can be more or less pronounced.

10.4.6 Mutation

The mutation operator modifies an individual to generate a similar one by scanning an encoding ξ one symbol at a time and deciding, with probability π_m, to modify the current symbol. The kind of modification depends on the encoding: for binary encoding it's a simple **flip** of ξ_i into $\xi_i' := 1 - \xi_i$, as shown in Figure 10.7.

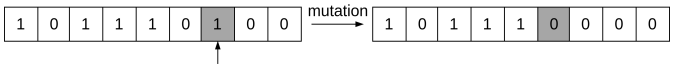


Figure 10.7: Mutation.

In the case of strings of symbols, each symbol ξ_c is replaced, with a probability π_m, with a random symbol $\xi_c' \in B_x \setminus \{\xi_c\}$ extracted with random probability. In the case of permutations there are many proposals: exchange two random elements in the permutation (*swap*), reverse the stretch between two random positions of the permutation and others.

Mutation and exchange heuristics Exchanging two elements or reversing the stretch resembles very much what happens in the TSP and in fact there are strong similarities between exchange heuristics and mutations and the mutation operator has strong relation with local search in general: the main differences are that

1. mutation modifies the symbol of an encoding, instead of exchanging elements of a solutions

2. mutation operates on random symbols, instead of exploring a neighbourhood systematically

3. mutation operates on a random number of symbols, instead of exchanging a fixed number of elements

However, classifying an operator can be a matter of taste.

10.5 The Feasibility Problem

Many heuristics have to manipulate or generally handle subsets, solutions, that are *unfeasible*, which generates the **Feasibility Problem**. Some heuristics such as constructive and exchange heuristics often succeed in avoiding this, but in some cases and particularly in metaheuristics that make heavy use of randomization, the existence of unfeasible subsets becomes so important that one can't ignore it. In the context of recombination heuristics, it can be seen in encoding functions: if these functions aren't invertible, meaning that not every encoding can be decoded into a solution, the crossover and mutations operations in Genetic Algorithms often generate encodings that aren't legal; so there's a distintion between **feasible encodings** that correspond to feasible solutions and **unfeasible encodings**, which correspond to legal, but unfeasible subsets. The existence of unfeasible encodings implies several disadvantages:

1. inefficiency: computational time is lost handling meaningless objects

2. ineffectiveness: the heuristic explores less solutions

3. design problems: fitness must be definesd also on unfeasible subsets

There are three main approaches to face this problem: the first is the use of special encodings and operators which allow to avoid or, at least, limit it. This was already discussed while introducing the three possible encodings; the incidence vector often yields unfeasible encodings so sometimes one resorts to the use of strings of symbols in order to avoid infeasibility. The second is the use of repair procedures and the third is to use penalty functions, meaning that infeasibility is accepted but heavily discouraged.

10.5.1 Special encodings and operators

The idea is to investigate encodings that nearly always yield feasible solutions, such as permutation encodings and order-first split-second decoding for partition problems such as CMSTP, VRP and so on and permutation encodings and a constructive heuristic decoding for scheduling problems such as PMSP; alternatively investigate crossover and mutation operators that maintain feasibility, such as operators that simulate k-exchange for the TSP. These methods tend to closely approximate exchange and recombination heuristics based on the concept of neighbourhood while giving up the idea of abstraction and focus on the specific problem, contrary to classical genetic algorithms.

10.5.2 Repair procedures

A repair procedure $x_R(\xi)$ receives an encoding ξ that yields an unfeasible subset $x(\xi) \notin X$ and returns a feasible solution $x_R \in X$. The procedure is applied to each unfeasible encoding $\xi \in X^{(g)}$ and one must decide if$\xi(x_R(x(\xi)))$ replaces ξ in $X^{(g)}$ or ξ is kept in the population and $x_R(\xi)$ is used to update the best known result, understanding if something nice has been found; in other words, the choice is between managing feasible solutions only or admitting unfeasible encodings in the population.

The first family introduce a strong bias in favous of feasible encodings and a bias in favour of the feasible solutions most easily obtained with the repair procedure. So an argument against the replacement of solutions is to keep the diversity in the population, instead of replacing them with the encoding of the repair solutions which could easily bring a lot of duplicates in the population.

10.5.3 Penalty functions

The last method to deal with unfeasibility is to introduce penalty functions. In this case, instead of avoid unfeasibility or translating it into feasibility, one simply accepts unfeasibility while highly discouraging it, working on the selection operator; if the objective function is extended to unfeasible subsets $x \in 2^B \setminus X$, the fitness function $\varphi(\xi)$ can be extended to any encoding, but many unfeasible subsets have fitness larger than the optimal solution, hence the selection operator tends to favour such unfeasible subsets! An example of this is, in the KP, taking a solution which exceeds the capacity of the knapsack which, quite likely, has a value higher than a solution that respects the volume constraing.

To avoid this situation, the fitness function must combine the objective value $f(x(\xi))$ and a measure of infeasibility $\psi(x(\xi))$ defined

$$\begin{cases} \psi(x(\xi)) = 0 & x(\xi) \in X \\ \psi(x(\xi)) > 0 & x(\xi) \notin X \end{cases}$$

If the constraints of the problem are expressed by equalities or inequalities $\psi(x)$ can be defined as a weighted sum of their violations. But then the problem turns to the choice of the weights and their variability or adaptivity.

Definition of the fitness function

The most typical combination for the fitness function is the **absolute penalty** where, given two encodings ξ and ξ', if both are feasible the one with smaller f is better; if exactly one is feasible,

the feasible one is better than the other and, finally, if none is feasible the one with smaller ψ is better than the other. This definition is suitable for rank and tournament selection, as in that case the individuals must be ordered, which can be done by comparison.

Another combination is that of **proportional penalty**, where the unfeasible encodings have a penalty increasing with the violation:

$$\varphi(\xi) = f(x(\xi)) - \alpha\psi(x(\xi)) + M$$

for maximisation problems and

$$\varphi(\xi) = -f(x(\xi)) - \alpha\psi(x(\xi)) + M$$

for minimisation problems; the offset M guarantees that $\varphi(\xi) \geq 0$ for all encodings. As the combined values have different units of measure, the coefficient α is used as a conversion coefficient, which must be tuned somehow, taking also in account the importance of unfeasibility with respect to the cost. There can be several constraints, so they must be combined with weights; this problem is seen in both proportional and absolute penalty and the weights are summarized in $\psi(x)$.

A third approach, where the **penalty is obtained by repair**, doesn't require many coefficients and it adopts a repair procedure; if the original unfeasible encoding is kept in the population, the fact that a repair procedure is used is not only due to the necessity of feasible solutions but also define the fitness:

$$\varphi(\xi) = f(x_R(\xi) \text{ or } \varphi(\xi) = UB - f(x_R(\xi))$$

since usually $f(x_R(x(\xi))) \geq f(x(\xi))$.

Proportional penalty functions: weight tuning

Experimentally, it's better to use the smalles effective penalty: if the penalty is too small, too few feasible solutions are found - once entered the unfeasible solution space one might never get out of it, as there are many unfeasible subsets which have a very good value and if the penalty is too large, the search is confined within a part of the feasible region and isn't crossed. A method proposed by Glover called *strategic oscillation* used for problems which have a complex-shaped search space, that is feasible regions are separated by unfeasible regions: it makes sense to cross the boundary and oscillate between the regions, so to be able to reach other feasible regions.

The problem is deciding how much to step inside the unfeasible region: this requires a good tuning of parameter α and maybe, if the unfeasibility $\psi(x)$ is composed of many terms, a good tuning of the different weights. The idea is that α could be tuned in time either in a dynamic way that is fixed, so that α is variable but the variations are decided once and for all at the beginning of the algorithm; α could be tuned adaptively, making it depend on the current situation: increases when unfeasible encodings dominate the population and decreases when feasible encodings dominate or, finally, an evolutionary method could be used by encoding α in each individual, in order to select and refine both the solutions and the algorithm parameter:

- **dynamic methods**: many parameters shouldn't keep the same value for the whole process of an algorithm; one idea is to change them with a fixed rule in time as it was for the temperature in SA. There must be a reason to modify a parameter, that is to obtain a certain result. In the case of SA the idea was to start with large values to diversify the search initially to intensify it later; α could start small to diversify the search incrementing as time goes by, in order to go back to the feasible region.

- **adaptive methods**: a second idea is to modify the parameter adaptively according to what's happening "currently" and not with an a-priori scheduling, so that it can be increased when there are lots of unfeasible encodings to avoid losing time and decreasing it when there are too few unfeasible encodings.

- **evolutionary methods**: the third, very sophisticated method proposed, is to encode in each individual a vector describing the value of α, so to describe in the individual not only its meaning but also how it should evolve in the future.

10.6 Other Recombination Metaheuristics Approaches

10.6.1 Memetic Algorithms

A different approach, though very similar to Genetic Algorithms, is that of **Memetic Algorithms**, inspired by the concept of *meme* coined by Richard Dawkins in 1989 and proposed by Moscato in 1989; the idea is that cultural evolution of ideas follow, somehow, the lines of genetic evolutions with selections, crossover, mutations but with some differences: individuals can *change* their ideas, so it's like having a Lamarckian evolution in which the memes, instead of being selected and destroyed or modified blindly they can be modified with some reason inside the individuals who propagate them.

Outside the metaphor, memetic algorithms combine *genotypic* operators that manipulate the encodings such as crossover and mutation and *phenotipyc* operators that manipulate the solutions, such as local search. In short, the solutions are improved with exchanges before reencoding. Of course, this can be done in many different ways, but the idea is always the same, having several parameters determine how to apply local search, defining how often (at every generation or after a sufficient diverification), to which individuals (all, the best ones, the most diverse one), for how long (until a local optimum is found, beyond, before) and with what method (steepest descent, VNS, ILS and so on).

10.6.2 Evolution Strategies

A second different approach called **Evolution Strategies**, proposed by Rechenberg and Schwefel in 1971, wasn't originally designed for combinatorial optimization problems so uses solutions encoded into real vectors (vectors of real numbers), as the original application was to engineering design problems in which objects such as engines were to be designed which have a number of real parameters which are hard to solve using optimization algorithms. The proposed idea was to fix a possible design and, by simulation, understand how good it is and then modify it slightly to see if a better model could be found but not necessarily with a descent algorithms but with some possible worsening in order to get out of local optima as usual.

The idea is that there's a population of indivuduals of size μ, typically smaller than the size of the population of genetic algorithms (in fact, originally, $\mu = 1$), and these individuals generate λ candidate descendants with a mutation operator that starts from the given encodings and adds to each of the real number a disturbance that is a random value with 0 average and a normal distribution with variance σ

$$\xi' := \xi + \delta \text{ with } \delta \in N(0, \sigma)$$

getting a new vector (the descendant itself); the two population of μ and λ compete to build the new population:

- in the (μ, λ) strategy the best μ descendants replace the original population, even if some are dominated
- in the $(\mu + \lambda)$ strategy the best μ individuals overall, predecessors or descenants, survive in the new population

Originally, crossover wasn't used but can be added later when the population grew. This strategy is usually applied to continuous optimization problems but some scholar applied it also to combinatorial optimization problems.

Part VI

Appendices

APPENDIX A

Laboratory on Constructive Heuristics

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.

Section A.1 defines the problem and describes the data structure and the basic procedures (implemented as C libraries) that will be used to manipulate the instances and the solutions. Section A.3 describes the implementation and evaluation of some constructive and destructive heuristics.

A.1 The Maximum Diversity Problem

A.1.1 Definition

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

- a set P of *points* in an abstract space (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 $\left(d_{ij} + d_{ji}\right)/2$, the value of any solution x is unchanged, since the sum $\sum_{i \in x} \sum_{j \in x} d_{ij}$ contains either both distances or none.

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

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.

A.1.2 Benchmark instances

The literature provides several classes of benchmark instances for the *MDP*. Since most of them have a fixed size, but we aim to discuss also the dependence of the results on size, we will generate another benchmark set, inspired by the available ones, but with a larger range of different sizes. The main features of these instances are:

- set P ranges from $n = 100$ to $n = 1\,000$ points by steps of 100;
- the integer number k is equal to $0.1n$, $0.2n$, $0.3n$, $0.4n$;
- the values d_{ij} of the distance function (with $i < j$) are random integer numbers uniformly distributed in $\{1, \ldots, 100\}$;
- the values for $i \geq j$ derive from the assumptions made in the previous section.

Overall, they are 40 instances (ten different sizes times four different values of k).

They are neither particularly significant nor realistic instances, but they are hard enough not to be trivially solved to optimality by any method, but easy enough to assume that the best known solution is probably close to the optimum (even if this has not been proved) and small enough to require a reasonable processing time from any polynomial algorithm.

The data are provided in text files. The name of each file reports the features of the instance: file `n[%n]k[%k].dat` corresponds to an instance with a set P of `%n` points, and a required cardinality equal `%k` for the solution.

The file adopts the AMPL format, a standard format to represent Integer Programming problems, used by general-purpose modelling languages and solvers for that family of problems. The format is rather self-evident (and anyway does not concern us, given that library `data.h` provides a function to load the data into the memory).

```
param n := 100 ;
param k := 10 ;
param D :=
[1,1] 0 [1,2] 42 [1,3] 10 [1,4] 75 [1,5] 53 ...
[2,1] 42 [2,2] 0 [2,3] 13 [2,4] 33 [2,5] 84 ...
...
```

A.2 Instance representation

The C library `data.h` provides the data structure

```
typedef struct data_s
 {
  int n;   /* cardinality of the set of points P */
  int k;   /* cardinality of the feasible solutions x \in X */
  int **d; /* distance matrix between the points */
 } data_t;
```

that we will use to represent each given instance I of the *MDP*, since it consists of three simple components: a set P, a metric d and an integer number k.

We will represent set P with the natural numbers from 1 to $n = |P|$, since this set has no other specification besides the metric. We do not adopt the classical C language convention that represents integer number sets starting from zero, in order to keep consistent with the data available in the literature, and to possibly use index 0 for special operations.

We will represent metric d with a square integer matrix. We could save space exploiting the symmetry of the metric, for example by representing only the values d_{ij} with $i < j$, but this would require to test at each access to d_{ij} whether $i < j$ or not, in order to exchange the two indices if the result of the test is negative. Since we expect to access the data a huge number of times, we choose to represent all of them, preferring time efficiency over space efficiency.

The distance matrix is dynamic, and will be allocated when loading the data from a text file and deallocated at the end of the algorithm. The C library `alloc.h` is already available, and provides functions to allocate integer vectors (given the actual number of elements) and matrices (given the actual number of rows and columns).

```
/* Allocate a vector of n int */
int *int_alloc (int n);
```

```
/* Allocate a matrix of (n1,n2) int */
int **int2_alloc (int n1, int n2);
```

To manage the instances, library `data.h` provides functions to load the data from a text file (in the standard AMPL format), to deallocate the struct `data_t` described above and to print the instance on the screen (once again in AMPL format).

```
/* Load from the AMPL file data_file the instance *pI */
void load_data (char *data_file, data_t *pI);

/* Deallocate the instance *pI */
void destroy_data (data_t *pI);

/* Print the instance *pI in AMPL format */
void print_data (data_t *pI);
```

As observed above, a point is just an abstract object with no associated information, represented by an integer number ranging from 1 to P. With some excess of zeal, we will however introduce a level of abstraction and distinguish:

- on the one hand, the numerical index of a point, that is an `int`;
- on the other hand the abstract point, with all its associated information (if any), that is a `point`.

For the sake of rigour, we shall use `int` variables to scan the indices (for example, in the distance matrix) and `point` variables to scan the actual points (for example, the elements of a solution x and of its complement $P \setminus x$, as will be described in the following section). As long as the two concepts coincide, we will identify them with the type definition:

```
typedef int point;
```

allowing future developments, in which a point could be associated to auxiliary information (e. g., coordinates, text strings, etc...) besides its numerical index. This approach allow more easily to modify the data structures without modifying the algorithms already implemented.

The disadvantage is that, in order to switch between points and indices, we must use the functions:

```
/* Get the index of point i in instance *pI */
int get_index (point i, data_t *pI);

/* Get the point of index id in instance *pI */
point get_point (int id, data_t *pI);
```

If the data structure will need to be modified, these functions will take care of the corresponding type conversions, but currently they just return in output the same value received in input. This is an inefficiency, that can be controlled by adopting suitable technological solutions, such as inline functions or macros to implement the conversion.

A.2.1 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

$$x_i = \begin{cases} \texttt{true when } i \in x \\ \texttt{false when } i \in P \setminus x \end{cases}$$

2. with a *list of elements*, which allows to scan only the elements of the solution $i \in x$

The choice between the two representations depends on the type of operations that the algorithm needs to perform: the inclusion test of a point in the solution is efficient in the first representation ($O(1)$), inefficient in the second one ($O(n)$); the opposite holds for the operation of scanning only the elements of the solution, or the elements out of it. The algorithms we want to implement mainly use operations on lists, both the internal and external points. For example, each step of a constructive algorithm scans the set $\Delta_A^+(x)$, that in the *MDP* is the complement of the solution, $P \setminus x$, whereas the computation of the value of a solution, $f(x)$, requires to scan its elements. However, we will adopt both representations to keep as flexible as possible. We will also represent the complementary set of the solution, $P \setminus x$ as another list, because many algorithms require to scan its points. If one of the two representations is not actually used, we can decide to remove it *a posteriori*. The C library `solution.h` provides the struct

```
typedef struct solution_s
{
  int f;          /* solution value */

  bool *in_x;     /* incidence vector: in_x[i] = true if i \in x,
                                       in_x[i] = false if i \notin x */

  /* Lists of points in solution x and in the complement P \setminus x */
  int head_x;     /* sentinel of the list of points in x */
  int head_notx; /* sentinel of the list of points in P \setminus x */
  int *next;      /* next     element for each point i in either list */
  int *prev;      /* previous element for each point i in either list */

  int card_x;     /* cardinality of the solution */

} solution_t;
```

The value f of the solution is saved in member `f` and kept up to date, so that it can be accessed in constant time $O(1)$, instead of recomputed every time. Its recomputation, in fact, would require $O(n^2)$ time if done scanning the incidence vector, $O(k^2)$ time if done scanning list x.

Note: since the distance matrix is symmetric and integer, $f(x)$ is certainly an even number, as it is the sum of pairs of equal terms. It is a common convention to report in `f` half of the overall sum. We shall discuss this point again later.

The boolean vector `in_x` represents the incidence vector. The `boolean` type, with its two values `false` and `true` is declared in the library `defs.h`.

```
typedef enum _bool bool;
enum _bool {false = 0, true = 1};
```

and dynamic vectors of booleans can be allocated thanks to the already cited library `alloc.h`, that provides function

```
/* Allocate a vector of n bool */
bool *bool_alloc (int n);
```

The two lists that represent the solution x and its complement $P \setminus x$ are *doubly-linked circular lists with sentinel*, so that every fundamental operation (insertion, extraction, etc...) can be performed in constant time, at the cost of a larger memory occupation. Briefly, such lists can be scanned in both directions and are never physically empty, because they always contain the fictitious element known as *sentinel* (by convention, an empty list is a list containing only the sentinel). This removes the need for different ways to operate on different parts of the list (the beginning, inner positions or the end).

We will adopt the implementation of the two lists with vectors and indices, instead of the implementation with pointers and dynamically allocated structures, because all possible points are defined once for all at the beginning of the algorithm, and only their positions change dynamically during the execution. Moreover, since the two lists do not intersect (they are complementary), we will exploit the same vectors and indices `next` and `prev` for the two lists; only the heads `head_x` and `head_notx` will be different. The sentinel of list x has index `0`, the sentinel of list $P \setminus x$ has index `card_N+1`. Only the intermediate values correspond to regular indices. The following example shows how the solution $x = \{1,3,7\}$ of an instance with point set $P = \{1,2,3,4,5,6,7\}$ will be represented:

```
f        46

           1  2  3  4  5  6  7
in_x     [ 1  0  1  0  0  0  1 ]

head_x    0
head_notx 8

         0  1  2  3  4  5  6  7  8
next   [ 1  3  4  7  5  6  8  0  2 ]
prev   [ 7  0  8  1  2  4  5  3  6 ]

card_x  3
```

The cardinality of the solution $|x|$ (`card_x`) should be fixed to k, but reporting it explicitly allows the structure to represent also general subsets of P, and in particular *partial solutions* of cardinality $< k$. In fact, the set of all partial solutions is the search space $\mathcal{F}_A$ for all the constructive algorithms considered in the following.

In order to avoid programming technicalities, we will hide many implementation details using library functions to access the data. This also allows, if necessary, to modify the low-level implementation without affecting already implemented algorithms. It can, however, imply some time inefficiencies, because it requires function calls instead of the simple direct access to data structures. Such inefficiencies can be easily overcome by using macros (or `inline` definitions in C++), so we accept them, but we do not describe how to do that to avoid technicalities.

The C library `solution.h` provides some functions to manage solutions:

```
/* Create an empty solution for a problem of size n */
void create_solution (int n, solution_t *px);

/* Deallocate the solution *px */
void destroy_solution (solution_t *px);

/* Turn a solution into the empty set for a problem of size n */
void clean_solution (int n, solution_t *px);

/* Copy solution *px_orig into solution *px_dest */
void copy_solution (solution_t *px_orig, solution_t *px_dest);

/* Print by increasing indices solution *px for a problem of size n */
void print_sorted_solution (solution_t* px, int n);
```

The creation of an empty solution corresponds to the typical initial step of a constructive heuristic, that starts from the empty subset. The deallocation is performed at the end of the algorithms. Cleaning a solution allows to restart an algorithm without deallocating and reallocating the memory. The copy function is useful to update the best known solution when the current one improves it. The print function is useful to analyse the results: it reports on the screen on a single row the name of the data file, the value of the objective and the list of points in solution x.

Another block of functions allow to access the solution x and its complement $P \setminus x$, avoiding any explicit reference to their concrete implementation. List x can be scanned with the following library functions:

```
/* Return the first and the last point of solution *px */
point first_point_in (solution_t *px);
point last_point_in (solution_t *px);

/* Return the point following and preceding i in solution *px */
point next_point (point i, solution_t *px);
point prev_point (point i, solution_t *px);

/* Indicate whether i is a regular point or a sentinel */
bool end_point_list (point i, solution_t *px);
```

In order to scan the complementary list $P \setminus x$, the functions to access the first and last point change, because the sentinel has a different index:

```
/* Return the first and the last point of the complement of solution *px */
point first_point_out (solution_t *px);
point last_point_out (solution_t *px);
```

but the functions that return the next and the previous point and the function that indicates whether the point is regular or the sentinel remain the same for both lists, because they share the same vectors and follow the same rules.

For example, given two points:

```
point i, j;
```

in order to scan solution x from the first to the last point, one can perform the loop

```
for (i = first_point_in(&x); !end_point_list(i,&x); i = next_point(i,&x))
```

and to scan the complement $P \setminus x$ from the first to the last point, one can perform the loop

```
for (j = last_point_out(&x); !end_point_list(j,&x); j = prev_point(j,&x))
```

The main manipulations of a solution in a constructive algorithm is the addition of a point, that requires to move it from the complementary list to the solution list, while at the same time updating the incidence vector and the value of the objective. It is adviseable to define a specific function for this basic operation, to make it as efficient as possible and to guarantee the consistency of the data structures. Functions operating on a single list would not make much sense for the overall problem and could easily introduce inconsistencies in the representation of the solution.

```
/* Add point i to solution *px */
void add_point (point i, solution_t *px, data_t *pI);

/* Delete point i from solution *px */
void delete_point (point i, solution_t *px, data_t *pI);
```

These functions must keep all components of the data structure consistent and up to date [1]. Function `add_point`:

1. adds to the objective function the sum of all distances of the newly added point from the previous ones (but not the reverse distances because `f` reports only half of the objective); this operation requires the distance matrix;
2. increases by one the cardinality `card_x`;
3. finds the index `id` of point `i` and sets the corresponding value of `in_x` to `true`;
4. extracts point `i` from list $P \setminus x$;
5. adds point `i` to list x.

Considering the previous example, adding point 4 to the solution yields the following data structure:

```
f        90

           1  2  3  4  5  6  7
in_x     [ 1  0  1  1  0  0  1 ]

head_x    0
head_notx 8

         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
```

[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.

The library also provides a function to remove a point from the solution, moving it from the solution list to the complementary list, and correspondingly updating the incidence vector and the value of the objective.

```
/* Delete point i from solution *px */
void delete_point (point i, solution_t *px, data_t *pI);
```

This function:

1. subtracts from the objective function the sum of all distances of the newly removed point from the remaining ones; this operation requires the distance matrix;
2. decreases by one the cardinality `card_x`;
3. finds the index `id` of point `i` and sets the corresponding value of `in_x` to `false`;
4. extracts point `i` from list x;
5. adds point `i` to list $P \setminus x$.

Removing point 1 from the solution previously augmented yields the following data structure:

```
f        72

           1  2  3  4  5  6  7
in_x     [ 0  0  1  1  0  0  1 ]

head_x    0
head_notx 8

         0  1  2  3  4  5  6  7  8
next   [ 3  8  5  7  0  6  1  4  2 ]
prev   [ 4  6  8  0  7  2  5  3  1 ]

card_x  3
```

All these operations require constant time, except for the update of the objective function, which requires $O(|x|)$ time for the addition and $O(n-|x|)$ time for the removal. This will be useful to implement destructive algorithms.

A.2.2 Consistency check

Notice that the use of a double representation for the solution implies a computational overhead and an additional effort to keep the two representations up to date. This choice should be justified by a better efficiency gained somewhere else. Moreover, this choice allows the risk to lose the consistency between different elements of the two representations. Such a risk is limited by the use of clearly defined functions to manipulate the solutions, but it is anyway impossible to remove completely. In our case, the data structure `solution_t` includes five potentially inconsistent components: objective value, cardinality, incidence vector, solution list and complementary list. The manipulation functions should guarantee the consistency between the five components. Even assuming that they are consistent at the beginning, every subsequent modification (for example, the introduction of new fields in the solution to allow other operations

or to perform the same operations more quickly) could introduce inconsistencies, and therefore errors.

When implementing a heuristic algorithm, it is therefore a very good practice to write and maintain a function to check the internal consistency of the data structures. This function usually assumes one of the components as valid *a priori*, and recomputes the other ones, checking whether their current values are correct or not. The choice of the valid component is arbitrary, provided that it is sufficient to derive all of the other ones. In general, one uses the simplest component, that it that which is less likely to be incorrect. In our case, the available check function starts from the incidence vector, and derives from it the objective value, the cardinality and the two lists.

```
/* Check the internal consistency of solution *px based on instance *pI,
   starting from the incidence vector */
bool check_solution (solution_t *px, data_t *pI);
```

If the function finds an inconsistency, it return the value `false`, and the user can decide whether to terminate the execution to correct the code. Of course, the check function is used only during the implementation of the algorithm, and does not appear in its final version.

A.2.3 The main function

The `main` function in all the algorithms presented in the following sections manages the parsing of the command line (that is, the interpretation of the parameters of each algorithm), the loading of the data, the allocation and deallocation of the data and the solution, the execution of the algorithm, the determination of the computational time and the print of the result on the screen. Its general structure can be described as follows.

```
parse_command_line(argc,argv,data_file,&param);

load_data(data_file,&I);

create_solution(I.n,&x);

start = clock();
...
end = clock();
CPUtime = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6lf ",CPUtime);
print_sorted_solution(&x);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

where `data_file` stands for the name of the text file reporting the instance and `param` is a specific structure for each family of algorithms tested, that collects the parameters that identify a single algorithm. `I` is the instance of the problem and `x` the solution obtained. The starting time, ending time and overall duration of the computation are `start`, `end` and `CPUtime`. Finally,

the dots (...) represent the actual call of the algorithm considered. The print of the results occurs on a single line of the screen, so that several calls to the algorithm can be collected in a *script*. This allows to run the algorithm on several benchmark instances, or with several different parameters, appending the results in a single file, one row for each run.

A.3 Constructive heuristics

A.3.1 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[2].

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.

A.3.2 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 $\delta f(x,i) = f(x \cup \{i\}) - f(x)$

$$\delta f(x,i) = \sum_{j \in x} d_{ji} + \sum_{j \in x} d_{ij} + d_{ii} = 2 \sum_{j \in x} d_{ji}$$

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

[2]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.

```
// 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.

A.3.3 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 A.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 A.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.

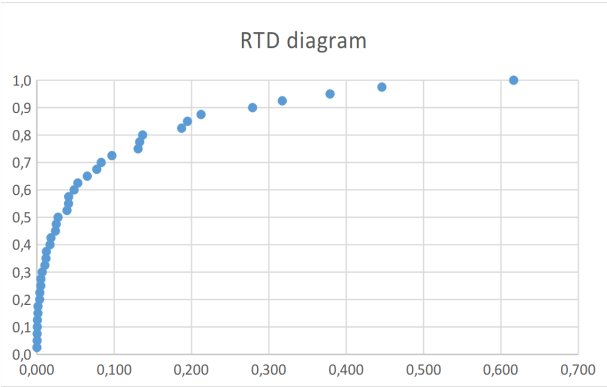


Figure A.1: *RTD* diagram for the greedy algorithm on the benchmark

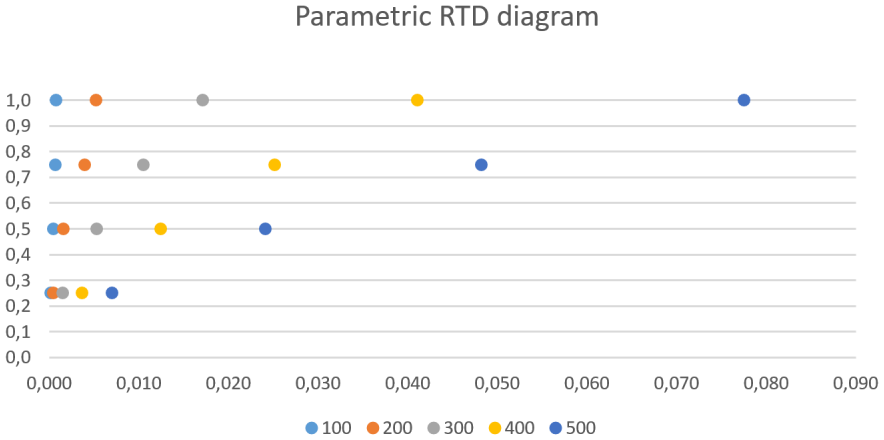


Figure A.2: Parametric *RTD* diagrams for the greedy algorithm on the benchmark

The correct tool to describe the dependence of the computational time T_A on the instance size P is the *scaling diagram* (see Figure A.3). The first remark is that T_A is rather low, even for large instances ($n = 1000$), 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 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 $T_A \in O\left(n^3\right)$.

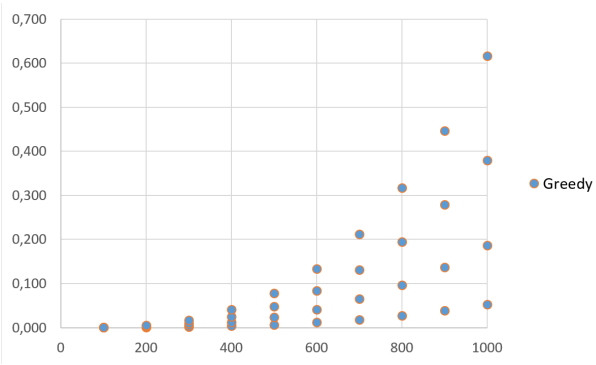


Figure A.3: Scaling diagram for the greedy algorithm on the benchmark

Let us verify whether it is correct by drawing the scaling diagram in a logarithmic scale:

$$T_A = \beta n^{\alpha} \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

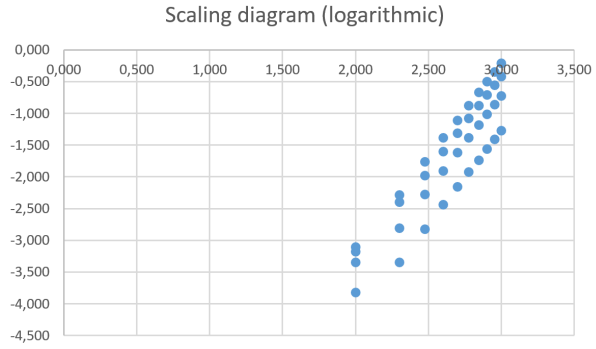


Figure A.4: Scaling diagram in logarithmic scales for the greedy algorithm on the benchmark

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, 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[3]:

$$\frac{LB - f_A(x)}{LB} \leq \delta_A(x) = \frac{f^* - f_A(x)}{f^*} \leq \frac{UB - f_A(x)}{UB}$$

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

[3]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 A.1: Results of the basic greedy algorithm

A.4 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 A.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.

A.5 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.

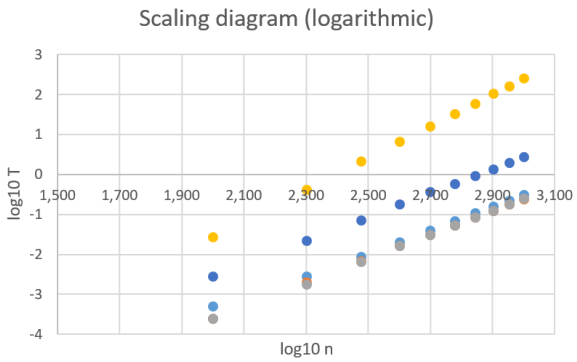


Figure A.5: Scaling diagram for the four greedy algorithms variants (and the stingy one) on the benchmark

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 A.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.

A.6 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 A.6 for the comparison, which clearly confirms the strict dominance of the last heuristic on the previous ones, the probabilistic dominance (at least on the given benchmark) of the stingy heuristic on the greedy ones, whereas the three greedy heuristics are more or less equivalent (with a slight predominance of the farthest point heuristic, especially concerning the maximum gap).

The diagram also shows a very strange phenomenon: the farthest pair heuristic has the same diagram as the basic greedy heuristic. In fact, they not only have the same distribution, but exactly the same results. Is this an unpredicted property of the two algorithms? It does not seem to be necessarily so, and yet this is what can be empirically observed.

The reason is very peculiar and depends on the specific structure of the benchmark. Hence, it is not instructive for the *MDP* in general, but for the need to keep an open eye on the generation of the benchmark. We have extracted the distance values from $\{1,\ldots,100\}$. An instance with n points has exactly $n(n-1)/2$ independent distance values, if we take into account the assumptions made in the introductory section. This means that each point i is very likely to admit at least another point j at distance $d_{ij} = 100$. In particular, it is very likely that point 1 admits such another point. But this implies that the farthest pair heuristic will chose a pair including point 1. On the other hand, the basic heuristic, after choosing point 1 will certainly proceed choosing the farthest point from it. Therefore, the two heuristics are very likely to start with the same pair, and, being deterministic, to proceed in the same way and obtain the same final result. This is not intrinsic in the two algorithms, but it depends on the structure of the benchmark instances.

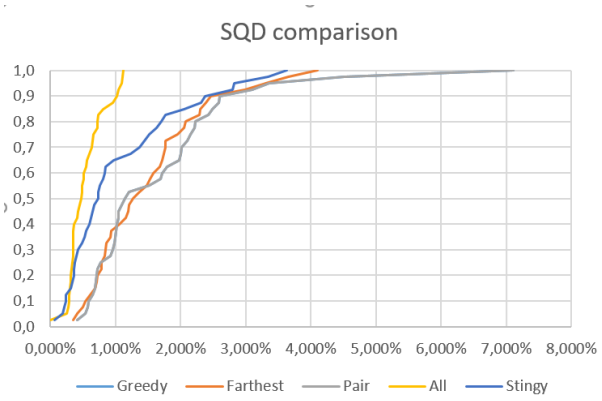


Figure A.6: Solution Quality Distribution diagram for the four greedy algorithms variants (and the stingy one) on the benchmark

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 A.7.

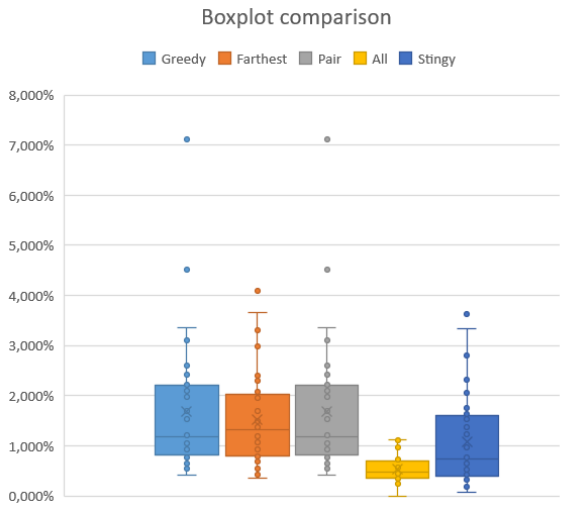


Figure A.7: Boxplots for the four greedy algorithms variants (and the stingy one) on the benchmark

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 A.8)? Is it different for different algorithms?

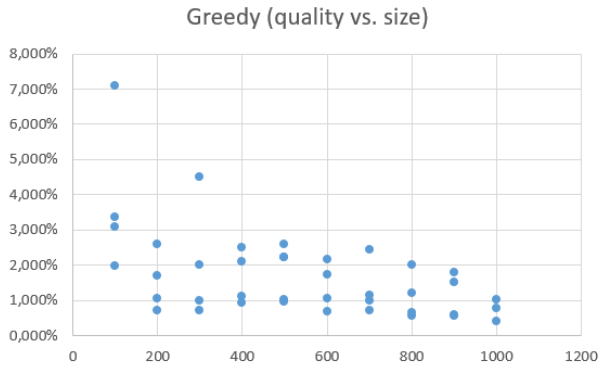


Figure A.8: Scaling diagram for the quality of the greedy algorithm with respect to the size of the instances of the benchmark

A.7 Roll-out heuristic

TO BE DONE

APPENDIX B

Laboratory on Constructive Metaheuristics

B.1 Introduction

This chapter discusses the application of constructive metaheuristics to the *Maximum Diversity Problem* (*MDP*). Constructive metaheuristic try to improve the results of a basic constructive heuristic by running it repeatedly with the introduction of mechanisms that modify its final result. In the end, of course, the algorithm returns the best of the solutions found during the process. The main mechanisms used by metaheuristic algorithms to enhance a constructive heuristic are:

1. the use of different selection criteria, typical of *multi-start* algorithms;
2. the use of *random choices*, typical of *GRASP*;
3. the use of *memory*, typical of the *Ant System*

In the following, we will implement *GRASP* and *Ant System* algorithms for the *MDP*, based on the constructive heuristics discussed in the previous chapter[1]. In the literature, these two approaches require the introduction of exchange procedures to improve the solutions generated by the constructive mechanism. In order to focus on the latter, however, we will avoid them.

Correspondingly, the `main` function allows to choose from the command line which of the two algorithms to apply (with option `-grasp` for the *GRASP* and `-grasp` for the *Ant System*) and to provide the numerical values of the following parameters:

- for the *GRASP* heuristic:
  - the total number of iterations ℓ
  - the randomness parameter μ
- for the *Ant System* heuristic:
  - the total number of iterations ℓ
  - the randomness parameter q
  - the oblivion parameter ρ

[1] Presently, the chapter only includes the *GRASP* algorithm.

plus the seed required to initialise the pseudorandom number generator.

The other operations (loading the data, allocating and deallocating the data and the solution, determining the computational time and printing the results on the screen) are the same as for the constructive heuristics, except for the fact that also the parameters are printed, so that the report keeps trace of how each single solution was obtained to help guarantee the reproducibility of the results[2]

```
parse_command_line (argc,argv,data_file,algo,&iterations,&mu,&q,&rho,&seed);

load_data(data_file,&I);
//print_data(&I);

create_solution(I.n,&x);

inizio = clock();
if (strcmp(algo,"-grasp") == 0)
  grasp(&I,&x,iterations,mu,&seed);
else if (strcmp(algo,"-as") == 0)
  ant_system(&I,&x,iterations,q,rho,&seed);

fine = clock();
tempo = (double) (fine - inizio) / CLOCKS_PER_SEC;

printf("%s ",data_file);
for (arg = 2; arg < argc; arg++)
  printf("%s ",argv[arg]);
printf("%10.6lf ",tempo);
print_solution(&x);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

B.2 Greedy Randomized Adaptive Search Procedure

The *Greedy Randomized Adaptive Search Procedure* (*GRASP*) is a development of the classical *semigreedy* algorithm, that we will actually implement. Its basic idea is to modify the scheme of constructive algorithms by replacing the deterministic choice of the element that provides the best value of the selection criterium

$$i^* := \arg \min_{i \in \Delta^+(x)} \varphi_A(i,x)$$

with a stochastic choice $i^*(\omega)$. This requires to define a probability distribution on set $\Delta^+(x)$, that should be biased so as to favour the best elements over the worst ones:

$$\varphi_A(i,x) \leq \varphi_A(j,x) \Leftrightarrow \pi_A(i,x) \geq \pi_A(j,x)$$

[2]Of course, the code could change and yield different results for the same parameter values, but this should no longer be the case when the implementation has reached a sufficiently stable status.

The following pseudocode provides the basic scheme of *GRASP* for maximisation problems, adapted to the specific application to the *MDP* by translating the termination condition into a check on the cardinality of the current subset, by returning the last visited subset (as it is the only feasible one) and by replacing the search for a minimum cost solution with the search for a maximum value one. The search procedure after each constructive phase is also neglected:

Algorithm 23 GRASP

```
1:  procedure GRASP(I, ℓ, μ)
2:      x* := ∅; f* := 0;                          ▷ Best solution found so far
3:      for l := 1 to ℓ do
4:          x := ∅;
5:          while |x| < k do                       ▷ Randomised constructive procedure
6:              φ_i := Σ_{j∈x} d_ij for all i ∈ P \ x;   ▷ φ_i := (f(x∪{i}) − f(x))/2
7:              i* := BiasedRandomExtraction(P \ x, φ, μ);
8:              x := x ∪ {i*};
9:          end while
10:         if f(x) > f* then
11:             x* := x; f* := f(x);
12:         end if
13:     end for
14:     return (x*, f*);
15: end procedure
```

The algorithm performs ℓ iterations, where ℓ is a number chosen by the user, based on the available time. It is easy to recognise in the inner **while** a modified version of the basic constructive heuristic presented in the previous chapter. The algorithm still computes all values of $\varphi_A(i,x)$, but, instead of choosing the largest one, it selects one at random based on a biased probability distribution (with parameter μ) that favours the largest ones. At the end of each iteration, the current solution possibly updates the best known one.

The scheme is quite similar to the one used for the greedy "try-all" heuristic, as it requires to create and iteratively fill and empty a current solution, while updating the best known one, that will be returned in the end. We just replace the function `best_point_to_add(px,pI)` used in the basic greedy heuristic, with a function `biased_random_point_to_add(px,pI,pseed,pmu)` that performs the evaluation of the selection criterium $\varphi_A(i,x)$ based on the current solution `x` and on the instance `I`, the extraction of a pseudorandom number depending on the `seed` and the biased stochastic selection of a new point `i` to add based on the parameter `mu` provided by the user.

```
create_solution(pI->n,&x);
for (iter = 1; iter <= iterations; iter++)
{
  while (get_card(&x) < pI->k)
  {
    i = biased_random_point_to_add(&x,pI,mu,pseed);
    add_point(i,&x,pI);
  }

  if (x.f > px->f) copy_solution(&x,px);
```

```
    clean_solution(&x,pI->n);
  }
  destroy_solution(&x);
```

B.2.1 Choice of the basic constructive heuristic

In the previous chapter, we have considered four alternative (though very similar) constructive heuristics, and a destructive one. we now discuss which of the heuristics should be adopted as the core of the *GRASP* approach.

We will leave aside the destructive heuristic, because it is conceptually different from the other ones, though it would indeed be interesting to compare the constructive and destructive approaches in a metaheuristic allowing to give them equal time[3]. We will also exclude the try-all heuristic, that was the best-performing one, but also the slowest, and in a sense it is already a sort of multi-start metaheuristic, using the initial point as a parameter whose value changes at every iteration. Since a metaheuristic is intrinsically less efficient than the basic heuristic on which it is based, we prefer to choose a simple and fast mechanism, rather than a slow and complex one, at least for a first experiment. There is always time to introduce complications and refinements, if justified by theory or by experience.

The basic heuristic had a strong drawback: it could only generate solutions including point 1. Is this drawback still present in a randomized version? The answer depends on the probability distribution selected. We remind that at the first step the selection criterium is equal to zero for all points. Therefore:

- a scheme based on a *Heuristic Biased Stochastic Sampling* (*HBSS*) would choose any point, but it would assign larger probabilities to the first points;
- a scheme based on a *Restricted Candidate List* (*RCL*) would choose with uniform probability one of the first points.

In the first case, all solutions can be obtained, but there is a bias towards those including the points with small indices, and such a bias is not justified by any good reason. In the second case, the points with larger indices could even be impossible to reach for some instances (that depends also on the values of the distance function). Since for the sake of simplicity, we are going to test only a *RCL* approach, the basic greedy heuristic is not a good choice, unless with some additional correction.

The farthest-point and the farthest-pair heuristic still introduce a bias, or a deterministic advantage, in favour of points with a large total distance, or pairs of very distant points. Such a bias is less unreasonable, but still not provably justified. Moreover, in the case of point pairs, since the number of possible distances is not huge, several pairs of points could have the same distance, and therefore the discrimination between them would end up being based on their indices, and that would not be reasonable.

A very simple idea to avoid an index-based bias at the first iteration of the procedure could be to select the first point at random with uniform probability. At the following step[4], the choice would be stochastically biased in favour of the point that is farthest from a first one selected uniformly at random. That is similar to the farthest-pair heuristic, but different in that at least one of the two indices would be selected at random, without any index-based bias. For the sake

[3] That's for a future laboratory, but it could be a good exercise.

[4] The following discussion is unrevised brainstorming: I do not expect it to be very clear, or even correct, but I think that the elements discussed are indeed relevant for the behaviour of the algorithm.

of simplicity, we will apply this idea. This means that when x is empty the procedure directly selects one of the points with probability $\pi_i = 1/n$ without computing the selection criterium (that would be trivially equal to zero for all points $i \in P$).

When x is not empty, we compute the selection criterium for all external points and proceed to a biased random extraction from the available alternatives. Instead of simply saving the maximum of these values, we save them progressively in a vector `phi` and the corresponding points in a vector `P`, returning the final length `num` of the two vectors, that corresponds to the number of possible extensions $|\Delta^+(x)| = P \setminus x$[5].

```
if (get_card(px) == 0)
  i = get_point(rand_int(1,pI->n,pseed),pI);
else
{
  P = point_alloc(pI->n+1);
  phi = int_alloc(pI->n+1);

  num = compute_selection_criterium(px,pI,P,phi);
  i = biased_random_extraction(P,phi,num,mu,pseed);

  free(P);
  free(phi);
}

return i;
```

The selection criterium is still the value of the objective, that is

$$\varphi_A(i,x) = f(x \cup \{i\}) = \sum_{j \in x \cup \{i\}} \sum_{k \in x \cup \{i\}} d_{jk}$$

replaced for the sake of efficiency by half of its variation $\delta f(x,i) = (f(x \cup \{i\}) - f(x))/2$

$$\delta f(x,i) = \sum_{j \in x} d_{ji}$$

It is therefore computed with the function `dist_from_x(i,px,pI)` that was implemented in the previous chapter.

```
cnt = 0;
for (i = first_point_out(px); !end_point_list(i,px); i = next_point(i,px))
{
  cnt++;
  P[cnt] = i;
  phi[cnt] = dist_from_x(i,px,pI);
}

return cnt;
```

[5]Strictly speaking, this value is therefore already known, so that it would not be necessary to retrieve it from the computation.

B.2.2 Pseudorandom number extraction

In order to generate random numbers, we exploit a classical pseudorandom number generator (the `ran1` generator described in the *Numerical recipes in C*). This is a function that receives in input a *seed*, that is a negative integer number, modifies that number (this is why the seed is passed by reference) and returns in output a real number ω that tends to assume a uniform distribution in the range $[0;1]$ as the function is repeatedly called. The first value of the seed is selected by the user, fed to the algorithm in the command line and determines the overall sequence of numbers generated. This is why the numbers of the sequence are denoted as *pseudorandom*. The ability to generate the same sequence in all runs is fundamental for the repeatability of the approach and the reproducibility of the results. In short, it is a basic condition for the scientific investigation of the problem.

B.2.3 Biased point selection

Given the pseudorandom number ω, it is necessary to determine the corresponding point in $\Delta^+(x) = P \setminus x$ with a biased scheme that favours the points with larger values of ϕ_i. This mechanism depends on the probability distribution adopted. In the following, for the sake of simplicity, we will adopt a *value-based Restricted Candidate List* (*RCL* scheme, in which:

1. a real parameter $\mu \in [0;1]$ is used to fix an intermediate threshold $\bar{\varphi}(x,\mu)$ between the minimum and maximum values available for ϕ_i;

2. the points i such that ϕ_i is better than (i.e., above) the threshold enter the *RCL*;

3. a point is selected from the *RCL* with uniform probability.

Other schemes commonly adopted in *GRASP* heuristics employ a cardinality-based *RCL*, a linearly decreasing or an exponentially decreasing probability profile on all external points. All of them assign decreasing probabilities to the possible choices $i \in \Delta^+(x)$ sorted by nondecreasing values of ϕ. In short, if i_r is the element in position $r = 1,\ldots,|\Delta^+(x)|$ in the ranking, $\phi(i_1,x) \geq \phi(i_2,x) \geq \ldots \geq \phi(i_{|\Delta^+(x)|},x)$. Moreover, the typical schemes adopted in *GRASP* heuristics define probabilities based on the ranking of the choices, and not on the absolute values of the selection criterium, that is $\phi(i_r,x)$ can be expressed as a function of r and $|\Delta^+(x)|$.

Value-based *RCL*

This scheme computes an adaptive threshold depending on the values of the selection criterium on the available extension.

$$\bar{\varphi}(x,\mu) = (1-\mu)\,\varphi_{\min} + \mu\,\varphi_{\max}$$

where

$$\varphi_{\min}(x) = \min_{i \in \Delta_A^+(x)} \varphi_A(i,x) \quad \text{and} \quad \varphi_{\max}(x) = \max_{i \in \Delta_A^+(x)} \varphi_A(i,x)$$

and defines a *RCL* as

$$\mathrm{RCL}(x,\mu) = \{i \in \Delta^+(x) : \varphi(i,x) \geq \bar{\varphi}(x,\mu)\}$$

Then, it assigns the elements keeping under the threshold a uniform probability, and the following ones a zero probability:

$$\pi_{i_r} = \begin{cases} \dfrac{1}{|\mathrm{RCL}(x,\mu)|} & \text{if } r \leq |\mathrm{RCL}(x,\mu)| \\ 0 & \text{if } r > |\mathrm{RCL}(x,\mu)| \end{cases}$$

The parameter $\mu \in [0;1]$ tunes the randomness of the choice, with $\mu = 0$ yielding a deterministic heuristic and $\mu = 1$ a random walk.

In order to implement this scheme, the procedure `biased_random_extraction` scans the vector `phi` a first time to determine its minimum and maximum values $\varphi_{\min}$ and $\varphi_{\max}$. Then, it computes the threshold based on parameter μ that discriminates from the other points the elements of the *RCL*, according to condition[6]

$$\varphi_i \geq (1-\mu)\,\varphi_{\max} + \mu\,\varphi_{\min}$$

Scanning again the *RCL* allows to determine the number of its elements and to move them at the beginning of the vector of points `P`. For the sake of efficiency, the points are copied overwriting the previous elements of the vector, because there is no need to keep the whole original content: only the elements of the *RCL* are necessary.

Identification of the selected point

Now, in order to extract one of them with uniform probability, it is enough to compute $\lceil \omega\,|\mathrm{RCL}(x,\mu)| \rceil$ and take the corresponding element of vector `P`. In *RCL*-based schemes, it is enough to multiply ω by the size of the *RCL* and round up the result. This provides the position of the point in the list, and therefore allows to access it directly.

The other schemes mentioned above require to first translate the pseudorandom number ω into a ranking position, and then to identify the point corresponding to that position in a sorted vector. Let us focus on the first phase. It is always possible to identify the ranking position that corresponds to ω by summing the probabilities associated to the subsequent rankings, and stopping when the sum reaches ω: the corresponding position is the required one. In practice, however, it is usually not necessary to perform this sum explicitly, since the structure of the values allows to compute the correct ranking more quickly, just as in the case of the *RCL* it was enough to multiply ω by the size of the list[7]

After computing the ranking position, one must still identify the corresponding point according to the order by nondecreasing values of ϕ. It is never required to fully sort vector `phi`: the extraction of the k-th largest element from a vector can in fact be performed in linear time applying suitable algorithms. We do not give details here since we limit our experiments to the *RCL* scheme.

```
phiMin = INT_MAX;
phiMax = -1;
for (cnt = 1; cnt <= num; cnt++)
```

[6]The condition is complementary to the one given in the slides, because the selection criterium must be maximised, but μ still measures randomness.

[7]I need to work on this point, that is not clearly discussed in any paper or textbook at the best of my knowledge. I am pretty sure that a linear probability profile allows to give a closed-form quadratic expression of the cumulated probability in each ranking position r, and therefore to find the position corresponding to ω in constant time solving a second-order equation. For the exponential profile, something more sophisticated is required.

```
{
  if (phi[cnt] < phiMin) phiMin = phi[cnt];
  if (phi[cnt] > phiMax) phiMax = phi[cnt];
}

barphi = (1-mu) * phiMax + mu * phiMin;

RCLsize = 0;
for (cnt = 1; cnt <= num; cnt++)
  if (phi[cnt] >= barphi)
  {
    RCLsize++;
    P[RCLsize] = P[cnt];  /* overwriting P for the sake of efficiency */
  }

cnt = rand_int(1,RCLsize,pseed);

return P[cnt];
```

B.2.4 Empirical evaluation

We can now evaluate the performance of the *GRASP* heuristic. Contrary to what we have done in the previous chapter, having acquired a certain understanding of what the heuristics are doing, we will try to avoid producing meaningless diagrams.

Computational time analysis

An *a priori* worst-case asymptotic analysis of the computational time can be based on the similar analysis made for the basic deterministic greedy heuristic. First of all, the constructive heuristic is run for a given number of iterations ℓ. This number is a relevant parameter, that could constant and totally unrelated to the size of the problem, but could also be chosen depending on it, with the idea that larger instances could require more iterations to be explored properly, or vice versa that larger instances allow less iterations because thery require a longer time for each run, and the overall time is limited. In general, therefore, the expression of the time complexity will include ℓ. The basic deterministic heuristic required time $O(nk^2)$ for each run. The randomised version requires additional time for the generation of the pseudorandom number (that can be assumed as constant), for the identification of the minimum and maximum values of the selection criterium and the construction of the *RCL* (that can be assumed as linear), for the biased random selection of the new point (constant time). Consequently, we can estimate an additional linear time per each of the k iterations of the constructive method: the resulting $O(nk)$ term is dominated asymptotically. Other additional terms are given by the comparison of each of the ℓ solutions obtained with the best known one and, possibly, the update of the latter. All these terms are asymptotically dominated, but they could have a perceivable influence on the empirical evaluation.

For the first experiment, we set $\ell = n$. The choice is clearly arbitrary, but it is motivated by the idea to allow each of the n points to be chosen with a reasonable probability as the starting point of the constructive heuristic. It also aims to obtain a computational time comparable to that of the "try-all" heuristic, which also had n repetitions of the basic constructive scheme. This

should allow to better estimate the impact of the additional operations required by the randomisation (not the maintenance of the best known solution, that occurs also in the deterministic heuristic).

Figure B.1 reports the *scaling diagram* for the whole benchmark, obtained setting $\mu = 0.1$. It can be noticed, however, that according to the theoretical analysis parameter μ should have very little influence on the computational time. The detailed reports show a limited slow-down as μ increases, but the difference is not significant. The diagram shows the expected increase of the computational time with size, and its logarithmic version in Figure confirms its polynomiality. The $O\left(\ell n k^2\right)$ theoretical estimate, with $\ell = n$ and $k \propto n$, suggests an overall $O\left(n^4\right)$ complexity, that is confirmed by the linear interpolation:

$$T_A = \beta n^\alpha \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

with $\alpha \approx 3.969$ and $\beta \approx 3.4 \cdot 10^{-10}$.

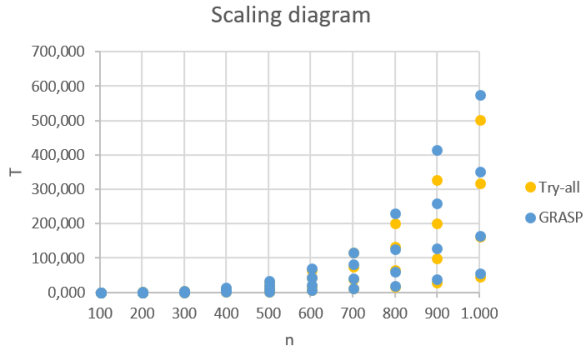


Figure B.1: Scaling diagram for the greedy algorithm on the benchmark

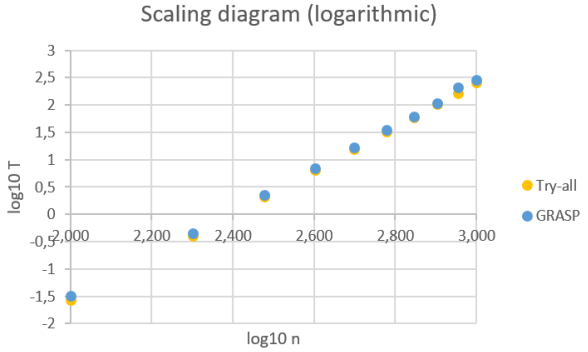


Figure B.2: Scaling diagram in logarithmic scales for the greedy algorithm on the benchmark

The two figures report also the scaling diagram of the greedy "try-all" heuristic, that was expected to have a very similar performance with respect to the computational time. In fact, the two profiles are very similar, with the *GRASP* heuristic only slightly higher, confirming that the additional operations to randomise the selection of points affect very little the overall complexity.

Solution quality analysis

Without forgetting that the benchmark is rather small and specific, we can now draw the *SQD* diagram (see Figure **??**), to compare the different parameter tunings with one another. We consider the following tunings: $\mu \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$, after some preliminary experiments showed that larger values provided worse results. The same diagram can be used to compare *GRASP* with the algorithms already developed. In particular, the figure shows the profile of the "try-all" heuristic, that is the most similar one in terms of behaviour (applying a sequence of n different constructive heuristics) and computational time. The diagram is not very clear, but the versions with larger values of the randomness coefficient μ seem to perform slightly worse (the trend becomes clearer for larger values). Indeed, the performance of the "try-all" heuristic is similar to that of the smaller values of μ and better than the other ones. This is rather disappointing, and poses the question whether the values tested are too large and should be reduced. In order to be sure, one would need to check the typical length of the *RCL* during the search. A rough estimate, based on the (unproved, but not unreasonable) assumption that the values of the selection criterium are uniformly distributed between $\varphi_{\min}$ $\varphi_{\max}$ is that the typical size decreases from μn to $\mu(n-k)$. For $\mu = 0.01$, n ranging from 100 to 1 000 and k ranging from $0.1n$ to $0.4n$, this corresponds to sizes ranging from 1 to 9, that do not seem so large.

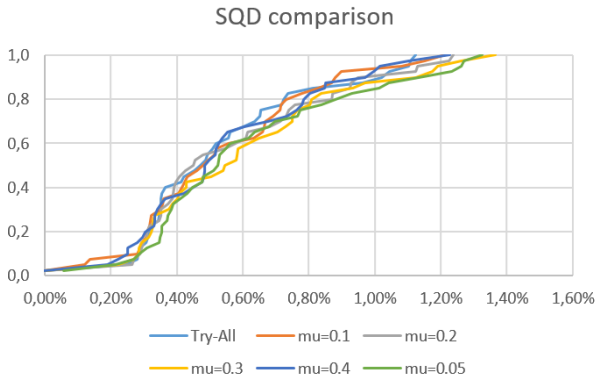


Figure B.3: Solution Quality Distribution diagram for the *GRASP* algorithm (with $\mu \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$) and the greedy "try-all" heuristic

The boxplots reported in Figure B.4 provide a similar, perhaps slightly clearer, intuition. It should anyway be noticed that the *GRASP* heuristic can be prolonged for more than $\ell = n$ iterations, probably improving the final results, whereas the deterministic heuristic cannot. As well, we could experiment with shorter runs, to determine whether the results obtained actually could require a lower number of iterations. Anyway, the results remain unpromising.

Statistical tests We now apply Wilcoxon's test to determine whether it can discriminate between the results of different parameter tuning of *GRASP* and the deterministic competitor. The results are:

- for $\mu = 0.01$

```
W+ = 338.50, W- = 402.50, N = 38, p <= 0.6478
```

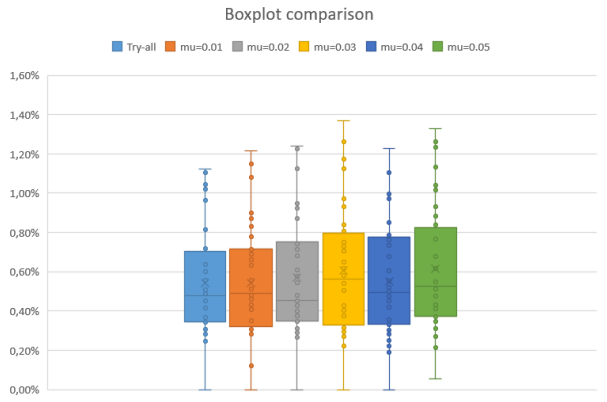


Figure B.4: Boxplots for the *GRASP* algorithm (with $\mu \in \{0.1, 0.2, 0.3, 0.4\}$) and the greedy "try-all' heuristic'

- for $\mu = 0.02$

```
W+ = 430.50, W- = 310.50, N = 38, p <= 0.3882
```

- for $\mu = 0.03$

```
W+ = 523.50, W- = 256.50, N = 39, p <= 0.06345
```

- for $\mu = 0.04$

```
W+ = 455.50, W- = 324.50, N = 39, p <= 0.3644
```

- for $\mu = 0.05$

```
W+ = 634.50, W- = 145.50, N = 39, p <= 0.0006617
```

In short, the difference, at first not statistically significant, tends to become more significant as μ increases (with an exception for $\mu = 0.04$. The only comparison that seems to exhibit a true dominance is the one between the deterministic heuristic and $\mu = 0.05$. The message is clearly not to exceed with randomicity.

Influence of the random seed An aspect that must be analysed when using random steps is the influence that randomness has on the final result obtained. If we run the algorithm a single time, in fact, the quality of the results achieved could be easily due to a lucky, or unlucky, choice of the random seed. In order to estimate the role of randomness, we should run the algorithm for several times, with different random seeds, and compare the results thus obtained. A simple index is given by the average quality of the solution with respect to a sufficiently large number of runs (at least 10, possibly more), but other indices of distribution are certainly relevant: the maximum and minimum values obtained, or the medians and quartiles. The description is very

similar in principle to that we have given with respect to the benchmark instances (numerical indices, boxplots, *SQD* diagrams), but it considers single instances and variable seeds, instead of a single seed and random instances.

In order to give an idea of this kind of investigation, we select a single instance, that we consider as significant for some reason, and we run the algorithm for a given number of times with different seeds. We will consider the instance `n0600k060`, because it is the instance on which the *GRASP* algorithm with $\mu = 0.01$, that is the best performing one on average (though with nonsignificant differences with other ones) obtains the largest gap: $\delta_A(I) = 1.21\%$ (see the maximum of the second boxplot in Figure B.4. The question investigated is whether this bad result was typical or derived from a particularly unlucky, or particularly lucky, choice of the random seed. In order to establish this, since the algorithm takes about 5 seconds to solve the instances, we run it 100 times with different random seeds, ranging from -1 to -100. Figure B.5 provides the *SQD* with respect to the random seed. The gap δ is never huge, but indeed it varies in a rather large range (between 0.6% and 1.4%). The red line in the picture, that corresponds to the single run of the previous phase of experiments, suggests that its result was indeed rather unlucky. This suggests that the *GRASP* algorithm (at least on this instance) is rather unstable, that its results could be in practice better (but also worse) than the ones discussed above, that were obtained in a single (not necessarily representative) run, and that the conclusions drawn from such results should be handled with much care.

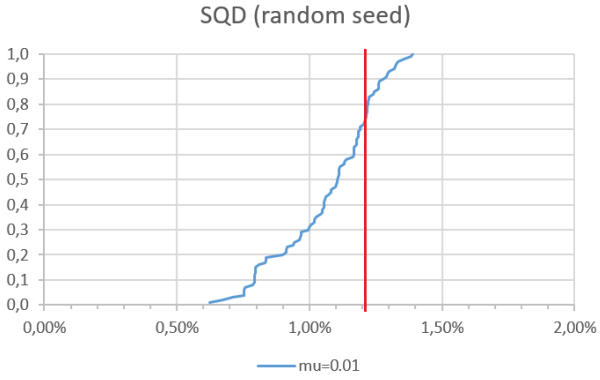


Figure B.5: Solution Quality Distribution diagram for the *GRASP* algorithm with $\mu = 0.01$ on instance `n0600k060` with 100 different random seeds: the red line corresponds to the single run of the previous phase of experiments.

B.3 Ant System

The *Ant System* (*AS*) is a development of the classical *cost perturbation* algorithm, as well as of the *semigreedy* algorithm. Its basic idea is to modify the scheme of constructive algorithms by replacing the deterministic choice of the element that provides the best value of the selection criterium with a stochastic choice, influenced by additional information provided by the memory of previously found solutions. In addition, the *AS* considers a *population* of algorithms that work in parallel, iteration by iteration. Among the several variants of *AS*, we are going to implement the one that:

- defines the *visibility* of a point $i \in P \setminus x$ as the selection criterium used in the constructive and the *GRASP* heuristic, that is
$$\eta(i,x) = \varphi(i,x) = \sum_{j \in x} d_{ji}$$
- maintains in a suitable vector a *trail* function $\tau(i)$ that depends only on the point i to be added, and is progressively updated during the execution;
- combines visibility and trail by multiplying them, as they both tend to associate larger values to better options;
- tunes the randomness of the choice with a parameter q, so that the choice is made selecting with probability $1-q$
$$i^* = \arg\max_{i \in P \setminus x} \varphi(i,x)\,\tau(i)$$
and with probability q a random point with probability distribution
$$\phi_i = \frac{\varphi(i,x)\,\tau(i)}{\sum_{j \in P \setminus x} \varphi(j,x)\,\tau(j)}$$
- applies a local update to the trail to diversify the search after each individual has built a solution;
- applies a global update to the trail to intensify the search on the points that belong to the best known solution found in the whole process.

The following pseudocode provides the scheme of this variant of the *AntSystem* adapted to the *MDP* as already done for *GRASP*. Also in this case the search procedure that should be run to improve the solutions built is neglected:

The algorithm performs ℓ iterations, in each of which it generates h different solutions; ℓ and h are numbers chosen by the user, based on the available time. It is easy to recognise in the inner **while** a modified version of the basic constructive heuristic presented in the previous chapter. The algorithm still computes all values of $\varphi_A(i,x)$, but, instead of choosing the largest one, it selects one at random based on a biased probability distribution (with parameter μ) that favours the ones with largest values of φ and τ. At the end of each iteration, the current solution possibly updates the best known one.

Algorithm 24 AntSystem

procedure ANTSYSTEM(I, ℓ, q, ρ)
  $x^* := \emptyset$; $f^* := 0$; ▷ Best solution found so far
  $\tau_i = \tau_0$ for all $i \in P$;
  for $l := 1$ **to** ℓ **do**
    for $g := 1$ **to** h **do**
      $x := \emptyset$;
      while $|x| < k$ **do** ▷ Randomised constructive procedure
        $\varphi_i := f(x \cup \{i\}) - f(x)$ for all $i \in P \setminus x$;
        $i^* :=$ BiasedRandomExtraction$(P \setminus x, \varphi, \tau, \mu)$;
        $x := x \cup \{i^*\}$;
      end while
      if $f(x) > f^*$ **then**
        $x^* := x$; $f^* := f(x)$;
      end if
      $\tau :=$ LocalTrailUpdate(x, τ, ρ);
    end for
    $\tau :=$ GlobalTrailUpdate(x^*, τ, ρ);
  end for
  return (x^*, f^*);
end procedure

APPENDIX C

Laboratory on Exchange heuristics

asdasdas

C.1 Introduction

This chapter discusses the application of exchange heuristics to the *Maximum Diversity Problem* (*MDP*). Exchange heuristic start from a given feasible solution $x^{(0)}$ (typically obtained with a constructive heuristic, or metaheuristic, or a random generation process) and try to improve the current solution x iteratively by adding a suitable subset A and deleting a suitable subset D of elements of the ground set. Of course, A consists of external elements ($A \subseteq B \setminus x$) and D of internal ones ($D \subseteq x$). The possible pairs of subsets are determined by a rule that takes the form of a *neighbourhood* function $N : X \to 2^X$, associating each feasible solution $x \in X$ with a subset of feasible neighbour solutions $N(x)$. The choice of the *incumbent*, that is the neighbour solution that replaces the current one, is done optimising a suitable selection criterium, that nearly always is the objective function value.

In the following, we will consider some alternative initialisation procedures (namely, the farthest point and the try-all constructive heuristics and a purely random generation). We will also adopt the most natural neighbourhood for the *MDP*, that is the *single-swap neighbourhood* N_{S_1}, which includes all the subsets obtained from x deleting a single element i and adding a single element j:

$$N_{S_1}(x) = \left\{x' = x \setminus \{i\} \cup \{j\} \text{ with } i \in x, j \in P \setminus x\right\}$$

Notice that $N_{S_1}(x) = N_{H_2}(x)$, that is, it coincides with the collection of feasible subsets having Hamming distance equal to 2 from x. However, the collection of all subsets at Hamming distance equal to 2 also includes the ones obtained adding or deleting two points, which are unfeasible. The single-swap neighbourhood, on the contrary, automatically satisfies the cardinality constraint that characterises the *MDP*, and this implies the strong advantage that the feasibility of the subset obtained with any swap operation is guaranteed *a priori* and needs not be verified. The exploration of the neighbourhood, therefore, simply consists in the computation of the objective function for each neighbour solution.

Thanks to the cardinality constraint, and to the lack of other complicating constraints, neighbourhood N_{S_1} always includes exactly $k(n-k)$ solutions, and this induces a strong relation between the number of neighbourhood explorations and the computational time (at least, if the neighbourhood is fully explored).

Finally, concerning the selection criterium, we will adopt the objective function, thus implementing the basic exchange heuristic known as *steepest ascent* (for maximisation problems as the *MDP*). We will discuss its theoretical and empirical computational complexity, and we

will improve it with a standard trick to allow the evaluation of quadratic objective functions in cardinality-constrained problems. We will compare the results of the different initialisation procedures and we will tune the size of the neighbourhood with the adoption of the *first-best* exploration strategy as opposed to the *global-best* one.

The `main` function, then, allows to choose from the command line which of the three initialization procedures to apply (with option `-gp` for the *farthest-point*, `-ga` for the try-all heuristic and `-r` followed by a negative integer seed for the random initialisation), and which of the two neighbourhood exploration strategies to apply (with option `-gb` for the *global-best* and `-fb` for the *first-best* strategy). The steepest ascent heuristics also returns the number of neighbourhood explorations performed, because we are going to investigate the influence of the exploration stategy on this value and its relation with the computational time. Apart from printing the number of iterations, all other operations are the same introduced for the previous heuristics.

```
parse_command_line(argc,argv,data_file,init_algo,visit_strategy,&seed);

load_data(data_file,&I);

create_solution(I.n,&x);

start = clock();
if (strcmp(init_algo, "-gf") == 0)
  greedy_farthest(&I, &x);
else if (strcmp(init_algo, "-ga") == 0)
  greedy_tryall(&I, &x);
else if (strcmp(init_algo, "-r") == 0)
  generate_random_solution(&I, &x, &seed);
steepest_ascent(&I,&x,visit_strategy,&niter);
end = clock();
tempo = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6f ",tempo);
printf("%8d ",niter);
print_sorted_solution(&x,I.n);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

C.2 The steepest ascent heuristic

The *steepest ascent* heuristic chooses the following solution from the neighbourhood of the current one by optimising a selection criterium $\phi(x,i,j)$ that is simply the value of the objective function $f(x \cup \{i\} \setminus \{j\})$ or, to be more precise, half of its variation:

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = \sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

This implies the following adaptation to the *MDP* of the general scheme:

Algorithm SteepestDescentMDP$\left(I, x^{(0)}\right)$

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

Stop := *false*;

While Stop = *false do*

$\tilde{x} := \arg \max_{i \in x, j \in P \setminus x} \left(\sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij} \right)$;

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

EndWhile;

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

The implementation of this algorithm is nearly straightforward. The main difference is that, for the sake of efficiency, the procedure `explore_neighbourhood` that selects the incumbent returns a pair of points (i^*, j^*) to be exchanged, and the resulting variation of the objective, instead of a whole solution $\tilde{x}$. Therefore, if the incumbent improves the current solution (that is, the variation is negative), the update operation $x := \tilde{x}$ is obtained performing the exchange suggested with procedure `swap_points`.

```
*pniter = 0;
do
{
  explore_neighbourhood(px,pI,visit_strategy,&p_in,&p_out,&delta_f);
  if (delta_f > 0)
  {
    (*pniter)++;
    swap_points(p_in,p_out,px,pI);
  }
} while (delta_f > 0);
```

It could be remarked that swapping points j and i is equivalent to deleting point j and adding point i, so that we do not actually need an additional procedure. However, implementing this procedure separately has the advantage to avoid any instruction that is useless when the two operations must be performed together (to give a trivial example, the cardinality of the solution remains unvaried, instead of being decremented and incremented).

The exploration of the neighbourhood trivially consists in two nested loops, as j scans the current solution x and i scans its complement, taking advantage of the corresponding lists. For each pair of solutions, the procedure estimates the variation of the objective function $\delta f(x, i, j)$ calling a suitable procedure `evaluate_exchange` and saves the best exchange and the corresponding variation of the objective.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px); p_in = next_point(p_in,px
  for (p_out = first_point_out(px); !end_point_list(p_out,px); p_out = next_point(p
  {
    delta_f = evaluate_exchange(p_in,p_out,px,pI);
    if (delta_f > *pdelta_f)
    {
      *pdelta_f = delta_f;
```

```
      *pp_in = p_in;
      *pp_out = p_out;
    }
  }
```

The procedure that evaluates each single exchange does not physically perform it. *Do not perform moves only to evaluate them* is a specific application of a general fundamental principle in the design of heuristic algorithms: *avoid all useless operations*. It simply computes

$$\sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

as follows[1].

```
delta = 0;

delta  = dist_from_x(p_out,px,pI);
delta -= dist_from_x(p_in,px,pI);
delta -= pI->d[get_index(p_in,pI)][get_index(p_out,pI)];

return delta;
```

C.2.1 Time complexity estimation

The computational complexity of the steepest ascent heuristic derives from three main sources:

1. the number of neighbourhood explorations $t_{\max}$ performed to reach the local optimum in which the search terminates;

2. the number of neighbour solutions (or, in general, subsets) whose objective value (and, in general, feasibility) is evaluated;

3. the computational time required to evaluate the objective value (and, in general, the feasibility) of each neighbour solution (or subset).

The first term is in general unknown and hard to estimate (unless for upper estimates such as the total number of feasible solutions, that are very loose). For the single-swap neighbourhood N_{S_1}, the number of neighbour solutions is exactly $k(n-k)$. The feasibility is automatically guaranteed, and the evaluation of the objective requires to compute the distance of two points from the current solution, that is $O(k)$ time. The resulting overall estimate is $O\left(t_{\max}(n-k)k^2\right)$.

C.2.2 Empirical evaluation

We can now evaluate the performance of the *steepest ascent* heuristic with the global-best strategy.

[1] This is just a detail, but it is probably better to add and remove d_{ij} rather than checking at every step whether $k = j$ or not.

Computational time analysis

From the detailed results, we can remark that the overall computational time (including both the initialisation constructive procedure and the following exchange procedure) ranges from fractions of a second to a couple of minutes. This is two orders of magnitude larger than the time required for the constructive heuristic alone (up to half a second), and therefore mostly depends on the exchange phase. It is comparable to the try-all heuristic only for the smaller instances, and one order of magnitude smaller for the larger ones (in fact, the try-all heuristic repeats the basic constructive heuristic n times, with n ranging from 100 to 1 000). The exchange heuristic is therefore less efficient than the constructive (and destructive) ones, but more efficient than the try-all heuristic.

Figure C.1 reports the semilogarithmic *scaling diagram* for the computational time of the steepest ascent heuristic on the whole benchmark. The diagram shows the expected polynomial increase with size. The $O\left(t_{\max} n k^2\right)$ theoretical estimate, with $k \propto n$, suggests an overall $O\left(t_{\max} n^3\right)$ complexity. The linear interpolation:

$$T_A = \beta n^{\alpha} \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

suggests that $\alpha \approx 4.3$ and $\beta \approx 5.75 \cdot 10^{-12}$. If we assume a cubic complexity for the neighbourhood exploration, this would imply that $t_{\max}$ increases slightly more than linearly with n. To test more precisely this conclusion we can compute and plot the ratio T/t_{max} of the total time T on the number of neighbourhood explorations t_{max} (see the yellow graph in Figure C.1) and make an interpolation on it. Since $\alpha \approx 2.7$ and $\beta \approx 3.6 \cdot 10^{-9}$, it seems that the cubic estimate for the time required to explore a single neighbourhood is excessive, and that $t_{\max}$ is more than linear in n, though not quadratic. One can also notice that the first diagram is much more irregular than the first one, meaning that $t_{\max}$ is not strictly dependent on n. Of course, we could also directly interpolate t_{max} as a function of size: a quick look at the detailed results for each fixed value of n suggests that t_{max} indeed strongly depends on k, increasing more than linearly: it becomes about 10 times larger as k goes from $0.1\,n$ to $0.4\,n$.

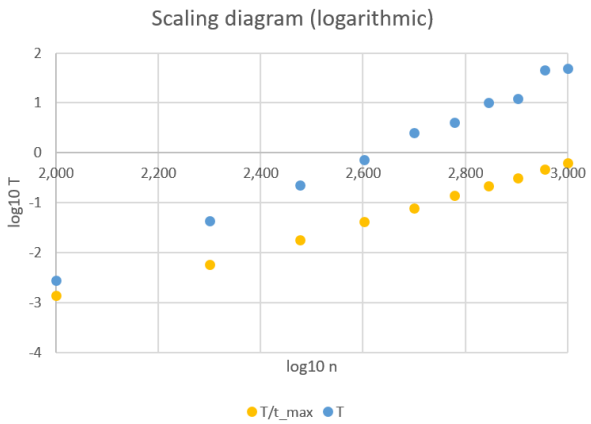


Figure C.1: Scaling diagram in logarithmic scales for the steepest ascent algorithm on the benchmark

Solution quality analysis

Figure C.2 reports the *SQD* diagram, compared with that of the initialisation farthest-point procedure. Of course, the former strictly dominates the latter, as it receives the solution in

input and proceeds by improving it with local search. What is interesting is the amount of the improvement, that is strong, but not huge: the average gap δ decreases from 1.52% to 0.97%. This suggests that the *MDP* has many local optima of various quality and with small basins of attraction. In fact, the try-all heuristic still performs better than the steepest ascent. Of course, as it takes much more time, we can't say that the latter is dominated. However, it is still impossible to dismiss the try-all heuristic as a viable approach. One can notice that the steepest ascent heuristic has a larger probability of finding very small gaps (below 0.3%), though not of finding the best known result. This suggests a region of stronger stability, possibly corresponding to the capacity of improving good initial results.

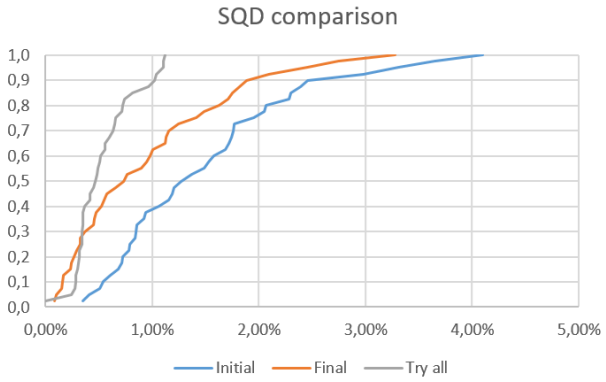


Figure C.2: Solution Quality Distribution diagram for the *steepest ascent* and the *farthest point* heuristics

The boxplots reported in Figure C.3 provide the same information.

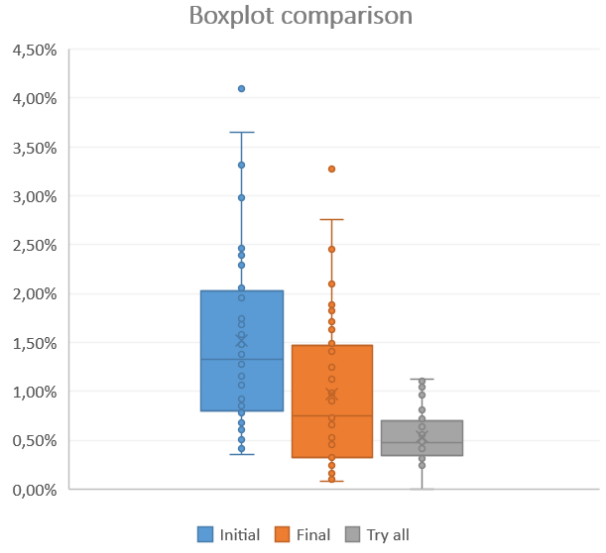


Figure C.3: Boxplots for the *steepest ascent* and the *farthest point* heuristics

We do not apply statistical tests to this comparison, because by definition the dominance of steepest ascent with respect to the farthest point heuristic is strict and the tests would not add anything to this fundamental information.

C.2.3 Constant-time neighbour evaluation

From theory we know that the variation of a quadratic objective function implied by a simple swap of elements can be estimated in constant time exploiting the formula

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = \sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

by saving in a suitable vector D_i the total distance of each point $i \in P$ from the current solution x, both for internal and external points. In fact, given this vector

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = D_j - D_i - d_{ij}$$

can be computed in two operations. Of course, whenever the current solution x changes into $x' = x \setminus \{i\} \cup \{j\}$, the vector D must be updated. This takes time $O(n)$ applying the following formula

$$D_k := D_k - d_{ik} + d_{jk} \qquad \text{for all } k \in P$$

Overall, this reduces the time to explore a single neighbourhood from $O\left((n-k)k^2\right)$ to $O\left((n-k)k\right)$, at the cost of adding an $O(n)$ term, that is dominated.

From the implementation point of view, we need to decide where to store vector D. The most natural approaches are either to keep it explicitly as a variable in procedure `steepest_ascent` or to "hide" it in solution x. In the former case, we will have to pass it as an argument to procedure `explore_neighbourhood`. In the latter case, we will have to update all the functions that manipulate objects of type `solution_t`. The choice mainly depends on whether we think that the vector will be used outside of the exchange heuristic or not. As we are going to use it also in the exchange metaheuristics, we will adopt this approach. Moreover, it make sense to go back to constructive heuristics and metaheuristics and evaluate whether the computational trick would provide an advantage also in those algorithms. Indeed, the $O\left(nk^2\right)$ complexity of the constructive heuristics depends on applying k times the basic constructive step, in which for all external points (hence, the $O(n)$ term) the distance from the current solution is estimated in time $O(k)$. The introduction of vector D would remove that term, and therefore the total time to $O(nk)$ at the cost of an additional $O(n)$ term, that is dominated. All the required work auxiliary work has been done in library `solution2`, that we will use in the following instead of `solution`. This will also require to modify accordingly the inclusion directive in file `local_search.h`.

Adding vector D to the `solution_t` data structure under the form of a dynamic integer vector (`int *`) requires to update the creation, destruction, copy and check procedures, as well as the manipulation procedures (`add_point`, `delete_point` and `swap_points`). After this update, it is possible to implement the evaluation of the exchange simply as follows.

```
delta  = px->D[get_index(p_out,pI)];
delta -= px->D[get_index(p_in,pI)];
delta -= pI->d[get_index(p_in,pI)][get_index(p_out,pI)];

return delta;
```

On the other hand, the `swap_points` procedure must also update the elements of vector D (in $O(n)$ time), but it can save $O(k)$ time for the update of the objective value.

Solving again the whole benchmark leads to exactly the same results as above, as expected. The computational time is however much smaller: in Figure C.4, the logarithmic scale clearly

shows a decrease in the slope, corresponding to a reduction of the exponent in the polynomial dependence of the computational time on the number of points[2]. Just to have an intuitive idea of the improvement, the computational times now range from fractions of a second to 1 second, instead of two minutes. This impressive result derives from having reduced the theoretical worst-case complexity from $O\left(t_{\max} nk^2\right)$ to $O\left(t_{\max} nk\right)$. The time is not strictly reduced by a factor of k (from 10 to 400 in the benchmark) because the contribution of secondary terms previously overwelmed by the evaluation of the value for the neighbour solutions now can becomes perceivable.

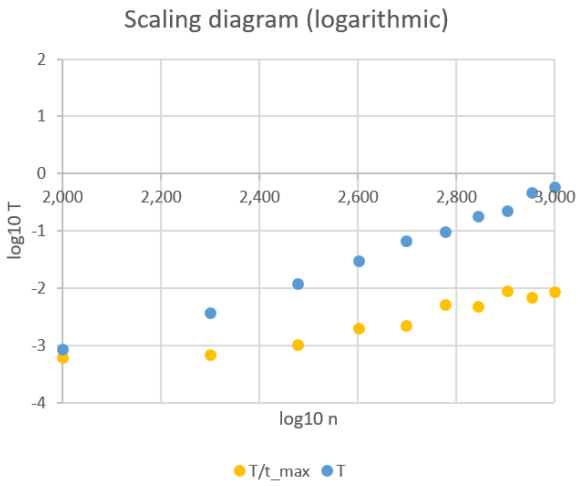


Figure C.4: Scaling diagram in logarithmic scales for the steepest ascent algorithm with the constant-time or the linear-time evaluation of neighbours

C.2.4 Comparison of initialisation procedures

We now investigate whether the initial solution $x^{(0)}$ exerts a long-term influence on the quality of the final solution returned by the exchange heuristic; in other words, whether good starting solutions tend to fall into the basin of attraction of good local optima. This would recommend the adoption of a good constructive procedure, provided that its computational time is not excessive. To investigate also this aspect, we consider a third variant in which the initial solution is obtained with the "try-all" heuristic[3].

Figure C.5 shows the three *SQD* diagrams for the initial solutions and for the final ones. The diagrams are in semilogarithmic scale because the gaps are widely distributed (part of the diagrams is missing because zero values cannot be represented on a logarithmic axis). This allows to appreciate the relative improvement obtained by the exchange procedure with respect to each different starting point This is particularly strong in the case of the random initialisation, whereas it becomes smaller for the constructive initialisations. In particular, it can be noticed

[2]Actually, α decreases from 4.3 to 2.9 for the overall time and from 2.7 to 1.3 for the time per iteration, which seems rather too much, but I have had no time to check the numbers and to investigate the reason of this behaviour.

[3]It is clear that this heuristic could be fully exploited by running an exchange procedure on each of the starting solutions it provides, but we are now focusing on the improvement power of a single run of an exchange heuristic. Applying the exchange heuristic to each solution generated by the "try-all" heuristic would be more on the line of a multi-start exchange metaheuristic. The same can be said about the classical *GRASP* mechanism that applied an exchange procedure to each solution generated by the semigreedy algorithm.

that the worse starting solutions are only slightly improved (in the case of the try-all heuristic, the second-worse solution is not improved, so that the diagram appears to reach the upper bound nearly for the same gap). It is however clear that better starting solutions tend to be associated with better final solutions: the exchange heuristic is not strong enough to overcome the initial advantage.

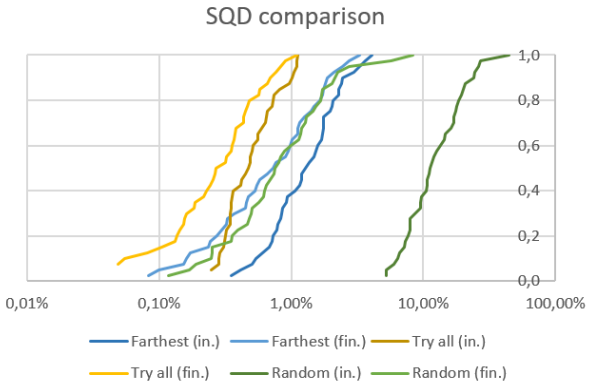


Figure C.5: Comparison of different initialisation procedures

Of course, the computational times are also relevant for the choice. Notice that we can apply the try-all heuristic only because the constant-time evaluation procedure can be extended to the constructive phase, so that this heuristic no longer takes several minutes to provide the starting solution for the larger instances. Indeed, its overall time requirement (constructive and improvement phase) ranges from fractions of a second to about 5 seconds, as opposed to about 1 second for the farthest point initialisation and 1.6 seconds for the random initialisation. The try-all initialisation is slower due to the more refined constructive phase; in fact, the number of neighbourhood explorations before reaching the local optimum is on average 15. The random initialisation has an extremely fast initialisation, but an average of 100 iterations. That implies the final longer time with respect to the farthest point heuristic, which only makes 25 iterations before hitting the local optimum.

Figure C.6 provides the *RTD* diagrams on the benchmark. We remind that such a diagram makes little sense for benchmarks collecting instances of different size, but it allows meaningful comparisons between different algorithms.

This suggests that a random initialisation is not a good idea (at least for the steepest ascent heuristic), whereas it is still open whether the farthest point or the try-all initialisation (or other non fully random ones) could be more effective.

C.2.5 Neighbourhood tuning: *global-best* versus *first-best*

Finally, we experiment with the idea of tuning the exploration of the neighbourhood, that is terminating it as soon as an improving solution is found with respect to the current one. This is known as *first-best* strategy, as opposed to the classical *global-best* strategy that visits the whole neighbourhood and returns the overall best solution it contains. The rationale is to accept a smaller improvement in each step of the search, in exchange for a much smaller computational time, that allows to perform many more steps, possibly getting earlier to the same local optimum. It must be noticed that, changing the rule that determines the following visited solution in general also changes the basins of attraction, and therefore the final local optimum reached. This could be worse or better than in the global-best case. To make things even more complex,

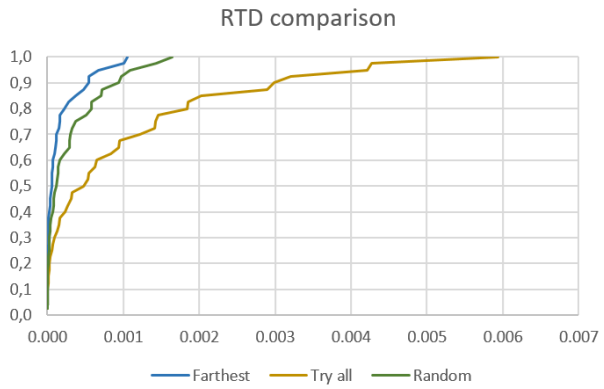


Figure C.6: Comparison of the overall running time of the exchange heuristic for different initialisation procedures

the local optimum returned also depends on the order in which each neighbourhood is visited, a fact that could be perhaps exploited somehow (possibly, starting with the most promising elements based on the distance D from x), with a possible increase of the computational cost. We are not going to explore this line of research: the neighbourhood will be explored scanning x and $P \setminus x$ with the corresponding lists, exactly as in the global-best strategy.

In order to impose the first-best strategy, an extremely simple modification must be made to the neighbourhood exploration procedure: as soon as an exchange with a positive effect is found, the procedure terminates returning that exchange.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px); p_in = next_point(p_in,px))
  for (p_out = first_point_out(px); !end_point_list(p_out,px); p_out = next_point(p_out,
  {
    delta_f = evaluate_exchange(p_in,p_out,px,pI);
    if (delta_f > *pdelta_f)
    {
      *pdelta_f = delta_f;
      *pp_in = p_in;
      *pp_out = p_out;
      if ( (delta_f > 0) && (strcmp(visit_strategy,"-fb") == 0) ) break;
    }
  }
```

For the sake of simplicity, we apply the farthest-point initialisation heuristic. The detailed results show that the number of iterations tends to be larger with respect to the global-best strategy, as expected, but the difference is not strong (30 versus 25 on average). In fact, the computational time is shorter, and the results comparable. Figure C.7 shows that the two algorithms have rather similar performance with respect to the solution quality (perhaps, the first-best strategy is slightly better).

Figure C.8, however, shows that the first-best strategy is clearly faster. This could be enough to suggest to adopt it instead of the classical global-best strategy.

In order to further support this choice, we can compare the two sets of results with Wilcoxon's test. The results concerning the solution quality are:

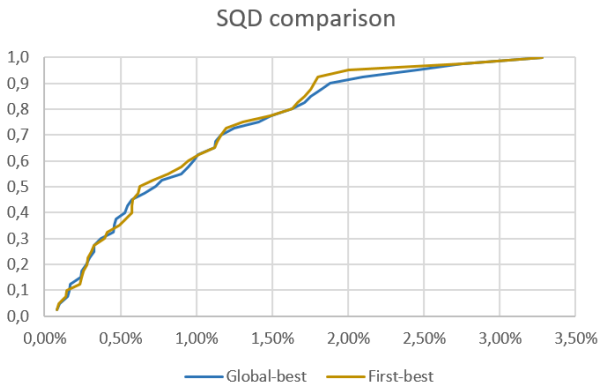


Figure C.7: Comparison of the solution quality of the exchange heuristic with the global-best and first-best neighbourhood exploration strategies

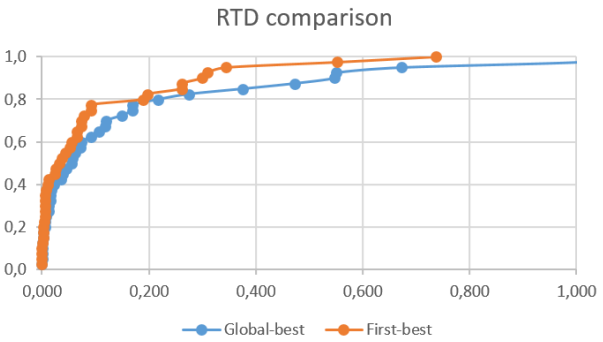


Figure C.8: Comparison of the computational time of the exchange heuristic with the global-best and first-best neighbourhood exploration strategies

```
W+ = 168, W- = 297, N = 30, p <= 0.188
```

and suggest that, while there is a slight predominance of the first-best strategy with respect to the global-best, it would not be unlikely that such a predominance be due only to a random extraction (the p-value is 18.8%, that is quite large).

On the other hand, the results for the computational time are:

```
W+ = 677.50, W- = 25.50, N = 37, p <= 9.094e-007
```

and indeed suggest that the global-best strategy takes a longer time, even if the first-best strategy requires more neighbourhood explorations (the p-value is approximately 10^{-6}).

APPENDIX D

Laboratory on Exchange Metaheuristics

D.1 Introduction

This chapter discusses the application of exchange metaheuristics to the *Maximum Diversity Problem* (*MDP*). Exchange metaheuristic extend the basic scheme of exchange heuristics modifying its elements (the starting solution, the neighbourhood, the objective function or the selection rule) in order to proceed with the search after reaching a locally optimal solution. In the following, we will consider two such extensions, both based on the basic *steepest ascent* heuristic described in the previous chapter. The former is a *Variable Neighbourhood Search* (*VNS*) heuristic, that restarts the search from a new solution generated with a *shaking* procedure on the best known solution. The latter is a *Tabu Search* (*TS*) heuristic, that prolongs the search beyond local optima by looking for the minimum cost neighbour solution that respects suitable *tabu* conditions, designed to avoid a cyclic behaviour.

For the sake of simplicity, we will build the starting solution with the farthest-point constructive heuristic introduced in Chapter 3 and apply the steepest ascent heuristic with the *first-best* strategy that proved to be equally effective and more efficient in Chapter 4. The command line arguments will allow the user to choose which of the two metaheuristics to apply and the associated parameters:

- for the *VNS* metaheuristic, option `-vns`, followed by five parameters: the total number of neighbourhood explorations $t_{\max}$, the minimum neighbourhood index $s_{\min}$, the index variation δs, the maximum neighbourhood index $s_{\max}$ and the seed of the pseudorandom number generator;
- for the *TS* metaheuristic, option `-ts`, followed by three parameters: the total number of neighbourhood explorations $t_{\max}$, the tabu tenure for the reinsertion of deleted elements L^{in}, and the tabu tenure for the removal of added elements L^{out}.

```
parse_command_line(argc,argv,data_file,exchange_algo,&niter,&seed,
                   &s_min,&delta_s,&s_max,&l_in,&l_out);

load_data(data_file,&I);

create_solution(I.n,&x);
greedy_farthest(&I,&x);

start = clock();
if (strcmp(exchange_algo,"-vns") == 0)
```

```
    variable_neighborhood_search(&I,&x,"-fb",niter,s_min,delta_s,s_max,&seed);
  else if (strcmp(exchange_algo,"-ts") == 0)
    tabu_search(&I,&x,"-fb",niter,l_in,l_out);
  end = clock();
  tempo = (double) (end - start) / CLOCKS_PER_SEC;

  printf("%s ",data_file);
  printf("%10.6f ",tempo);
  printf("%8d ",niter);
  print_sorted_solution(&x,I.n);
  printf("\n");

  destroy_solution(&x);
  destroy_data(&I);
```

D.2 Variable Neighbourhood Search

The *Variable Neighbourhood Search* metaheuristic applies a basic steepest ascent heuristic, and restarts it every time this terminates in a locally optimal solution. The restart is performed with a *shaking* procedure that modifies the current best known solution generating it at random in a suitable neighbourhood, whose size is the main parameter of the method[1]. The scheme is the following:

Algorithm VariableNeighbourhoodSearch$(I, x^{(0)}, \ell, s_{\min}, \delta s, s_{\max})$
$x :=$ SteepestAscent$(x^{(0)})$; $x^* := x$;
$s := s_{\min}$;
For $l := 1$ *to* ℓ *do*
  $x' :=$ Shaking(x^*, s);
  $x' :=$ SteepestAscent(x');
  If $f(x') > f(x^*)$
    then $x^* := x'$; $s := s_{\min}$;
    else $s := s + \delta s$;
  If $s > s_{\max}$ *then* $s := s_{\min}$;
EndWhile;
Return $(x^*, f(x^*))$;

This scheme is general enough to not require any specific adaptation for the *MDP*, except for the replacement of steepest descent with steepest ascent, as usual because it is a maximisation problem. However, we will also modify the termination condition, replacing the number of restarts ℓ with the total number of neighbourhood explorations $t_{\max}$ in order to get a better control of the computational time (of course, we could directly fix the total computational time, and that would be even more precise). To this purpose, we exploit the information on the number of neighbourhood explorations performed that is already provided by the steepest ascent procedure, but we also need to terminate if before reaching a locally optimal solution whenever

[1]This parameter is generally denoted as k (as in the slides of the theoretical lessons), but here we will denote it as s to distinguish it from the required number of points in the feasible solutions of the *MDP*.

the total number of available explorations has been consumed. For the sake of simplicity, we will adopt the steepest ascent heuristic discussed in the previous chapter, based on the single-swap neighbourhood N_{S_1} with the first-best exploration strategy. It is therefore straightforward to implement the `VNS` procedure as follows.

```
  truncated_steepest_ascent(pI,px,visit_strategy,niter,&iter);
  tot_iter = iter;

  create_solution(pI->n,&x_star);
  copy_solution (px,&x_star);

  s = s_min;
  while (tot_iter < niter)
  {
    shaking(pI,px,s,pseed);
    truncated_steepest_ascent(pI,px,visit_strategy,niter-tot_iter,&iter);
    tot_iter += iter;

    if (px->f > x_star.f)
    {
      copy_solution(px,&x_star);
      s = s_min;
    }
    else
    {
      s += delta_s;
      if (s > s_max) s = s_min;
    }
```

The variable `tot_iter` saves the cumulative number of neighbourhood explorations performed, in order to compare it with the maximum imposed value `niter`. Moreover, the `truncated_steepes` procedure coincides with the already implemented `steepest_ascent` procedure, with the additional termination condition of stopping as soon as the remaining `niter-tot_iter` explorations have been performed. The modified procedure is already provided in a modified `localsearch.c` library.

```
*pniter = 0;
if (max_iter <= 0) return;
do
{
  explore_neighbourhood(px,pI,visit_strategy,&p_in,&p_out,&delta_f);
  if (delta_f > 0)
  {
    swap_points(p_in,p_out,px,pI);
    (*pniter)++;
  }

} while ( (delta_f > 0) && (*pniter < max_iter) );
```

The only part of the `variable_neighborhood_search` procedure that remains unimplemented is the `shaking` procedure, that receives the reference solution (that is the best known one, as we are applying the basic version of the *VNS*) the current value of parameter s and the seed of the pseudorandom number generator, and returns the perturbed solution that will be used as a starting point for the following application of the steepest ascent procedure. In order to implement it, we need to define a hierarchy of neighbourhoods from which to extract a random solution. In general, these neighbourhoods should be progressively increasing, in order to allow a controllable amount of intensification (using the first neighbourhoods) or diversification (using the last ones). In the specific case of the *MDP*, swaps are the most natural operation to generate neighbourhoods, due to the cardinality constraint, that guarantees the feasibility of every solution they generate, while proving the unfeasibility of the subsets generated by any other kind of operation. Therefore, we will adopt the hierarchy formed by the swap neighbourhoods N_{S_s}. Every solution in N_{S_s} is obtained performing s single swaps of a point in the current solution with a point out of it. A technical detail to be defined is whether we consider:

- s possibly overlapping swaps, in which a point deleted by one of the swaps can be added again by a following swap;
- s disjoint swaps, that is we swap s points from the starting solution with s points out of the starting solution.

The former definition allows to obtain also solutions that could be obtained with any number $s' \leq s$ of swaps (because swapping points i and j, followed by j and k is equivalent to swapping directly i and k). Therefore, it is a more general definition and it guarantees that each neighbourhood in the hierarchy include the previous ones. The latter definition restricts the neighbourhood only to disjoint swaps and yields disjoint neighbourhoods. However, it has the advantage that all neighbour solutions have a Hamming distance exactly equal to $2s$ from the reference solution x^*. This property seems particularly desirable from the point of view of controlling the size of the perturbation introduced (in the former case, a perturbation with a very large s might generate a solution very close to x^*, possibly even coincident with it). We therefore adopt the second definition. Generating a random s-swap is then straightforward: it requires to save in a vector the elements of the solution and extract s uniformly without repetitions. The same must be done for the elements out of the solution. Then, we can match the two sets of points in s pairs and swap the points to obtain the perturbed solution. There are several ways to do that, whose time and space complexity can be discussed in detail. Choosing one arbitrarily, we will write in a single vector of n elements the indices of all points: those in the solution in its first positions and the other ones in the last positions (using the incidence vector to distinguish them). Then, we will randomly extract s indices from the first and s from the second subvector, moving them to the ends of the vector to avoid generating duplicate terms and finally swap the corresponding points with the `swap_points` procedure. It can be argued that an *ad hoc* procedure could be more efficient (in particular for large values of s), but the shaking procedure is applied rarely enough to assume that it will take a negligible part of the overall computational time of the algorithm[2]

[2]This assumption should be verified, of course.

```
/* Build a vector with the indices of the internal points in the first k positions
   and the indices of the external point in the last n-k positions */
Indices = int_alloc(pI->n+1);
i_in = 1; i_out = pI->n;
for (i = 1; i <= pI->n; i++)
  if (px->in_x[i] == true)
    Indices[i_in++] = i;
  else
    Indices[i_out--] = i;

/* Select s internal points and move their indices to the first s positions of Indi
for (i_in = 1; i_in <= s; i_in++)
{
  i = rand_int(i_in,pI->k,pseed);
  temp = Indices[i_in];
  Indices[i_in] = Indices[i];
  Indices[i] = temp;
}

/* Select s external points and move their indices to the last s positions of Indic
for (i_out = pI->n; i_out >= pI->n-s+1; i_out--)
{
  i = rand_int(pI->k+1,i_out,pseed);
  temp = Indices[i_out];
  Indices[i_out] = Indices[i];
  Indices[i] = temp;
}

/* Perform s exchanges between the first and the last points of Indices */
for (i_in = 1, i_out = pI->n; i_in <= s; i_in++, i_out--)
    swap_points(get_point(Indices[i_in],pI),get_point(Indices[i_out],pI),px,pI);

free(Indices);
```

D.2.1 Time complexity estimation

The computational complexity of the *VNS* metaheuristic combines that of the steepest ascent heuristic with that of the shaking procedure. The former has already been estimated as $O(t_{\max}(n-k)k)$, with the modification that $t_{\max}$ is now the total number of neighbourhood explorations, fixed by the user. Applying the first-best exploration strategy means that the number of explored solutions should be $\leq (n-k)k$, though usually smaller (probably much smaller in the first iterations, when it is easier to improve the current solution, and nearly equal later when approaching the local optimum). The latter term is given by the initialisation in $O(n)$ time of the index vector, followed by the generation of s pseudorandom number (constant time, though probably not a very small constant) and s swaps, that take (as discussed in the previous chapter) $O(n)$ time each. Overall, this is $O(\ell ns)$ time, where $s \leq \min(k, n-k)$ (as discussed in the following) and ℓ is the number of restarts (unknown *a priori*). However, s and ℓ tend to be inversely correlated, since smaller perturbations probably imply shorter paths to the local optimum and

more restarts, whereas larger perturbations imply longer paths to the local optimum and less restarts. A rough guess is that each shaking application should take a time comparable to a neighbourhood exploration, so that overall its contribution should be negligible with respect to the steepest ascent heuristic, but this needs to be verified experimentally.

D.2.2 Empirical evaluation

We can now evaluate the performance of the *VNS* metaheuristic. We have decided to use the total number of neighbourhood explorations as a termination condition. This value should be set so as to allow the heuristic to experiment with every possible working condition. However, we also want to get results in a reasonably short time. In the previous chapter, we have estimated that the average number of iterations performed from a starting solution to the corresponding local optimum ranges between 15 and 30 in our benchmark instances. It will probably be smaller for starting solutions generated with small perturations from the best known solution. Since $s \leq k$, it should take at most $30 \cdot 400$ neighbourhood explorations to reach the strongest perturbations on the largest instances. Therefore, $t_{\max} = 10\,000$ could be a good choice, but we will adopt $t_{\max} = 1\,000$ to fit the experiments in the space of a lesson.

In the first experiments, we will let parameter s autotune, by fixing $s_{\min}$, δs and $s_{\max}$ to their simplest values: $s_{\min} = 1$, $\delta s = 1$ and $s_{\max} = \min(k, n-k)$.

Computational time analysis

The detailed results show that the overall computational time (including the initialisation constructive procedure, the shaking procedure and the steepest ascent procedure) ranges from fractions of a second to less than four seconds. This is even shorter than the exchange heuristic initialised by the try-all heuristic, which is a quite promising fact, should the results prove at least as good. Let us remind that the steepest ascent procedure applied to the farthest-point procedure could not enhance it enough to overcome the advantage provided by the better try-all initialisation procedure, even if its computational time was smaller. We can investigate whether the use of *VNS* allows exploit the shorter computational time to get better results.

Figure D.1 reports the logarithmic *scaling diagram* for the computational time of the *VNS* heuristic on the whole benchmark. The diagram shows a very regular polynomial increase with size, also thanks to the fixed number of iterations $t_{\max}$. The $O((n-k)k)$ theoretical estimate, with $k \propto n$, suggests a quadratic complexity, which is in extremely good accordance with the linear interpolation:

$$T_A = \beta n^{\alpha} \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

since $\alpha \approx 2.008$ and $\beta \approx 2.6 \cdot 10^{-6}$.

Solution quality analysis

Figure D.2 reports the *SQD* diagram of the *VNS* metaheuristic, compared with that of the steepest ascent heuristic applied to the result of the farthest-point and of the try-all heuristics: the former allows the comparison with the result obtained stopping at the first local optimum; the latter allows a comparison with a heuristic taking a similar computational time. The improvement with respect to the former is clear (the average gap decreases from 0.93% to 0.33%): it is a strict dominance, given that they visit the same solutions, and the *VNS* proceeds when the competitor just terminates. Also with respect to the try-all initialisation the average gap is better (0.33% versus 0.35%) and the *SQD* diagram shows a much larger fraction of very good results,

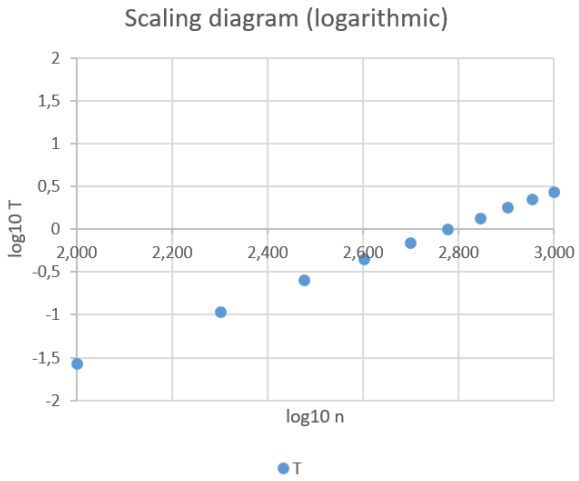


Figure D.1: Scaling diagram in logarithmic scales for the *VNS* algorithm on the benchmark

but also a larger fraction of bad results. Notice that the computational time is not the same: a true comparison should be made in perfectly equal conditions.

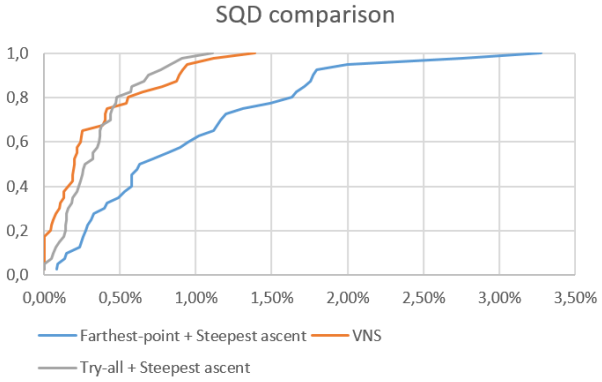


Figure D.2: Solution Quality Distribution diagram for the *VNS* metaheuristic compared with the *steepest ascent* heuristic initialised with the *farthest point* and the *try-all* procedures

The boxplots reported in Figure D.3 provide the same information: *VNS* provides both better and worse results with respect to the *steepest ascent* initialised by the try-all heuristic.

In order to have a rough idea of the corresponding computational times, Figure D.4 provides the *RTD* diagram, from which it is apparent that *VNS* has some margin of further improvement for the slower runs, that correspond to the largest instances. On the contrary, it is already slower for the faster runs, that is the smallest instances. Therefore an equal-time comparison is clearly necessary to be fair. Performing one, however, is complicated when some of the competing algorithms have an intrinsic termination condition: we should build a time-limited version of the *VNS* heuristic, run the steepest ascent heuristic saving its computational time, and feed it to the *VNS* heuristic. This is beyond the scope of the course.

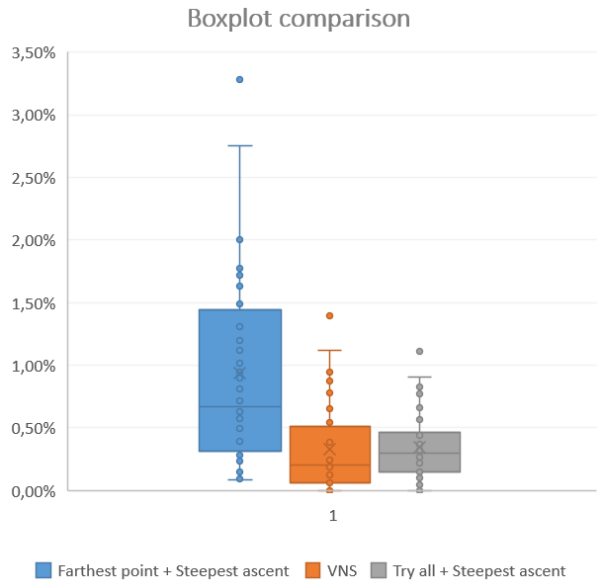


Figure D.3: Boxplots for the *VNS* metaheuristic compared with the *steepest ascent* heuristic initialised with the *farthest point* and the *try-all* procedures

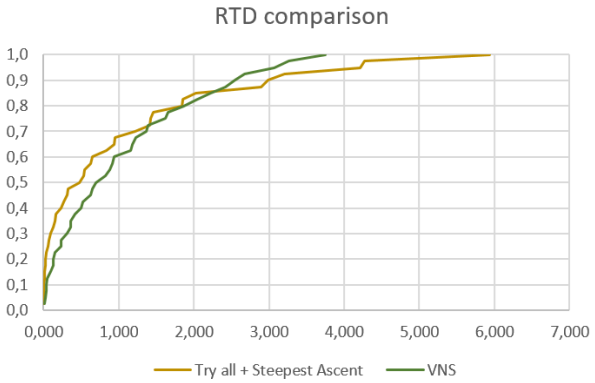


Figure D.4: Solution Quality Distribution diagrams for the *VNS* metaheuristic and the *steepest ascent* heuristic initialised with the *try-all* procedure

D.2.3 Parameter tuning

It is often possible to improve the performance of an algorithm by tuning the values of its parameters. In the case of the *VNS*, this corresponds to increasing the smallest neighbourhood used to restart the search, or decreasing the largest one, or skipping some intermediate neighbourhoods. The last possibility is useful when the number of neighbourhoods is huge. In this case, s ranges from 1 to 400, that is quite large. As we have fixed a rather small total number of neighbourhood explorations $t_{\max}$, it is possible (and it can be easily verified printing the value of s at each shaking operation) that on the larger benchmark instances only small shakings are performed. In order to test this aspect, we compare some alternative parameter configurations. The test is very limited, considering the following configurations, chosen so as to cover rather extreme cases:

- $s_{\min} = 1$ and $s_{\max} = k$: the trivial configuration (probably not fully exploited, for the insufficient number of iterations);
- $s_{\min} = 1$ and $s_{\max} = 10$: an intensifying configuration, imposing rather small perturbations;
- $s_{\min} = 1$ and $s_{\max} = k/2$: a configuration avoiding only very large perturbations (this could coincide with the trivial fone if the insufficient number of neighbourhood explorations forbids to increase s beyond the upper bound);
- $s_{\min} = k/2$ and $s_{\max} = k$: a configuration producing strong perturbations since the beginning to diversify the search and try to avoid falling back in the reference solution;
- $s_{\min} = k$ and $s_{\max} = k$: a degenerate configuration in which the maximum possible perturbation is applied (all points must in the best known result are replaced by new random points);

The number of configurations and the number of iterations for each configuration are too limited to provide really significant results, but they are a good occasion for some intuitions and further discussion. A good way to formulate reasonable configurations is to print the values of s that improve the best known result. Increasing $s_{\min}$ and decreasing $s_{\max}$ to approach such values will modify the overall behaviour of the algorithm, because the same sequence of pseudorandom numbers will imply completely different choices, but it is probably a good idea: if no improvement is found out of a certain range of values of s, this probably means that the radius of the basins of attraction falls in a similar range, and therefore a good perturbation should be in the same range.

A quick glance at the computational times suggests that they are nearly independent from the parameters: the variations are usually below 10%, that in a period of few seconds is probably just due to random fluctuations. The quality of the results is roughly indicated by Table D.1 which reports the average gaps with respect to the best known result. The diversifying configuration ($[k/2,k]$) is the best one, followed by the relaxed configuration ($[1,k]$) and the slightly intensifying one ($[1,k/2]$), that are very similar, as expected. The strongly intensifying and the strongly diversifying configuration appear to be the worst. It can be interesting to notice that the strongly diversifying configuration ($[k,k]$) actually proved the best in one of my personal contributions to the literature, which combined rather short runs of a *TS* metaheuristic with a *VNS* restart mechanism. This is not unreasonable, given that the *TS* procedure probably guarantees a good exploration of the region surrounding the current best known result (better than

$s_{\min}$	$s_{\max}$	Average gap
1	k	0.33%
1	10	0.45%
1	$k/2$	0.34%
$k/2$	k	0.22%
k	k	0.44%

Table D.1: Average gaps with respect to the best known result of the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

the steepest ascent procedure adopted here), so that stronger perturbations in the restart make sense.

Figures D.5 and D.6 show the *SQD* and boxplot diagrams of the five configurations, that clearly confirm the dominances suggested by the average gaps. Indeed, checking the results of the steepest ascent heuristic initialised with the try-all procedure, the diversifying configuration seems to be comparable even in the upper part of the diagram.

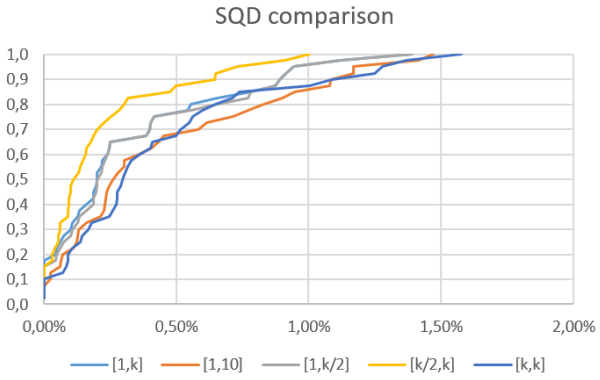


Figure D.5: Solution Quality Distribution diagrams for the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

Statistical tests

We can also compare the performance of the five configurations with statistical tests. Using Wilcoxon's test poses a methodological problem: the test is designed to compare two empirical populations. It is possible to use it on more than two algorithms, applying it to all pairs or choosing a reference heuristic and comparing the other ones to it. This, however, requires to handle with care the interpretation of the results. The p-value obtained, in fact, estimates the probability of observing the empirical results under the assumption that the two samples compared derive from the same population, that is, that the two configurations compared are equally effective. When p is small, this interpretation of the results can be rather safely rejected, but it must not be considered as straightforwardly false. If the test is applied n_t times, the probability that at least one of the interpretations drawn from the test is false becomes larger and larger. This means that our conclusions should be based on stricter requirements. The literature offer several methods to correct the estimates provided by a pairwise test. The simplest one is the *Bonferroni correction*, that is based on Boole's inequality for the *familywise error rate*

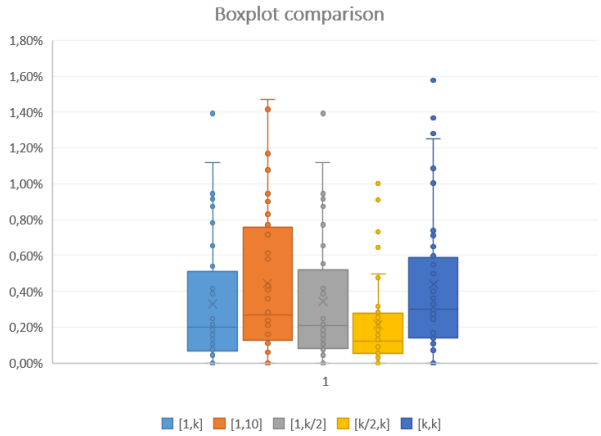


Figure D.6: Boxplot diagrams for the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

(*FWER*):

$$FWER = P\left[\bigcup_{i=1}^{n_t}(p_i \leq \alpha)\right] \leq \sum_{i=1}^{n_t} P\left[(p_i \leq \alpha)\right]$$

In other words, the sum of the p-values obtained gives an overestimate of the p-value for the overall observation. A simple way to impose it every given threshold on the significancy of the results (for example, the classical 5%) consists in dividing such a threshold by the number n_t of the tests.

For example, let us check the hypothesis that the diversifying configuration is better than the other four. This corresponds to the four following different pairwise tests:

1. $[k/2,k]$ versus $[1,k]$

```
W+ = 470, W- = 91, N = 33, p <= 0.0007329
```

2. $[k/2,k]$ versus $[1,10]$

```
W+ = 630, W- = 73, N = 37, p <= 2.743e-005
```

3. $[k/2,k]$ versus $[1,k/2]$

```
W+ = 499, W- = 96, N = 34, p <= 0.0005896
```

4. $[k/2,k]$ versus $[k,k]$

```
W+ = 612, W- = 18, N = 35, p <= 1.197e-006
```

The hypothesis is consistent with the observations even after applying Bonferroni's correction, since the sum of all p-values is 0.2% and each one is smaller than $5\%/n_t = 1.25\%$ (even a threshold much tighter than 5% would be respected).

Of course, other systems of assumptions, such as a full ordering among the configurations, could be checked, but they would probably be less interesting: we are mainly interested in finding the best performing configuration on the given benchmark.

D.3 Tabu search

The *Tabu Search* metaheuristic applies a basic steepest ascent heuristic, but modifies it introducing a limitation of the neighbourhood that forbids to accept as incumbent a solution already visited, but also (in the standard attribute-based version) a solution similar to the ones already visited. In order to adapt the *TS* metaheuristic to the *MDP*, we will initialise the search with the farthest-point heuristic and adopt the basic single-swap neighbourhood N_{S_1} as done for the *VNS*. This should make the comparison between the two approaches fairer, allowing to understand whether the problem is better attacked by restarting or prolonging the search after hitting a locally optimal solution. The general *TS* scheme can be easily adapted to the *MDP* with the usual replacements due to the maximising nature of the problem.

Algorithm TabuSearch$\left(I, x^{(0)}, t_{\max}, L\right)$
$x := x^{(0)}$; $x^* := x^{(0)}$;
$\bar{A} := \emptyset$;
For $t := 1$ *to* $t_{\max}$ *do*
  $x' := \emptyset$;
  For each $y \in N(x)$ *do*
    If $f(y) > f(x')$ *then*
      If Tabu$\left(y, \bar{A}\right) =$ *false or* $f(y) > f(x^*)$ *then* $x' := y$;
    EndIf
  EndFor
  $x := x'$;
  $\bar{A} :=$ Update$\left(\bar{A}, x', L\right)$;
  If $f(x') > f(x^*)$ *then* $x^* := x'$;
EndFor
Return $(x^*, f(x^*))$;

First, we must decide whether to apply the basic version of *TS* or the most common attribute-based one. In the *MDP* literature, the best performing algorithm proposed so far to solve the problem actually uses the basic version. However, this algorithm introduces several refinements, and we are interested in the *MDP* mainly as an example for the application of *TS* to general Combinatorial Optimization problems. We will therefore implement an attribute-based *TS* metaheuristic. It is rather natural to define a pair of complementary attribute sets, that are the presence and the absence of given points in the current solution. Setting $A = x$ and $A' = P \setminus x$ means that every time a point is deleted from the solution, it becomes tabu for a given number L^{in} of neighbourhood explorations to add it back; conversely, every time a point is added to the solution, it becomes tabu for a given number L^{out} of neighbourhood explorations to delete it.

Procedure *tabu_search* implements both tabu lists on a single integer vector T, each of whose components reports the last iteration at which the corresponding point has changed its status: if i currently belongs to x, T[i] is the iteration at which i has entered the solution; if i currently does not belong to x, T[i] is the iteration at which i has gone out of the solution. At the beginning, T[i] is set to -MAX_INT for all points, so that the check on the tabu status of any point states that the point is not tabu, and therefore free for exchanges. Procedure explore_neighbourhood_with_tabu differs from the explore_neighbourhood procedure implemented in the previous chapter for the steepest ascent heuristic in that it takes into account also the tabu status (in fact, it requires the current iteration index iter, vector T, the tabu tenures Lin and Lout, and the value of the best known solution w_star.f, to apply the aspiration criterium). The update of the tabu list simply amounts to saving the current iteration index in the two positions of vector T associated to the points swapped. We should also take into account the possibility that all moves in the neighbourhood are tabu. In that case, the neighbourhood exploration procedure should return the solution with the oldest tabu status. The current implementation simply returns no point and performs no move, remaining idle until some tabu expires. This is an inefficient implementation, that shall be corrected in future versions of the algorithm.

```
create_solution(pI->n,&x_star);
copy_solution(px,&x_star);

T = int_alloc(pI->n+1);
for (i = 1; i <= pI->n; i++)
  T[i] = INT_MIN;

for (iter = 1; iter <= max_iter; iter++)
{
  explore_neighbourhood_with_tabu(px,pI,visit_strategy,
                                  iter,T,l_in,l_out,x_star.f,
                                  &p_in,&p_out,&delta_f);
  if (p_in != NO_POINT)
  {
    swap_points(p_in,p_out,px,pI);
    T[get_index(p_in,pI)] = T[get_index(p_out,pI)] = iter;
    if (px->f > x_star.f) copy_solution(px,&x_star);
  }
}

free(T);

copy_solution(&x_star,px);
```

The exploration of the neighbourhood is performed exactly as in the steepest ascent heuristic, with the addition of a further check. If the currently explored swap is tabu (and this is checked by function is_tabu, then the move is performed only if it improves upon the best known one, that is if the improvement δf applied to the current solution value yields a value strictly better than the best known one (*aspiration criterium*). In this special case, in fact, the new solution is only apparently violating a tabu, and is on the contrary providing a precious

overall improvement to the search process. We still apply the first-best exploration strategy, returning the first improving solution in the neighbourhood, but only if it is nontabu, or it satisfies the aspiration criterium.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px); p_in = next_point(p_in,px))
  for (p_out = first_point_out(px); !end_point_list(p_out,px); p_out = next_point(p_out,
  {
    delta_f = evaluate_exchange(p_in,p_out,px,pI);
    if ( (delta_f > *pdelta_f) &&
         (!is_tabu(p_in,p_out,pI,iter,T,l_in,l_out,px->f+delta_f,fstar)) )
    {
      *pdelta_f = delta_f;
      *pp_in = p_in;
      *pp_out = p_out;
      if ( (delta_f > 0) && (strcmp(visit_strategy,"-fb") == 0) ) break;
    }
  }
```

Finally, the check of the tabu status simply consists in determining whether the current iteration index has reached or not the value at which the tabu expires. This must be checked both for the point i that is leaving the solution (and in that case the tabu tenure is `l_out`) and for the point j that is entering the solution (in that case the tabu tenure is `l_in`) .

```
if (f > f_star) /* aspiration criterium */
  return false;
else
  return ((iter <= T[get_index(p_in,pI)] + l_out) ||
          (iter <= T[get_index(p_out,pI)] + l_in));
```

D.3.1 Time complexity estimation

It is rather obvious that the computational complexity of the *TS* metaheuristic coincides with that of the basic steepest ascent heuristic, as all additional operations require constant time in their respective locations:

- the evaluation of the tabu status adds a constant number of operations to the evaluation of the cost of each explored solution;
- the update of the tabu list adds a constant number of operations to the execution of the move, that is the exploration of a neighbourhood;

The allocation, initialization and deallocation of vector `T` add $O(n)$ time to the overall algorithm. Therefore, the overall complexity remains $O(t_{\max}(n-k)k)$, where the number of neighbourhood explorations $t_{\max}$ is a parameter provided by the user as the termination condition.

D.3.2 Empirical evaluation

We can now evaluate the performance of the *TS* metaheuristic. We set the total number of neighbourhood explorations to $t_{\max} = 1\,000$, as for the *VNS* metaheuristic, in order to allow a meaningful comparison between them, even if a completely fair comparison would require to give them the same computational time.

Cyclic or erratic behaviours

Contrary to the *VNS*, where it is usually very easy to determine a default configuration for the parameters $s_{\min}$, δs and $s_{\max}$, the most influential parameters of the *TS*, that is the *tabu tenures* need to be tuned with a certain care since the beginning. Two basic complementary risks must be avoided:

- *cyclic behaviours*: if the tabu tenure is too short, the search can get stuck in a cyclic sequence of solutions, because the search is attracted by locally optimal solutions that have already been visited and the tabu expires before the cycle starts again;

- *erratic behaviours* or empty neighbourhoods: if the tabu tenure is too long, the search can wander in the solution space avoiding the more promising regions because these are too close to solutions that have already been visited; the neighbourhood can even become fully tabu.

Since the number of points out of the solution varies from $n-k = 60$ to $n-k = 900$, and the number of points in the solution varies from $k = 10$ to $k = 400$, the tenures should keep below these values, but above a few units, which is still a rather large range. The tenures could also possibly depend on the size of the instance. It is also likely that the tenure for reinsertion L^{in} should be larger than the tenure for redeletions L^{out}, because the candidate elements for insertion (the $n-k$ external ones) are more numerous than the candidate elements for deletion (the k internal ones). A simple way to verify the occurrence of the cyclic behaviours and (less evidently) of erratic ones, is to plot the profile of the objective function along the search.

In order to give a quick idea of how reasonable values are introduced, we will fix focus on the smallest instance (`n0100k010.txt`), and investigate different possible values for L^{in} while trivially setting $L^{\text{out}} = 1$, so that the tabu mechanism is mainly based on the reinsertion of deleted elements. Figure D.7 reports the profile of the objective function for the first 100 neighbourhood explorations. The configuration with $L^{\text{in}} = 5$ exhibits a clearly periodic profile, suggesting that the tenure is too short. The one with $L^{\text{in}} = 6$ appears much better, but long cycles (with a period of 106 iterations actually arise after a while). The configuration with $L^{\text{in}} = 8$ hits the best known result ($f^* = 3\,561$) several times during the whole run. The configuration with $L^{\text{in}} = 30$ starts moving erratically in regions of worse quality. Finally, the configuration with $L^{\text{in}} = 100$ seems to avoid good solutions and also shows sequences of iterations in which the value of the objective does not change because the whole neighbourhood is tabu ($L^{\text{in}} = 100 > n-k = 90$) and our simple implementation trivially waits for the tabu to expire.

Figure D.8 reports the profile of the objective function for the whole run on the largest instance (`n1000k400.txt`). The configurations with the smallest values of L^{in} actually prove better, without exhibiting cyclic behaviours. This (absolutely nonobvious) behaviour is possibly due to the much larger size of the neighbourhood, that allows the algorithm to choose the incumbent in a larger set, and thus reduces the risk of repeating the same sequence of moves.

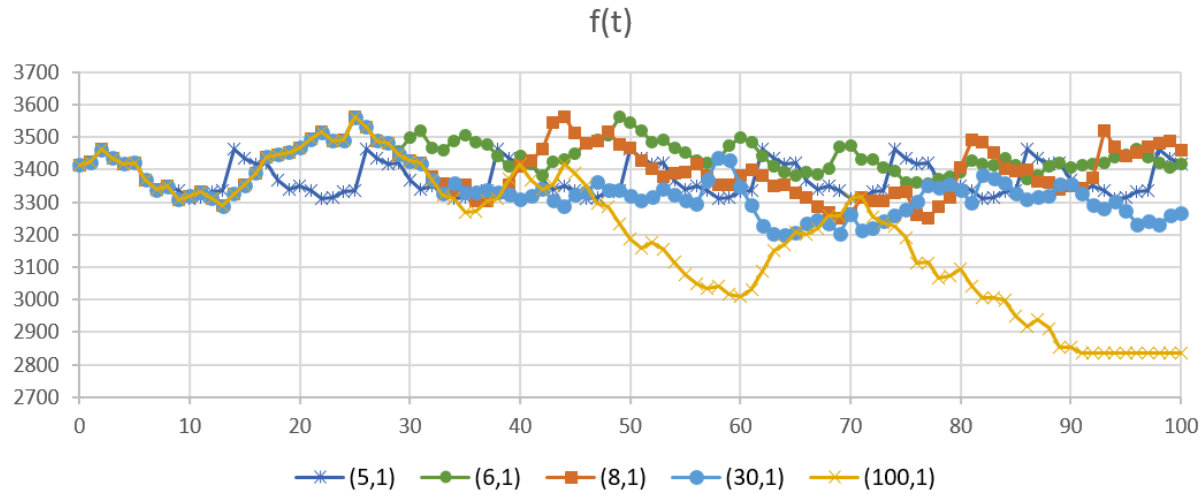


Figure D.7: Profile of the objective function for the *TS* metaheuristic on instance `n0100k010.txt` with different values of tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

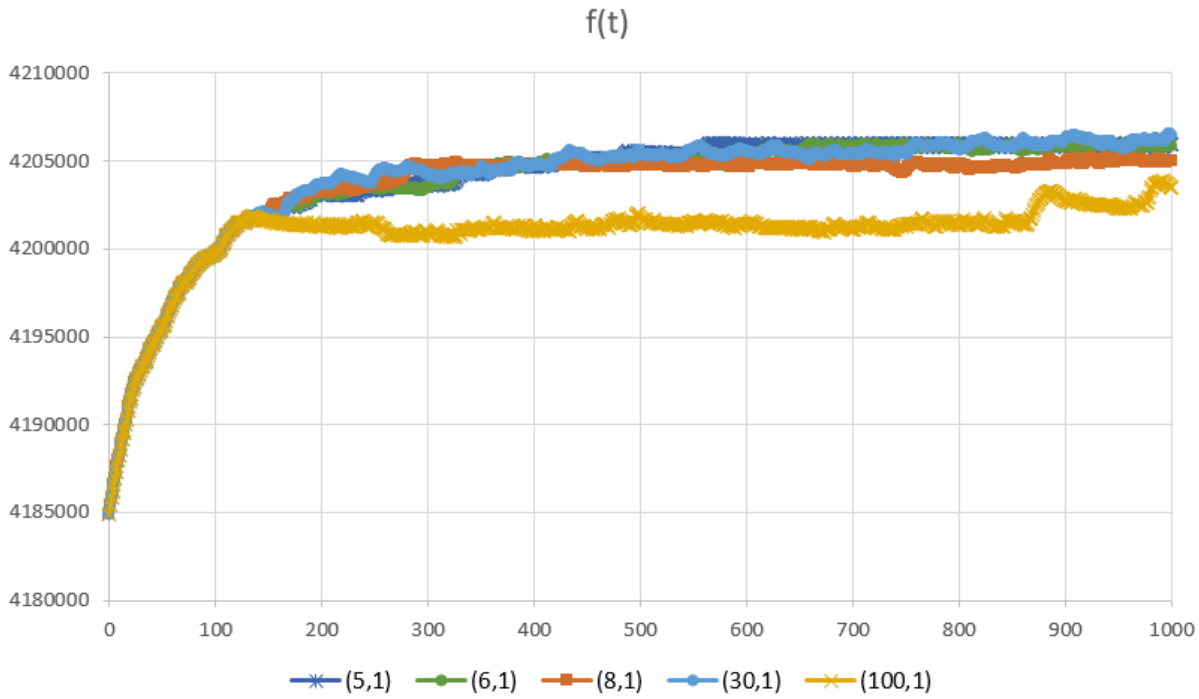


Figure D.8: Profile of the objective function for the *TS* metaheuristic on instance `n1000k400.txt` with different values of tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

Computational time analysis

The overall computational time, including the initialisation procedure is very similar to that of the *VNS* metaheuristic: it ranges from fractions of a second to about four seconds. It is therefore another interesting candidate to provide an efficient solving approach. Figure D.9 reports the logarithmic *scaling diagram* on the whole benchmark. The diagram shows a polynomial increase with size, in good accordance with the $O((n-k)k)$ theoretical estimate, which corresponds to a quadratic complexity when $k \propto n$. In fact, the linear interpolation is:

$$T_A = \beta n^{\alpha} \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

since $\alpha \approx 2.012$ and $\beta \approx 2.7 \cdot 10^{-6}$.

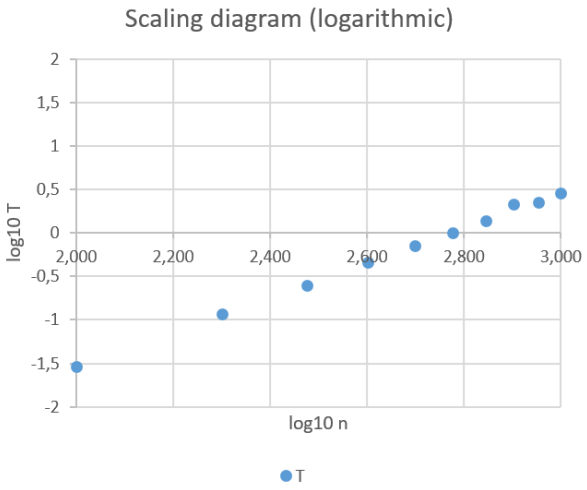


Figure D.9: Scaling diagram in logarithmic scales for the *TS* algorithm on the benchmark

D.3.3 Parameter tuning

We now compare a small number of configurations in which both tenures L^{in} and L^{out} assume different values. For the sake of simplicity, we apply the same values to all instances of the benchmark, also based on the results of the previous experiments, which showed that rather short tenures are enough to avoid cyclic behaviours, and at the same time to allow the search to focus on good quality solutions, both on the smallest and the largest instance. We consider the six configurations obtained setting $L^{\text{in}} = 5$, 6 or 8 and $L^{\text{out}} = 1$ or 2. This is, of course, only a very simple illustrative investigation. Table **??** reports the average gaps over the whole benchmark. They are quite promising: most of them are smaller than the corresponding values obtained by the *VNS* metaheuristic; the larger tenures, in particular, provide the best gap.

Figure D.10 reports the *SQD* diagram of the six configurations, confirming the better performance of the configuration with the larger tenures.

The corresponding boxplots are reported in Figure D.11: they seem to contradict the *SQD* diagram ($L^{\text{in}} = 1$ and $L^{\text{out}} = 8$ look better than $L^{\text{in}} = 2$ and $L^{\text{out}} = 8$), but that mainly depends on the automatic definition of outliers by Excel[3] It seems anyway justified to consider larger tenures as better. Indeed, the experiments should now include larger values, to determine whether it is

L^{in}	L^{out}	Average gap
5	1	0.31%
6	1	0.26%
8	1	0.19%
5	2	0.17%
6	2	0.18%
8	2	0.14%

Table D.2: Average gaps with respect to the best known result of the *TS* metaheuristic with different tunings of the tabu tenures L^{in} and L^{out}

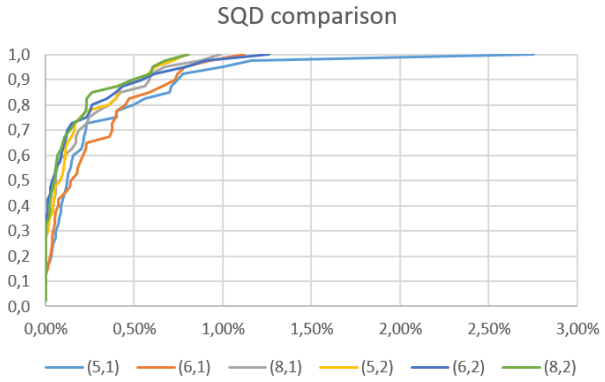


Figure D.10: Solution Quality Distribution diagrams for the *TS* metaheuristic with different tunings of the tabu tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

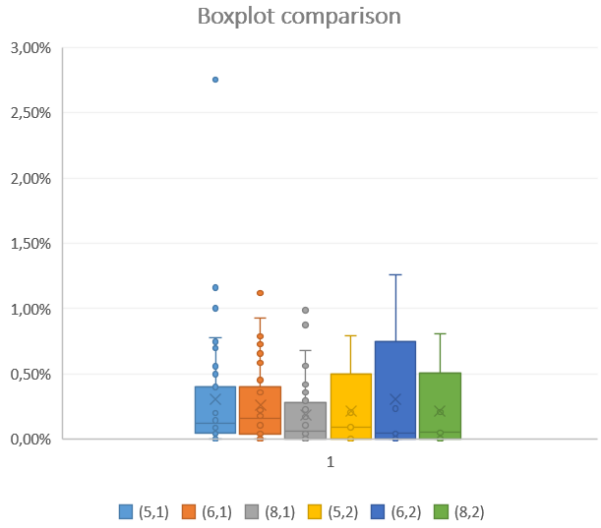


Figure D.11: Boxplot diagrams for the *TS* metaheuristic with different tunings of the tabu tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

possible to further improve the results, but we stop here.

The application of Wilcoxon's test to compare $\left(L^{\text{in}}, L^{\text{out}}\right)$ with the other configurations gives the following results:

1. $(8,2)$ versus $(5,1)$

```
W+ = 432, W- = 64, N = 31, p <= 0.0003233
```

2. $(8,2)$ versus $(6,1)$

```
W+ = 494, W- = 101, N = 34, p <= 0.0008056
```

3. $(8,2)$ versus $(8,1)$

```
W+ = 261.50, W- = 173.50, N = 29, p <= 0.3469
```

4. $(8,2)$ versus $(5,2)$

```
W+ = 313, W- = 152, N = 30, p <= 0.09987
```

5. $(8,2)$ versus $(6,2)$

```
W+ = 248, W- = 187, N = 29, p <= 0.5165
```

which suggests that only the first two comparisons, with configurations having both tenures shorter, are significant (even applying the Bonferroni correction), whereas the other comparisons could easily be the result of a random sampling.

D.4 Comparison between *VNS* and *TS*

To conclude our experiments, we compare the *VNS* and the *TS* metaheuristic, with their best performing configurations, both with respect to the quality of the solutions and the computational time. The corresponding *SQD* and *RTD* diagrams are reported in Figures D.12 and D.13, and show that the *TS* metaheuristic is more effective while taking the same time as the *VNS* algorithm. The result is confirmed by the following response of Wilcoxon's test:

```
W+ = 131, W- = 464, N = 34, p <= 0.00454
```

It is important not to overestimate the range of these conclusions. They refer to a not very large benchmark of a specific nature, to a computation consisting of $t_{\max} = 1\,000$ iterations, to a pair of configurations that have been obtained with a very short investigation. They are however a preliminary result of a certain soundness, obtained with an experimental methodology based on the formulation and verification of hypotheses.

[3] There seems to be a formula to label a value as an outlier. I do not know whether this is a statistic standard or just an Excel convention.

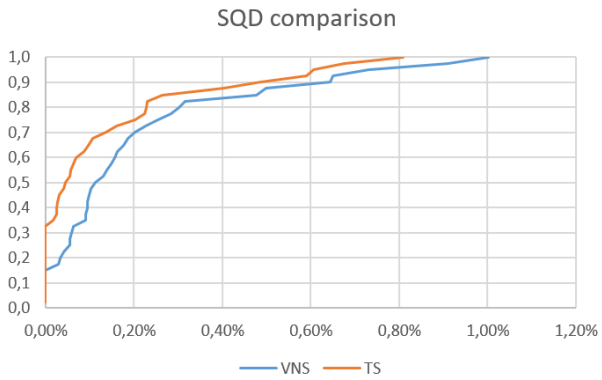


Figure D.12: Solution Quality Distribution diagrams for the *VNS* metaheuristic and the *TS* metaheuristic

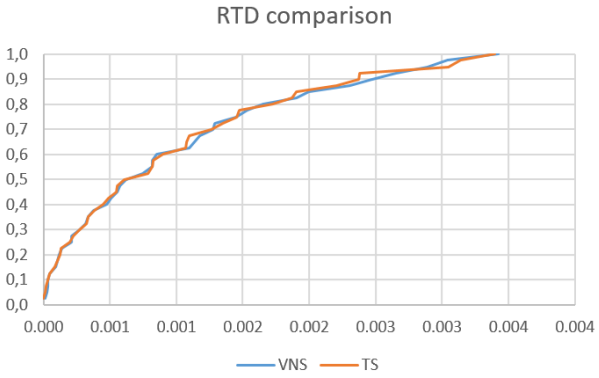


Figure D.13: Run Time Distribution diagrams for the *VNS* metaheuristic and the *TS* metaheuristic

APPENDIX E

Recombination metaheuristics

E.1 Introduction

In this chapter we consider a couple of recombination metaheuristics for the *Maximum Diversity Problem* (*MDP*), namely a *Path Relinking* (*PR*) and a *Scatter Search* (*SS*) approach. Recombination-based approaches do not have a common basic scheme, and they very frequently use randomization or memory. Therefore, the distinction between heuristics and metaheuristics is much less easy to draw than for constructive and exchange approaches. In general, however, they manipulate a reference set of solutions extracting from it suitable subsets and combining the elements of such subsets to generate new solutions. The aim is to retrieve promising portions from different solutions and integrate them in a better way, instead of labouriously modifying the bad portions of a single solution while keeping the good ones.

Since *PR* and *SS* algorithms are mainly deterministic methods based on the idea of exploiting as much as possible the information provided by the data, the initial population usually consists of locally optimal solutions generated by a previous exchange heuristic or metaheuristic. In order to make both methods virtually unlimited, we need a mechanism to generate new solutions even when the recombination mechanism fails to do so. For the sake of simplicity, we generate random solutions and improve each of them to local optimality, as we have done in Chapter 4, to restart the steepest ascent heuristic. In order to understand whether the performance of the two approaches depends on the recombination mechanism or not, we will compare them with the simple steepest ascent heuristic with random restart without applying any recombination procedure. In order to allow a fair comparison among these very different approaches, we will impose a limit on the total computational time as a termination condition.

The neighbourhood used in all three approaches will be the usual single-swap neighbourhood N_{S_1}, explored with the first-best strategy. We will discuss later in detail how the mechanism to generate new solutions is integrated in the schemes of *PR* and *SS*. These will be, therefore, more complicated than the ones reported in the theoretical lessons, which assumed the approaches to terminate as soon as the recombination operations fail to update the reference set. We will consider first *PR* and then *SS*, contrary to the theoretical lessons, because the former is simpler and this will allow to introduce more gradually different concepts and algorithmic components.

The command line arguments of the `main` procedure allow the user to choose which of the three metaheuristics to apply and the associated parameters:

- for the *PR* metaheuristic, option `-pr`, followed by three parameters: the total time $\tau_{\max}$, the cardinality of the reference set $|R|$, and the seed of the pseudorandom number generator;

- for the *SS* metaheuristic, option `-ss`, followed by four parameters: the total time $\tau_{\max}$, the cardinality of the best set $|B|$ and of the diverse set $|D|$, and the seed of the pseudorandom number generator;

- for the *random restart* metaheuristic, option `-rr`, followed by two parameters: the total time $\tau_{\max}$ and the seed of the pseudorandom number generator.

The structure of the `main` function is therefore the usual one, with the exception that no starting solution is generated, because the population is initialized inside the two recombination procedures.

```
parse_command_line(argc,argv,data_file,exchange_algo,&time,&seed,&nb,&nd);

load_data(data_file,&I);

create_solution(I.n,&x);

start = clock();
if (strcmp(exchange_algo,"-ss") == 0)
  scatter_search(&I,&x,"-fb",niter,nb,nd,&seed);
else if (strcmp(exchange_algo,"-pr") == 0)
  path_relinking(&I,&x,"-fb",niter,nb,&seed);
else if (strcmp(algo,"-rr") == 0)
  random_restart(&I,&x,"-fb",tauMax,&seed);
end = clock();
tempo = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6f ",tempo);
printf("%8d ",niter);
print_sorted_solution(&x,I.n);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

E.2 Path Relinking

The *Path Relinking* metaheuristic manages a *reference set*, that is composed of the best known solutions. We will denote it as B (instead of R) to stress the similarities with the design of the *Scatter Search* approach. The basic scheme of *PR* extracts pair of solutions from B and applies an auxiliary exchange procedure to move from the first to the second solution of each pair. Then, it identifies the best solution along each path thus determined and improves it with an exchange procedure. Usually, for the sake of simplicity, the exchange procedure used for both purposes is the same used to generate the solutions of the reference set. The only difference is that, when drawing the relinking path, the objective is to minimize the Hamming distance from the final solution, and only secondarily to optimize the objective of the problem. The locally optimal solution found for each pair is checked to decide whether it is worth inserting in the

reference set. The process terminates when all solutions generated in this way are rejected, and the reference set is not updated.

Therefore, this scheme has an intrinsic termination condition. In order to prolong it indefinitely, if the candidate solutions generated by the relinking paths are less than a given number, new ones are generated at random and improved by steepest ascent. The same process is applied (with no candidate obtained by recombination) to generate the starting reference set. Then, these candidate solutions are tested for insertion in the reference set. This is not the only possible approach: many others are used in the literature, but this one has the advantage of being simple. An important distinction is between the *static update*, which collects the new solutions in a pool and tests them for insertion in the reference set at the end of the recombination phase, and the *dynamic update*, which immediately tests the new solutions. The former approach has the advantage of a simpler implementation and of exploiting all the generated solutions, whereas the latter can remove a solution from the reference set before using it for recombination. On the other hand, the latter approach is more aggressive, as it immediately exploits the new solutions, possibly leading to good results earlier. We will adopt the static update for the sake of simplicity.

The resulting scheme is the following:

Algorithm PathRelinking$(I, \tau_{\max}, n_B)$
$B := \emptyset; P := \emptyset;$
While Time() $\leq \tau_{\max}$ *do*
  { Build or integrate the candidate population }
  While $|P| < n_B$ *do*
    $x :=$ RandomSolution(I);
    $x :=$ SteepestAscent(x);
    If $x \notin P$ *then* $P := P \cup \{x\}$;
  EndWhile
  { Test the candidate solutions for insertion in the reference set }
  For each $x \in P$ *do*
    If $x \notin B$ *then* $B :=$ UpdateBestSet(B, x);
  EndFor
  { Recombine the solutions in the reference set into candidate ones }
  $P := \emptyset$;
  For each $(x, y) \in B \times B$ *do*
    $\Gamma_{x,y} :=$ FindRelinkingPath(x, y, I);
    $z := \arg \max_{w \in \Gamma_{xy} \setminus \{x,y\}} f(w)$;
    $z :=$ SteepestAscent(z);
    If $z \notin P$ *then* $P := P \cup \{z\}$;
  EndFor
EndWhile
$x^* := \arg \max_{x \in B} f(x)$;
Return $(x^*, f(x^*))$;

Function *Time()* returns the time elapsed from the beginning of the algorithm, in order to

enforce the termination condition. In C, this is done with function `clock()`, that returns the number of time units[1] elapsed since the beginning of the execution. For the sake of simplicity, we will call this function only in the outer loop. If performing a single iteration is slow, due to the size and features of the instance, to the size of the reference set or any other factor, the time limit will be actually violated. This should be avoided checking the time also inside the inner loops. Of course, too frequent checks negatively affect the overall computational time.

The first inner loop builds a population of candidate solutions by generating random solutions and improving them with steepest ascent. For the sake of simplicity, the number of candidates is set large enough to fill the reference set (n_B), though in general it could be larger. Duplicate candidates are not accepted, because they would not provide any advantage in the following operations. There is therefore a risk that this loop could not terminate, if many identical locally optimal solutions are obtained. This is rather unlikely, however, with a sufficiently diversifying generation mechanism, such as a random extraction. Notice that in the following iterations of the outer loop, the random candidates will be used only to integrate the solutions generated by recombination when the latter are too few, and that the candidates derived from recombination could be more numerous than n_B, in which case this loop will simply be skipped. Additional parameters could be introduced to tune all these aspects.

The second inner loop tests each candidate solution to check whether it deserves to be included in the reference set B. This is done with a suitable procedure that, in the positive case, adds the solution to update the subset. At the end, the population is cleared, because all its useful elements have been added to the reference set.

The third inner loop considers each pair of solutions (x,y) from B, draws a relinking path from x to y and finds the best solution z on this path (excluding the two extremes). This is then improved by steepest ascent and saved in the population P. The exchange procedure will, as usual, exploit the single-swap neighbourhood, but it will use the global-best strategy when drawing the relinking path, and the first-best strategy when improving the solution. The reason is that the neighourhood used to move between candidate solutions is severely limited by the requirement to strictly reduce the Hamming distance from the final solution, so that a more careful exploration of the neighbourhood seems preferable. When all pairs of reference solutions have been considered, the algorithm starts a new iteration.

The procedures required by this phase of the implementation are in part available in the library `solutionpool`, that manages the allocation and deallocation of solution pools (`create_solutionpool` and `destroy_solutionpool`), the search for a given solution in a pool (`is_in_solutionpool`), the append of a new solution at the end of a pool with residual room (`add_solution_to_pool`), the insertion of a new solution in a pool of solutions sorted by increasing objective values (`update_best_set`), and the removal of all solutions from a pool (`clean_solutionpool`). The generation of a random solution, that was already implemented in Chapter 3, is provided in library `randomsolution`.

The practical implementation creates at the beginning and destroys in the end a current solution `x` and two pools of solutions `B` and `P`, representing the reference set and the population of candidate solutions. Two pools are required by the static update, that first builds all candidates and then tests them for insertion in the reference set. Pool `B` has maximum size `nb`; it is sorted by nondecreasing values of the objective, and solutions are added to it in the second loop by procedure `update_best_set`. Pool `P` is unsorted, solutions are added at the end of the available space in the first and the third loop; its size is `nb*(nb-1)`, because the first loop stops when it

[1] Typically, they are milliseconds or microseconds, depending on the machine. QUESTO VA ANTICIPATO NEL COMMENTO AL MAIN DEL PRIMO CAPITOLO, DATO CHE SI USA ANCHE LI', LASCIANDO QUI SOLO UN RICHIAMO.

reaches `nb` solutions and the third loop considers all pairs of solutions of `B`, though the actual number will probably be much smaller. At the end of the second loop, pool `P` is cleaned: that is simpler and more efficient than deallocating and reallocating it, exactly as it is preferable to clean the auxiliary solution `x` after its use[2]. The only procedure that is used and declared, but not yet implemented, is `find_relinking_solution`, that determines the relinking path and returns its best solution (excluding the two extremes x and y). This is then improved by steepest

[2] An interesting point to discuss on the management of solutions pools is whether to add copies of the new solutions, as it is done at present, or move them physically into the pool. I have not pondered this point enough to take a clear decision.

ascent and included among the candidate ones, if it is not a duplicate.

```
start = clock();
create_solution(pI->n,&x);
create_solutionpool(nb,&B);
create_solutionpool(nb*(nb-1),&P);

while (((double)clock() - start) / CLOCKS_PER_SEC < tauMax)
{
  /* Build or integrate the candidate population */
  while (P.card < nb)
  {
    clean_solution(&x,pI->n);
    generate_random_solution(pI,&x,pseed);
    steepest_ascent(pI,&x,visit_strategy,&iter);
    if (!is_in_solutionpool(&x,pI->n,&P)) add_solution_to_pool(&x,pI->n,&P);
  }

  /* Test the candidate solutions for insertion in the reference set */
  for (s = 1; s <= P.card; s++)
    update_best_set(P.S[s],pI->n,&B);
  clean_solutionpool(&P);

  /* Recombine the solutions in the reference set into candidate ones */
  for (s = 1; s <= B.card; s++)
    for (s2 = 1; s2 <= B.card; s2++)
      if (s != s2)
      {
        clean_solution(&x,pI->n);
        find_relinking_solution(B.S[s],B.S[s2],pI,"-gb",&x);
        steepest_ascent(pI,&x,visit_strategy,&iter);
        if (!is_in_solutionpool(&x,pI->n,&P)) add_solution_to_pool(&x,pI->n,&P);
      }
}
if (B.card > 0) copy_solution(B.S[1],px);

destroy_solution(&x);
destroy_solutionpool(&B);
destroy_solutionpool(&P);
```

The implementation of function `find_relinking_solution` builds upon the steepest ascent procedure. There are two main differences:

1. the procedure minimizes first the Hamming distance from the final solution y, then the objective function (in case of ties);

2. the procedure terminates when solution y is reached.

The special features of the *MDP*, in particular its only constraint fixing the cardinality of the solution, implies that:

1. the only way to reduce the Hamming distance of the current solution z from y (always exactly by 2) is to swap a point in $z \setminus y$ with a point in $y \setminus z$;
2. it is possible to focus on the moves that reduce the Hamming distance and avoid all other moves of N_{S_1};
3. there are $|z \setminus y| \cdot |y \setminus z|$ such moves, that is a positive number, as long as $z \neq y$: the best of these moves will be performed.

Since the size of this neighbourhood and the number of moves is probably much smaller than for a typical single-swap, we will adopt the global-best visit strategy for the relinking path identification. An experimental comparison with the first-best strategy should be performed, but we will not do it for lack of time.

```
create_solution(pI->n,&z);
copy_solution(px,&z);

clean_solution(pz,pI->n);
do
{
  explore_neighbourhood_for_relinking(&z,py,pI,visit_strategy,&p_in,&p_out,&delta_f
  if (p_in != NO_POINT)
  {
    swap_points(p_in,p_out,&z,pI);
    if (z.f > pz->f) copy_solution(&z,pz);
  }

} while (p_in != NO_POINT);
```

The neighbourhood is explored in procedure `explore_neighbourhood_for_relinking` exactly as in the steepest ascent heuristic, with two limitations to guarantee that the solutions explored are closer to the final one than the current solution: the deleted point `p_in` must not belong to the final solution `y`, and the added point `p_out` must belong to the final solution `y`. This is similar to what happens in *Tabu Search*, but much simpler, as it is just a straightforward reduction of the neighbourhood. Notice that, if $k < n/2$, it would be more efficient to scan `p_out` in `py` and check that it is not in `px`.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px); p_in = next_point(p_in,px
  if (!py->in_x[get_index(p_in,pI)])
    for (p_out = first_point_out(px); !end_point_list(p_out,px); p_out = next_point
      if (py->in_x[get_index(p_out,pI)])
      {
        delta_f = evaluate_exchange(p_in,p_out,px,pI);
        if (delta_f > *pdelta_f)
        {
          *pdelta_f = delta_f;
          *pp_in = p_in;
          *pp_out = p_out;
        }
      }
```

E.2.1 Time complexity estimation

An *a priori* estimation of time complexity of the *PR* metaheuristic is quite complex, as it consists of procedures of a very different nature, that include conditions whose occurrence is usually unpredictable. Moreover, in our experiments the computational time is fixed by the user. However, the analysis can suggest the relative weight of the different components of the algorithm, and consequently on which procedures and parameters to focus in order to improve it.

The first inner loop, in which random candidates are generated and improved, is particularly hard to characterize: we know from Chapter 3 that random solutions are usually reduced to a locally optimal one in a number of neighbourhood explorations $t_{\max}$ that increases more than linearly with size. Of course, each exploration takes $O(k(n-k))$ time. The main problem is that this loop ends when the population P includes at least n_B solutions, which can take zero time (if the relinking paths are enough to fill it) or a potentially infinite time (if the random initializations repeat over and over again the same locally optimal solutions). A very rough estimate can be $O(n_B t_{\max} k(n-k))$ time.

The second inner loop tests all candidate solutions for insertion in B. In the worst case, the candidates could be $n_B(n_B-1)$; each one could require to scan all the reference solutions point by point (if all candidates and all reference solutions have the same value, an extremely unlikely case); finally, each candidate could be copied into the pool: overall, that would be $O\left(n_B^3 k + n_B^2 n\right)$ time. In practice, the candidate solutions are usually n_B and most of them are tested against some reference solutions using only the objective value (a single reference solution will be enough to reject bad candidates), so the time could be as low as $O(n_B)$.

The third inner loop draws a relinking path for $n_B(n_B-1)$ pairs of reference solutions. Each path takes at most k neighbourhood explorations and each exploration takes $O(k(n-k))$ time, as usual (probably with smaller multiplying coefficients, due to the limitation of the neighbourhood). Then, the best solution along the path is improved by steepest ascent. Overall, this should be $O\left(n_B^2\left(k^2(n-k)+t_{\max}k(n-k)\right)\right)$ time, where $t_{\max}$ is probably smaller than for a random initialization. This is probably the most expensive component of the algorithm, though the first one could also be relevant. A profiler could confirm or disprove it empirically, but its use exceeds the scope of the present discussion[3].

E.2.2 Empirical evaluation

We can now evaluate the performance of the *PR* metaheuristic. We will experiment with typical values for the number of reference solutions n_B, ranging from 5 to 20. Smaller values would not be enough to generate enough new solution by recombination, whereas larger values would take too much time to systematically test all pairs and would also probably generate many duplicate or bad quality locally optimal solutions. Another possible experiment could be to compare the effect of using the global-best or the first-best strategy in the two exchange procedures that, respectively, improve the recombined and the random solutions or build the relinking paths. For the sake of briefness, we will not perform this analysis.

The only other parameter is the total time. From the previous chapters we know that for the largest instances of the benchmark 1 000 neighbourhood explorations correspond to about 4 seconds and that a randomly generated solution requires around 30 neighbourhood explorations. Each generation of the *PR* framework requires to improve about n_B^2 solutions, some of which are fully random, whereas most derive from the relinking paths. A very rough computation suggests that a generation could take about $20^2 \cdot 30/1\,000 \cdot 4 = 48$ seconds. This is quite long

[3] Next year, may be.

with respect to the time used in the previous chapters, though it is probably an overestimate, because the solutions derived from the relinking paths are probably quicker to optimize. As a compromise, we will set the time to $t_{\max} = 30$ seconds.

Notice that if the time limit is tested only at the beginning of each generation, it could expire before the generation has ended, but the algorithm would terminate only later, violating the limit. This happens also for the random restart approach, because the steepest ascent procedure runs to completion before testing the time limit. To avoid these violations, the condition should be tested more often; possibly, a truncated version of steepest ascent should be designed. However, for the sake of simplicity we will skip this step. Our experiments, in fact, show times ranging from 30 to 30.9 seconds: a more precise termination would be desirable, but is not strictly required for our rough analysis.

E.2.3 Parameter tuning

We now compare the effect on the quality of the solution of four different sizes of the reference set, namely $n_B = 5$, 10, 15 and 20. Table E.1 reports the resulting average gaps over the whole benchmark. For comparison purposes, the table also reports the average gap obtained by the random restart approach. The best performing configurations set $n_B = 15$ and $n_B = 20$.

	PR				*RR*
n_B	5	10	15	20	
Gap	0.26%	0.20%	0.12%	0.12%	0.10%

Table E.1: Average gaps with respect to the best known result of the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (*RR*)

Figure E.1 reports the *SQD* diagram of the four configurations, confirming the relations suggested by the average gap (possibly with a prevalence of $n_B = 15$ on $n_B = 20$, since the latter looks more or less dominated by random restart, whereas the former does not).

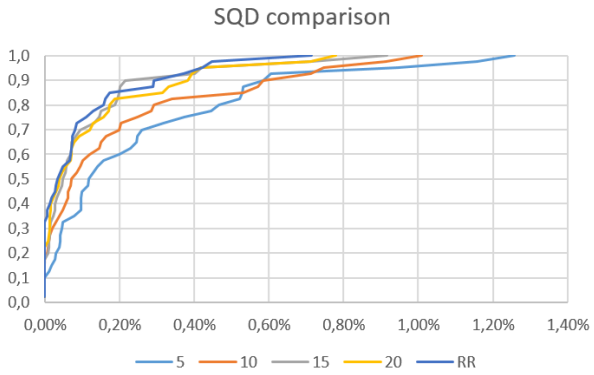


Figure E.1: Solution Quality Distribution diagrams for the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (*RR*)

The corresponding boxplots are reported in Figure E.2, and approximately suggest the same conclusions.

Finally, applying Wilcoxon's test to compare the *PR* algorithm with $n_B = 15$ with the other tunings we obtain the following results:

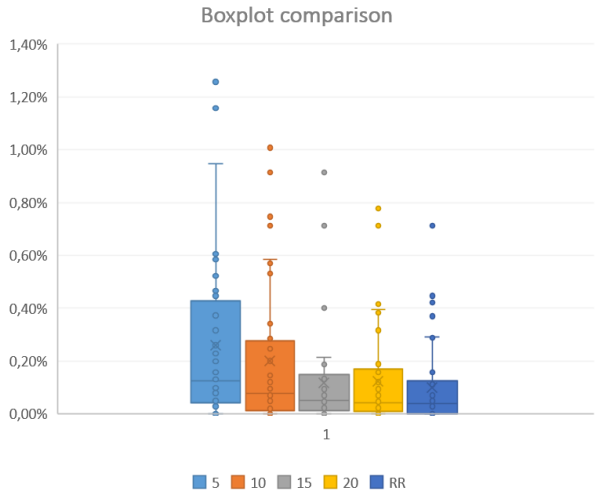


Figure E.2: Boxplot diagrams for the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (*RR*)

1. $n_B = 15$ versus $n_B = 5$

```
W+ = 569, W- = 134, N = 37, p <= 0.001062
```

2. $n_B = 15$ versus $n_B = 10$

```
W+ = 206, W- = 119, N = 25, p <= 0.2473
```

3. $n_B = 15$ versus $n_B = 20$

```
W+ = 144, W- = 109, N = 22, p <= 0.581
```

which suggests that only the first comparison is significant, whereas the other ones could easily be the result of a random sampling. So, the results are inconclusive.

Comparison with random restart

The comparison between the *PR* metaheuristic and the steepest ascent with purely random restart allows to estimate the contribution of recombination to the performance. In fact, the *PR* metaheuristic adopts the same method to initialize the population and to integrate it when recombination is not enough to fill the required minimum number of solutions. From the tables and pictures presented above, all configurations of the *PR* algorithm appear to be worse than random restart: even the best one has an average gap of 0.12% versus 0.10%, and a *SQD* diagram that is dominated in most of the range, though not for all values.

Wilcoxon's test however yields the following results:

1. *RR* versus $n_B = 5$

```
W+ = 411, W- = 184, N = 34, p <= 0.05337
```

2. *RR* versus $n_B = 10$

```
W+ = 301, W- = 294, N = 34, p <= 0.9591
```

3. *RR* versus $n_B = 15$

```
W+ = 247, W- = 314, N = 33, p <= 0.5554
```

4. *RR* versus $n_B = 20$

```
W+ = 185, W- = 221, N = 28, p <= 0.6903
```

While we could expect some tests to be inconclusive, it is rather surprising to find out that all of them are. Analyzing the detailed results, one can remark that random restart seems to perform better on smaller instances and worse on larger ones. Now, our application of Wilcoxon's test considers the absolute differences, whereas the *SQD* considers the relative differences, or percent gaps. The former favours the algorithm that is better on large instances (*PR*), whereas the latter favours the algorithm that is better on small instances (*RR*). This could perhaps explain the results. Anyway, the most correct interpretation is probably to suspend any judgment.

Another interesting insight can be obtained by plotting the values of the solutions met along the relinking paths: we can see that typically the objective is good (that is, large) in the two extreme solutions and decreases in the intermediate ones. Therefore, the best solution is usually the first or the last along the path, that is a neighbour of the given locally optimal solutions[4]: the subsequent exchange heuristic is unlikely to find better solutions. This means that the improvement is mainly due to the random generation step, and therefore increasing the time dedicated to this step by removing the recombination mechanism is actually profitable. In the largest instances, however, this is not always the case: the recombination mechanism starts yielding new solutions, whose quality is better than that of the random ones. These are however just speculations, working hypothesis for further studies, that should be verified checking the actual behaviour of the algorithm on some instances.

E.3 Scatter Search

The *Scatter Search* metaheuristic also manages a *reference set R*, that is however composed of two subsets: B includes the best known solutions, D includes the most diverse solutions. The basic scheme extracts pairs of solutions, one from B and one from $B \cup D$, and combines them with a suitable procedure. Then, it improves the new solution by an exchange procedure and checks whether the locally optimal solution thus obtained is worth inserting in B (first) or in D (then). The process terminates when all solutions generated in this way are rejected, and

[4] A possible idea could be to abandon the standard scheme of *PR* and improve an intermediate solution, instead of the best one along the relinking path.

the reference set is not updated. This scheme has an intrinsic termination condition, at least if the recombination procedure is deterministic. In fact, applying it to the same set of reference solutions, it generates the same locally optimal solutions. If the recombination mechanism is randomized, different solutions could be obtained in different iterations, but the process is very intensifying, so that, even if it is profitable at first, in the long term it is likely to lead to repeated solutions and stagnation.

In order to implement a potentially unlimited algorithm with a user-defined time limit, we exploit the same mechanism used for *PR* in the previous section, that is also the same mechanism used to generate the starting population: new solutions are produced by generating random solutions and improving them with steepest ascent. Then, all solutions of the new population are tested for insertion in the reference set, first in B and then in D. For the sake of simplicity, we will adopt once again a static update mechanism.

The resulting scheme is the following.

Algorithm ScatterSearch$(I, \tau_{\max}, n_B, n_D)$
$B := \emptyset; D := \emptyset; P := \emptyset;$
While Time() $\leq \tau_{\max}$ *do*
  { Build or integrate the candidate population }
  While $|P| < n_B + n_D$ *do*
    $x :=$ RandomSolution(I);
    $x :=$ SteepestAscent(x);
    If $x \notin P$ *then* $P := P \cup \{x\}$;
  EndWhile
  { Test the candidate solutions for insertion in the reference set }
  For each $x \in P$ *do*
    If $x \notin B$ *then* $B :=$ UpdateBestSet(B, x);
    If $x \notin B \cup D$ *then* $D :=$ UpdateDiverseSet(D, x, B);
  EndFor
  { Recombine the solutions in the reference set into candidate ones }
  For each $(x, y) \in B \times (B \cup D)$ *do*
    $P := P \cup$ Recombine(x, y, I);
  EndFor
EndWhile
$x^* := \arg\max_{x \in B} f(x);$
Return $(x^*, f(x^*))$;

Notice the strong structural similarity with the scheme of *PR*. The first inner loop builds a population of candidate solutions by generating random solutions and improving them with steepest ascent. For the sake of simplicity, the number of candidates is set large enough to fill the reference set ($n_B + n_D$). As in *PR*, duplicate candidates are not accepted, because they would provide no advantage, and a (very limited) risk of not terminating the loop exists.

The second inner loop tests each candidate solution to check whether it deserves to be included in either of the two subsets B and D. This is done with suitable procedures that, in the positive case, also add the solution to update the subset. These procedures return `true` or `false` to indicate whether the candidate solution has been accepted or not. A solution added to

B is not tested on D. Since the definition of "diverse" solution refers to both subsets, the update of the diverse set requires set B, or at least information on the Hamming distance of each current solution $y \in D$ from $B \cup D \setminus \{y\}$. At the end, the population P is cleaned, as in PR.

The third inner loop considers each pair of solutions (x, y) from $B \times (B \cup D)$ and recombines them to produce a solution z that is saved in pool P. Now the algorithm goes back to the random generation, in case the recombined solutions are not enough to provide the sufficient number of solutions to update the reference set.

The implementation can exploit many of the procedures already discussed: those provided in library `solutionpool` to manages solution pools, and the generation of random solutions. Of course, we need an additional pool `D`. As mentioned above, the update of the diverse set requires information on the Hamming distance: the two vectors `Hmin` and `Htot` provide, respectively, the minimum and total distance of each reference solution from the other ones. We use both the minimum and total distance to better discriminate between solutions: in fact, the Hamming distance is an integer value between 1 and k, and therefore it could easily assume identical values for several solutions. The update procedure should take care to keep these vectors sorted in the same way as the solutions in D.

Procedure `recombine_solutions` simply recombines two solutions into a third one in one of the several ways discussed in the theoretical lessons. In the present phase, we simply declare

it without defining its content.

```
start = clock();
create_solutionpool(nb,&B);
create_solutionpool(nd,&D);
create_solutionpool(nb*(nb+nd),&P);

create_solution(pI->n,&x);

while (((double)clock() - start) / CLOCKS_PER_SEC < tauMax)
{
  /* Build or integrate the candidate population */
  while (P.card < nb + nd)
  {
    generate_random_solution(pI,&x,pseed);
    steepest_ascent(pI,&x,visit_strategy,&iter);
    if (!is_in_solutionpool(&x,pI->n,&P))
    {
     add_solution_to_pool(&x,pI->n,&P);
    }
    clean_solution(&x,pI->n);
  }

  /* Test the candidate solutions for insertion in the reference set */
  for (s = 1; s <= P.card; s++)
  {
    insert = update_best_set(P.S[s],pI->n,&B);
    if (!insert) insert = update_diverse_set(P.S[s],pI->n,&D);
  }

  /* Recombine the solutions in the reference set into candidate ones */
  for (s = 1; s <= B.card; s2++)
  {
    for (s2 = 1; s2 <= B.card; s2++)
      if (s != s2)
      {
        recombine_solutions(B.S[s],B.S[s2],pI,&x);
        add_solution_to_pool(&x,pI->n,&P);
      }
    clean_solutionpool(&P);

    for (s2 = 1; s2 <= D.card; s2++)
    {
      recombine_solutions(B.S[s],D.S[s2],pI,&x);
      add_solution_to_pool(&x,pI->n,&P);
    }
  }
}

if (nb > 0) copy_solution(B.S[1],px);
```



```
destroy_solutionpool(&B);
destroy_solutionpool(&D);
destroy_solutionpool(&P);
```

`solutionpool`. It is however more complicated because pool `D` is sorted with respect to the (minimum and total) Hamming distance from $B \cup D$. Therefore...

PROBLEM: THE DEFINITION OF POOL DOES NOT INCLUDE FIELDS FOR Htot AND Hmin. USING EXTERNAL VECTORS MAKES THE CODE MORE COMPLEX AND POSES THE QUESTION WHETHER THE FUNCTION SHOULD BE INCLUDED OR NOT IN THE LIBRARY, PLUS THE FOLLOWING ADDITIONAL PROBLEMS: WHEN ARE THE HAMMING DISTANCES OF THE RECOMBINED SOLUTION COMPUTED? WHERE ARE THEY COMPARED WITH THE HAMMING DISTANCES OF THE SOLUTIONS IN D?

`TO BE COMPLETED`

The core of the *SS* metaheuristic is the recombination procedure. Usually, its first step consists in initializing the new solution `x` with the intersection of the two parent solutions `x1` and `x2`. Then, the partial solution is augmented with elements drawn from the two parents. This can be performed in several ways, concerning two main aspects:

- whether the choice of the elements is random or greedy (or any semigreedy combination): the first approach allows a given pair of parent solutions to generate many different new solutions, the second usually generates better solutions;
- whether the elements are chosen alternatively from the two parents or freely: the first approach guarantees the largest possible distance from the two parent solutions, the latter allows a larger variety (in particular, if combined with choices at least partly random).

In the following, for the sake of simplicity, we will draw random elements alternatively from the two parent solutions, counting on the steepest ascent procedure as a tool to generate good solutions (rather than on a greedy choice) and on the recombination of good solutions as a tool to intensify the search (rather than on a biased extraction favouring one parent). It is in general possible that the constraints of the problem forbid to build a whole solution simply drawing random elements from the two parents. In the case of the *MDP*, however, this is possible, thanks to the very simple cardinality constraint. We have therefore simply to create an empty solution, add to it the points that belong to both parents (for example, by scanning one and checking which of its elements also belong to the other), put the other points of both solutions in suitable vectors from which they can be extracted at random, and perform the extraction as already done in several occasions in the previous chapters.

UPDATE: FIRST, SCAN A SOLUTION AND PUT IN THE DESTINATION THE POINTS THAT ARE ALSO IN THE OTHER SOLUTION. THEN, BUILD VECTORS FOR THE REMAINING CANDIDATE POINTS FROM THE TWO SOURCES. FINALLY, ALTERNATIVELY EXTRACT ONE POINT FROM EACH OF THE TWO SOURCES, AND ADD IT TO THE RECOMBINED SOLUTION. THERE IS MUCH IN COMMON WITH OTHER RANDOM EXTRACTIONS USED IN THE PREVIOUS CHAPTERS.

`TO BE COMPLETED`

E.3.1 Time complexity estimation

A time complexity estimation of the *SS* metaheuristic is quite complex, as it consists of procedures of a very different nature, including tests whose outcome is usually impossible to predict *a priori*. The generation of the reference set, for example, requires a random extraction in $O(kn)$

time, where coefficient n depends on the update of vector D. This apparently plays no role, and therefore seems damaging, but it still strongly accelerates the steepest ascent procedure, that probably gives a large contribution to the overall computational time: an assumption to be verified empirically. Then, it requires a steepest ascent procedure, whose complexity has already been characterized as $O(t_{\max}(n-k)k)$, where $t_{\max}$ is the number of neighbourhood exploration from a random solution to the local optimum in which steepest ascent terminates, that is an unknown function of n and k. The generation is repeated until B and D are both full, which requires at least $n_B + n_D$ iterations, but it could require many more, if the locally optimal solutions found are often the same. This depends strongly on chance and on the landscape of the problem (the *MDP* should have a sufficiently rugged landscape to generate many different local optima, but this is just a guess). Then, $n_B(n_B + n_D - 1)$ pairs of solutions are taken into account[5]. For each pair, the recombination procedure requires to find the intersection and the two set differences of the parent solutions (in time $O(k)$) and to add the points of the intersection and random points of the differences to build the new solution (in time $O(n)$ for each of the k points, once again for the update of vector D). Each solution must be checked for insertion in the best set B in time $O(k)$ (in the worst case, that becomes probably quite rare after a while, because it corresponds to improving most of the best solutions at every attempt). If this fails, it must be checked for insertion in the diverse set B in time WHO KNOWS? The main problem is that there is no way to predict how many times these operations will be repeated overall. Therefore, the theoretical analysis is mainly useful to determine the relative weight of the different contributions: the steepest ascent procedure appears to be the most computationally intensive, and this justifies the choice to keep vector D even if it slows down some of the other procedures.

E.3.2 Empirical evaluation

We can now evaluate the performance of the *SS* metaheuristic. We consider reference sets of the same size used for *PR*. In this case, however, we have two subset, B and D. For the sake of simplicity, we assume them to have the same size. Since the pairs of solution to be recombined are $n_B(n_B + n_D - 1)$, we will experiment with the following configurations: $n_B = n_D = 5$, $n_B = n_D = 10$ and $n_B = n_D = 15$. In this way, the number of pairs will not be very different from the one considered by *PR*.

OTHER POSSIBLE PARAMETERS: GREEDY CHOICE VERSUS RANDOM CHOICE; FREE CHOICE VERSUS ALTERNATE CHOICE

Computational time analysis

As for *PR* the computational time is fixed to 30 seconds overall.

TEST WHETHER THE TIME IS VIOLATED DUE TO THE CHECK MADE ONLY AT THE BEGINNING OF EACH ITERATION (PRESUMABLY NOT). IF THIS HAPPENS, ADD OTHER CHECKS.

E.3.3 Parameter tuning

AVERAGE RESULTS

[5]This means that every pair of best solutions is considered twice: once in each direction. The random choice of elements during the recombination suggests that such a repetition could be nonredundant, but this should be verified empirically.

Figure E.3 reports the *SQD* diagram of the *SS* metaheuristic, compared with that of the steepest ascent heuristic applied to purely random solutions. This allows to estimate the contribution of recombination to the overall performance. In fact, the *SS* metaheuristic adopts the same method to initialize the population and to integrate it when recombination is not enough to fill the required minimum number of solutions

Figure E.3: Solution Quality Distribution diagram for the *SS* metaheuristic compared with the *steepest ascent* heuristic initialised with random solutions with a time limit of 30 seconds

The boxplots reported in Figure E.4 provide the same information: SUMMARY OF THE PICTURE

Figure E.4: Boxplots for the *SS* metaheuristic compared with the *steepest ascent* heuristic initialised with random solutions with a time limit of 30 seconds

WILCOXON'S TEST

E.4 Comparison with *PR* and random restart