Integer Linear Programming

1. Discrete linear optimization problems

At the beginning of the course we formulated several problems, some of which implied the use of logical variables (with values 0 or 1) or of integer variables. If the objective function and the constraints are linear, this class of problems is known as *discrete linear optimization*.
Let us consider an *Integer Linear Programming* (ILP) problem of the following type:

$$\begin{aligned} \min\ & cx \\ & Ax \geq b \\ & x \in \boldsymbol{Z}^n_+ \end{aligned}$$

where $\boldsymbol{Z}^n_+$ denotes the set of dimensional vectors n having integer non-negative components. Figure 1 shows the geometric representation of such a problem, where the feasible region is given by the only points falling on the crossings of the squared area.

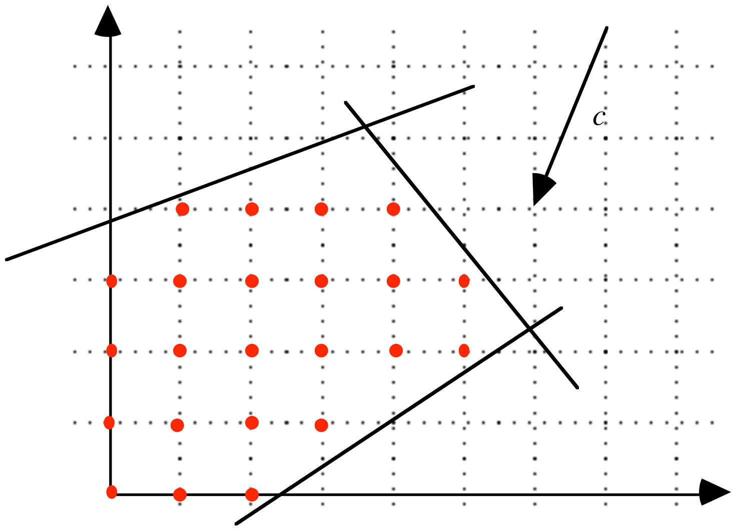

Fig. 1: the points of the feasible region are marked in red

Note that the feasible region (consisting of a discrete set of points) is <u>not</u> a convex set anymore, as it was the case in linear programming. Consequently, the theory we developed for LP, as well as the relevant algorithms we examined, cannot be directly applied to this class of problems.

In addition to integer linear programming problems, the discrete linear optimization class of problems also includes *Mixed Linear Programming* problems (MLP), that is the LP class of problems in which only one subset of variables has integer values. Here is an example of MLP problem:

$$\begin{aligned} \min\ & cx + dy \\ & Ax + Dy \geq b \\ & x \in \boldsymbol{Z}^n_+,\ y \in \mathbf{R}^n \end{aligned}$$

A particular case of integer linear programming is represented by *Combinatorial Optimization* (CO), that is the class of problems in which the feasible region is a subset of the vertices of the unit hypercube $F \subseteq \boldsymbol{B}^n = \{0,1\}^n$, i.e., more simply, problems in which variables can only take value 0 or 1. Linear {0,1} (or binary) programming problems, such as the one exemplified below, belong to this class.

$$\begin{aligned} \min\ & cx \\ & Ax \geq b \\ & x \in \boldsymbol{B}^n \end{aligned}$$

In the previous chapters we saw various CO problems, such as the knapsack problem, the traveling salesperson problem and the spanning tree problem.
Discrete optimization problems are generally difficult to solve. Yet, they represent a very important class of problems because they provide a very effective tool for the modeling of real problems, as we previously saw and as this chapter will show as well.
To round off what already explained in Chapter 1, here we will examine a few formulations of discrete optimization problems.

Example: disjunctive constraints
So far we have seen examples of problems in which the feasible region is described by a convex polyhedron. But in some cases this property might be missing and the feasible region might be given by a non-convex polyhedron. Let us consider, for example, the feasible region described in Figure 2.

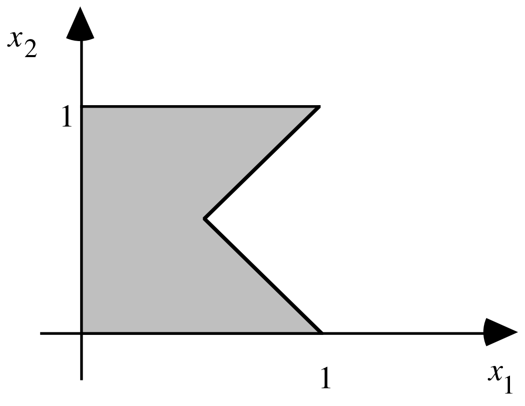

Fig. 2: example of non-convex feasible region

As illustrated in Figure 3, the region can be described as a union of the two convex polyhedra *P* and *Q*.

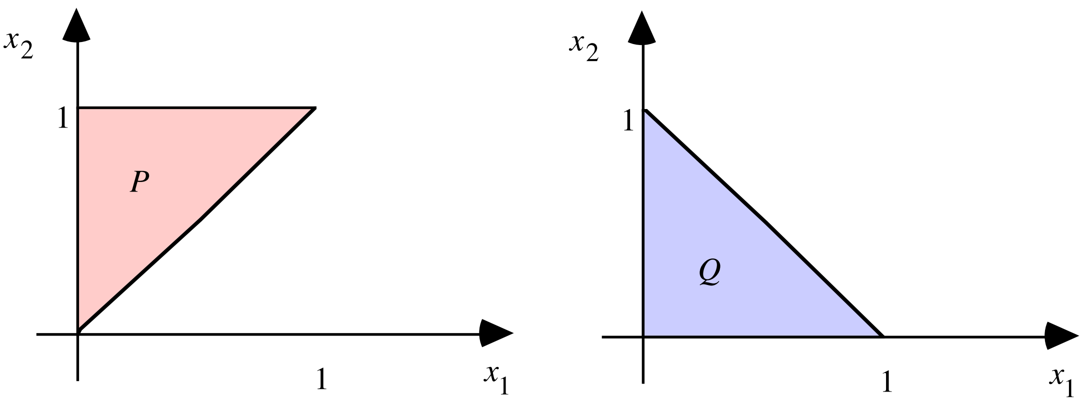

Fig. 3: decomposition of the polyhedron

The polyhedron *P* is given by $P=\{x\in\boldsymbol{R}^2: x_1 - x_2 \leq 0, x_2\leq 1, x_1\geq 0, x_2\geq 0\}$, whereas $Q=\{x\in\boldsymbol{R}^2: x_1 + x_2 \leq 1, x_1\geq 0, x_2\geq 0\}$. We know that, in order to find the intersection of the two polyhedra, it suffices that all constraints of *P* and *Q* hold at the same time; otherwise, if we are seeking the union, we have to impose that at least one of the two groups of constraints holds. First of all, we observe that some constraints are common to the two polyhedra, in particular $\{x_2\leq 1, x_1\geq 0, x_2\geq 0\}$, so we can actually perform the intersection of these constraints, whereas the constraints more specifically characterizing the two feasible regions are $x_1 - x_2 \leq 0$ as regards *P* and $x_1 + x_2 \leq 1$ as regards *Q*. We have to take care that in our formulation at least one of these two constraints holds; considering the problem from the opposite point of view, this means that at most one of these two constraints can be made redundant. As illustrated in Figure 4, canceling one of these constraints means translating it by a quantity sufficient to make it include the whole feasible region of the other polyhedron.

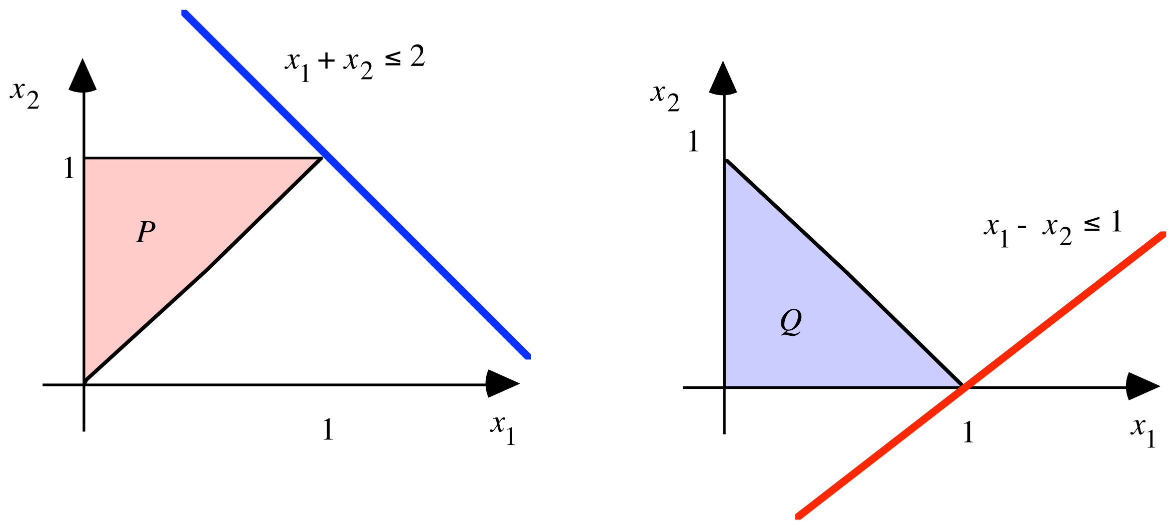

Fig. 4: translation of constraints

In order to perform the operation of canceling at most one constraint (or one group of constraints) of the two that had been found, we introduce the variables 0-1 y_P and y_Q having value 1 if we decide to cancel the constraint relative to P or to Q, respectively. Hence, the formulation of the region shown in Figure 2 becomes:

$$
\begin{aligned}
&x_2 \leq 1\\
&x_1 \geq 0\\
&x_2 \geq 0\\
&x_1 - x_2 \leq 0 + M_P y_P\\
&x_1 + x_2 \leq 1 + M_Q y_Q\\
&y_P + y_Q \leq 1\\
&y_P, y_Q \in \{0,1\}
\end{aligned}
$$

The constraint $y_P + y_Q \leq 1$ allows to cancel at most one of the two groups of constraints. The choice of the two constants M_P and M_Q has to be such as to guarantee that the feasible region is completely contained inside the translated constraint. Therefore, in the examined case we have $M_P \geq 1$ and $M_Q \geq -1$.

A particular case in the application of the technique of disjunctive constraints concerns machine task scheduling problems. Let us pose the problem of processing n tasks $1,\ldots,n$ on m machines denoted by $1,\ldots, m$. Each task must be first performed on machine 1, then on machine 2, and so on up to the last machine m. The execution time of task j on machine k is given by s_{jk}, $j=1,\ldots,n$ and $k=1,\ldots,m$. The different machines can perform one task at a time, and the tasks performed by each machine must be brought to an end, i.e., they cannot be interrupted and resumed. The problem consists in determining the sequencing of the different tasks on the different machines, trying to minimize the end time of the last task (in technical language called *makespan*). To determine the sequencing we need variables t_{jk} that set the beginning time of task j on machine k. These variables make it easy to formulate the constraints relative to the making sequence of each task:

$$t_{jk} + s_{jk} \leq t_{jk+1} \qquad j=1,\ldots,n,\ k=1,\ldots,m-1$$

Task j cannot start to be performed on machine k+1 before having been completed by machine k. In order to properly represent the objective function, which is of the bottleneck type (min max), we introduce a variable T that will be greater than or equal to the end time of the last task on the last machine:

$$t_{jm} + s_{jm} \leq T \qquad j=1,\ldots,n$$

Obviously, the objective function will tend to minimize T, so that in the optimal solution T will precisely take the end time of the last task:

$$\min \quad T$$

Now we just have to formulate the constraints preventing to perform two tasks on the same machine simultaneously. This idea of non-simultaneity is expressed by considering each pair of tasks *i* and *j* and each machine *k*, and by imposing that on machine *k* either *i* precedes *j* or *j* precedes *i*, that is:

$$t_{ik} + s_{ik} \leq t_{jk}$$

or

$$t_{jk} + s_{jk} \leq t_{ik}$$

To express the disjunction between the two constraints we need a variable 0-1 y_{ijk} having value 1 if *i* precedes *j* on *k* and 0 otherwise. The (disjunctive) constraints are:

$$\begin{array}{ll} t_{ik} + s_{ik} \leq t_{jk} + M(1-y_{ijk}) & i,j=1,\dots,n,\ (i\neq j),\ k=1,\dots,m \\ t_{jk} + s_{jk} \leq t_{ik} + My_{ijk} & i,j=1,\dots,n,\ (i\neq j),\ k=1,\dots,m \end{array}$$

where *M* is a value being high enough.

If we generalize the technique of disjunctive constraints that we saw for the non-convex region case and for the scheduling problem, we can now consider the problem of determining a solution satisfying at least *k* ($1 \leq k \leq n$) of the constraints:

$$A_i x \leq b_i, \quad i = 1,\dots,n.$$

Let $M \in \boldsymbol{R}$: $A_i x - b_i \leq M$, for each feasible x, $i = 1,\dots,n$.
The problem can be formulated as follows:

$$\begin{array}{lll} A_i x & \leq b_i + My_i, & i = 1,\dots,n, \\ \sum_{i=1}^{n} y_i & \leq n - k, & \\ y \in \{0,1\}^n. & & \end{array}$$

Exercise
Formulate the following problem as ILP: min $\{cx: x \in X\}$, where the feasible region X is indicated in Figure 5.

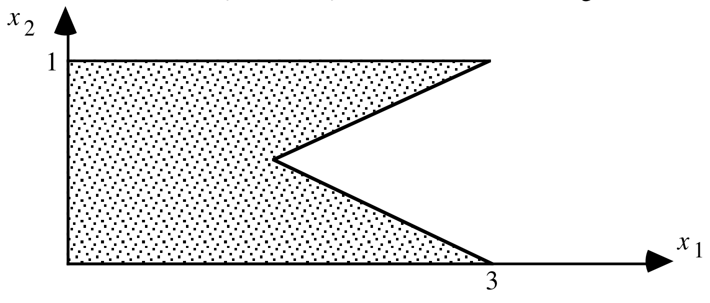

Fig. 5

Example: piecewise linear functions
Before Italian telephony was deregulated, the rates for private subscribers were the following: for the first 140 units the cost was 50 liras, from the 141th to the 220th unit the cost was 207 liras and for the following units it was 127 liras. Figure 6 shows a scheme of such cost function, where b_0 represents a fixed cost (for example the bimonthly fee).

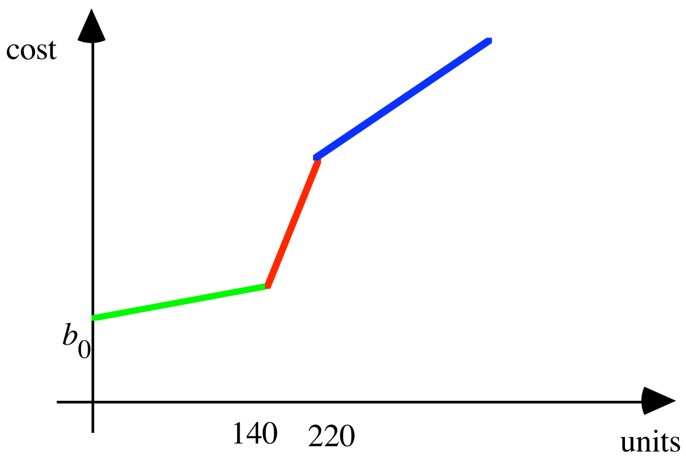

Fig. 6: old Sip rates for private subscribers

Hence, if x indicates the number of units and $f(x)$ the cost function, we formally have:

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ b_0 + 50x & \text{if } 0 < x \leq 140 \\ b_1 + 207x & \text{if } 140 < x \leq 220 \\ b_2 + 127x & \text{if } 220 < x \end{cases}$$

where b_1 and b_2 are the values of the intercept of the red and the green lines represented in the figure. If we want to use this cost function in an optimization problem, we have to face the difficulty of an objective function which is not linear but *piecewise linear*. A way of dealing with these functions is to use integer linear programming models. The idea is to deal with the three intervals [0,140], [140,220], [220,M] separately, with M being an appropriately large constant. Let us introduce the continuous variables z_0, z_1 and z_2 indicating how many units are spent in the first, in the second and in the third band, respectively. We also have to introduce a mechanism preventing to spend the units of an interval if the units of the preceding interval have not already been used up. This is done by introducing variables 0-1, one for each interval, (y_0, y_1 and y_2) having value 1 only if we have started to spend some units in the interval they refer to. The objective function and the constraints that operate the mechanism we mentioned are the following:

$$\begin{array}{lllll} \min & b_0y_0 + 50z_0 + 207z_1 + 127z_2 \\ & 140\, y_1 & \leq\ z_0 & \leq & 140\, y_0 \\ & 80\, y_2 & \leq\ z_1 & \leq & 80\, y_1 \\ & 0 & \leq\ z_2 & \leq & M\, y_2 \\ & x = z_0 + z_1 + z_2 \\ & y_0, y_1 \text{ and } y_2 \in \{0,1\} \end{array}$$

Note that if, for example, $z_1>0$ then y_1 must be equal to 1, which implies that z_0 must be equal to 140, thus we must have used up all the units of the preceding interval.
Wishing to apply what we saw for the telephone rate example to a more general context, we have to consider the following piecewise linear function to be minimized, whose graphical representation is shown in Figure 7:

$$f(x)=\begin{cases} 0, & \text{if } x = a_0, \\ b_0 + c_0(x - a_0), & \text{if } a_0 < x \leq a_1, \\ b_0 + c_0(a_1 - a_0) + b_1 + c_1(x - a_1), & \text{if } a_1 < x \leq a_2, \\ \dots & \dots \\ \sum_{i=1}^{k-1} [b_{i-1} + c_{i-1}(a_i - a_{i-1})] + b_{k-1} + c_{k-1}(x - a_{k-1}), & \text{if } a_{k-1} < x \leq a_k. \end{cases}$$

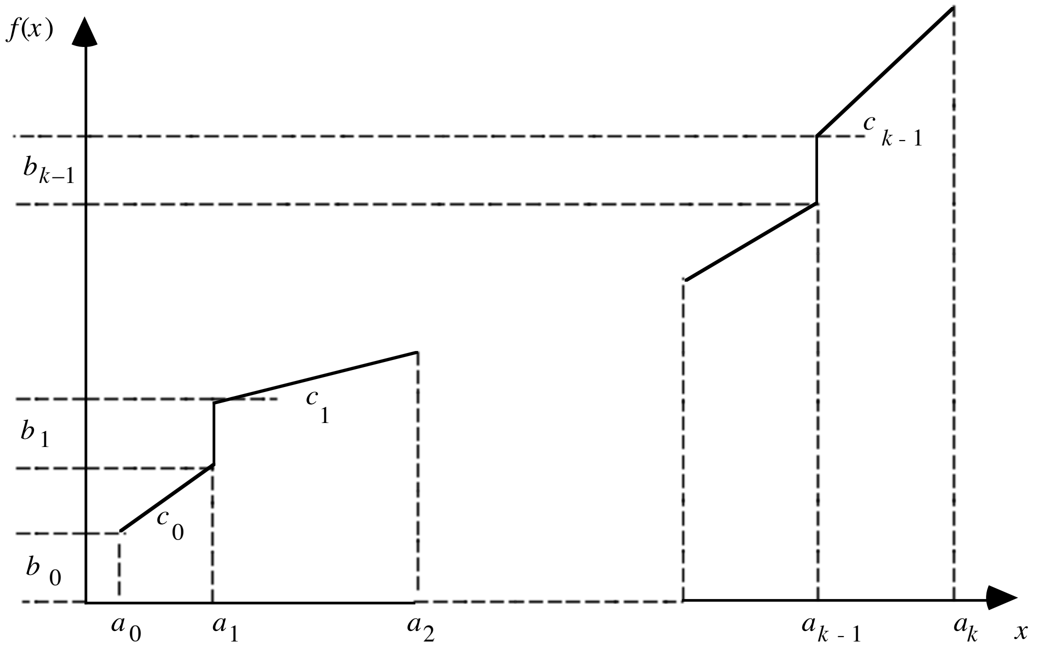

Fig. 7: general piecewise linear function

Problem *P* can be formulated as an ILP problem as follows:

$$
(P') \qquad \min \sum_{i=0}^{k-1} b_i y_i \quad + c_i z_i
$$

$$
\begin{aligned}
&(a_1 - a_0)y_1 \;\le\; z_0 \;\le (a_1 - a_0)y_0,\\
&(a_2 - a_1)y_2 \;\le\; z_1 \;\le (a_2 - a_1)y_1,\\
&\qquad\cdots \qquad\quad \cdots \qquad \cdots\\
&\qquad\qquad 0 \;\le\; z_{k-1} \;\le (a_k - a_{k-1})y_{k-1},\\
&x = \sum_{i=0}^{k-1} z_i \; + a_0,\\
&y_i \in \{0,1\},\, z_i \in \boldsymbol{R},\, i = 0,\ldots, k\text{-}1.
\end{aligned}
$$

Let $(\bar{y}, \bar{z}, \bar{x})$ be a feasible solution for *P*'. We want to show that it gives

$$
f(\bar{x}) = \sum_{i=0}^{k-1} b_i \bar{y}_i + c_i \bar{z}_i.
$$

Consider $\bar{y}$. Assume that it gives $\bar{y}_{r-1} = 1$, $\bar{y}_r = 0$, $1 \le r \le k\text{-}1$.

$$
\bar{y}_r = 0 \quad \Rightarrow \quad \begin{cases} \bar{y}_i = 0, & i \ge r, \\ \bar{z}_i = 0 & i \ge r; \end{cases}
$$

$$
\bar{y}_{r-1} = 1 \Rightarrow \quad \begin{cases} \bar{y}_i = 1, & i \le r\text{ -}1, \\ \bar{z}_{i-1} = a_i - a_{i-1} & i \le r\text{ -}1, \\ 0 \le \bar{z}_{i-1} \le a_i - a_{i-1}, & i = r; \end{cases}
$$

Hence:

$$\sum_{i=0}^{k-1} b_i\bar{y}_i + c_i\bar{z}_i = \sum_{i=0}^{r-1} b_i + \sum_{i=1}^{r-1} c_{i-1}(a_i - a_{i-1}) + c_{r-1}\bar{z}_{r-1}$$
$$= \sum_{i=0}^{r-1} b_i + \sum_{i=1}^{r-1} c_{i-1}(a_i - a_{i-1}) + c_{r-1}(\bar{x} - a_{r-1}).$$

The last equality derives from the fact that, according to construction, we have:

$$\bar{x} = a_{r-1} + \bar{z}_{r-1} \text{ with } 0 \le \bar{z}_{r-1} \le (a_r - a_{r-1}),$$

hence $f(\bar{x}) = \sum_{i=0}^{k-1} b_i\bar{y}_i + c_i\bar{z}_i$.

In some particular cases it is not necessary to introduce variables 0,1 in order to induce the sequencing mechanism in the growth of variables z. One of these cases is, for instance, the minimization of a convex piecewise linear function, as exemplified in Figure 8.

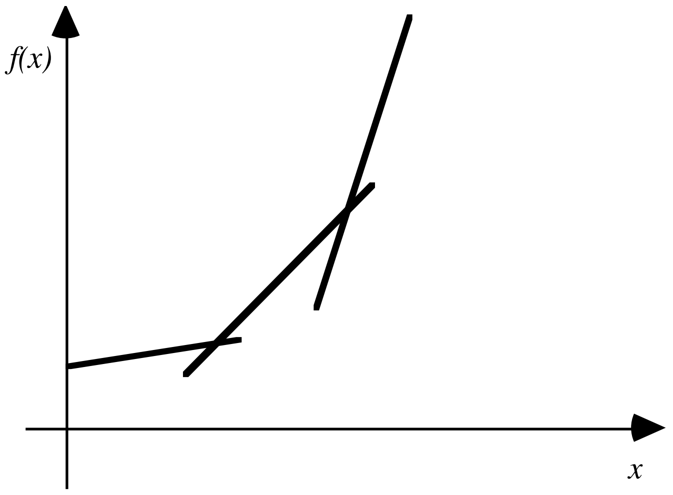

Fig. 8: convex piecewise linear function

With such function we can intuitively understand that it is more advantageous to use the initial part of the function, with lower unit costs, before moving on to the following parts. Clearly, such functions can be formally represented using the technique of bottleneck objective functions that we saw in the first chapter. Let there be the convex piecewise linear function:

$$f(x) = \begin{cases} b_0 + c_0 x & \text{if } 0 \le x \le a_0 \\ b_1 + c_1 x & \text{if } a_0 < x \le a_1 \\ \dots & \dots \\ b_n + c_n x & \text{if } a_{n-1} < x \end{cases}$$

If we consider the different lines, including those outside the interval definition field, we see that the function is always given by the higher line, therefore:

$$f(x) = \max\{b_i + c_i x,\ i=0,\dots n-1\}.$$

Exercise

Consider a concave piecewise linear objective function $f(x)$. Formulate the problem of maximizing $f(x)$ in terms of linear programming.

Example: the traveling salesperson problem on a directed graph

Consider the traveling salesperson problem, i.e., the problem of determining a minimum cost Hamiltonian cycle on a weighted symmetric graph $G=(N,A,c)$ where $|N| = n$. In Chapter 2 we saw a possible formulation of the traveling salesperson problem on undirected graphs; let us now examine the directed case.

Let $x = [x_{ij}]$ be a vector of binary variables with the following meaning:

$$x_{ij} = \begin{cases} 1, \text{ if arc } (i,j) \text{ belongs to the Hamiltonian cycle we look for,} & 0, \\ \text{otherwise.} \end{cases}$$

A first set of constraints says that exactly one arc must arrive at and leave from each node:

$$X_1 = \{x: \sum_{i=1}^{n} x_{ij} = 1, \sum_{j=1}^{n} x_{ij} = 1, x_{ij} \in \{0,1\}, i = 1,..., n, j = 1,..., n \}.$$

Just like in the undirected case, we have to remove subcycles, i.e., cycles not traversing all the graph nodes. A possible way to do this is the following:

$$X_2 = \{x: \sum_{i \in N_1} \sum_{j \in N_2} x_{ij} \geq 1, \forall (N_1, N_2,) : N = N_1 \cup N_2, N_1 \cap N_2 = \emptyset, N_1 \neq \emptyset, N_2 \neq \emptyset\},$$

Considering any partition of nodes into two non-empty subsets, the number of arcs in the solution with tail in the first subset and head in the second one must be greater than or equal to 1. An alternative way is to introduce other (continuous) variables u_i that determine the order according to which each node i is visited. Constraints are used to define the value of variables determining the order.

$$X_3 = \{x: u_i - u_j + nx_{ij} \leq n\text{-}1 , u_i \in \mathbf{R}, i = 1,..., n, j = 2,\ldots n, i \neq j \}.$$

The set X_3 merely includes cycles containing node 1. In fact, we consider a cycle composed of $k \leq n$-1 nodes and not containing node 1; by adding together the constraints $u_i - u_j + nx_{ij} \leq n$-1 corresponding to the arcs of the cycle, we get the contradiction $kn \leq k$ (n-1). Observe that X_3 does <u>not</u> include the constraints $u_i - u_1 + nx_{ij} \leq n$-1, $u_i \in \mathbf{R}$, $i = 1,..., n$, hence it is satisfied by cycles containing node 1. Consequently, the set $X_1 \cap X_3$ defines the set of Hamiltonian cycles on G.
We can see now that variables u_i provide the search order in a feasible solution.
If arc (i,j) belongs to the Hamiltonian cycle ($x_{ij} = 1$), the corresponding constraint in X_3 becomes:

$$u_i + 1 \leq u_j.$$

Observe that a vector $\bar{u}$ satisfying the preceding constraints is given by $\bar{u}_1 = 0$, $\bar{u}_i = k$, $i \neq 1$, if i is the k+1-th node of the cycle starting from node 1.
If arc (i,j) does not belong to the Hamiltonian cycle ($x_{ij} = 0$), the corresponding constraint in X_3 becomes:

$$u_i - u_j \leq n\text{-}1,$$

which is always satisfied by $\bar{u}$ because $\bar{u}_i \leq n$ -1 and $\bar{u}_j \geq 0$.
The traveling salesperson problem can be defined in one of these two alternative ways:

$$\text{TSP1:} \quad \min \{ \sum_{i,j=1}^{n} c_{ij} x_{ij} : x \in X_1 \cap X_2\};$$

$$\text{TSP2:} \quad \min \{ \sum_{i,j=1}^{n} c_{ij} x_{ij} : x \in X_1 \cap X_3\}.$$

Observe that TSP1 is defined by $O(2^n)$ constraints, whereas TSP2 is defined by $O(n^2)$ constraints.

<u>Example: selection of subsets</u>
Many problems having a practical importance are formulated as a selection of subsets belonging to a specific family. For example, problems of staff shifts belong to this class of problems.
Let there be:
- a set $I = \{1, 2,\ldots, n\}$,

- a family F of m subsets of I : $F = \{J_1, J_2, \ldots, J_m\}$, $J_j \subseteq I$, $j = 1,\ldots, m$,
- the cost c_j of the subset J_j, $j = 1,\ldots, m$.

The purpose is to determine a minimum cost family $D \subseteq F$, while respecting the constraint that each element of I belongs to:

i) at least one subset of D (*cover* problem);
ii) exactly one subset of D (*partitioning* problem);
iii) at most one subset of D (*packing* problem).

For example, in the case of shifts of airline pilots the set I is given by the flights to be covered, whereas each subset J_j is defined by the flights that can be performed by the same pilot during a working week, while respecting union rules as well as the compatibility of flights depending on simple time constraints and on the type of aircraft. Finally, c_j is the salary that must be paid to the pilot for a specific work shift. It is a partitioning problem (we have to assign a pilot to each flight) but, since it may be quite difficult to reach a feasible solution, we can consider it as a cover problem, taking the risk to have more than one pilot on the same flight.

If we formulate the above problems in terms of integer linear programming, the family F can be represented by the matrix $A = [a_{ij}]$, $i = 1,\ldots, n, j = 1,\ldots, m$, where:

$$a_{ij} = \begin{cases} 1, & \text{if } i \in J_j, \\ 0, & \text{otherwise.} \end{cases}$$

If we indicate the subset choice variables by

$$x_j = \begin{cases} 1, & \text{if the set } J_j \in D, \\ 0, & \text{otherwise,} \end{cases}$$

then:

- the *cover* problem can be formulated as:

$$\begin{aligned} \min \quad & cx \\ & Ax \geq e \\ & x \in \{0,1\}^m \end{aligned}$$

  where e denotes the vector whose components are equal to 1;
- the *partitioning* problem can be formulated as:

$$\begin{aligned} \min \quad & cx \\ & Ax = e \\ & x \in \{0,1\}^m; \end{aligned}$$

- the *packing* problem can be formulated as:

$$\begin{aligned} \min \quad & cx \\ & Ax \leq e \\ & x \in \{0,1\}^m. \end{aligned}$$

<u>Example: vehicle routing</u>

A classical optimization problem in the transportation sector is the *Vehicle Routing Problem*. Let there be: a set of clients to be served $\{1,\ldots,n\}$, each having a demand of good d_i, and a set of k vehicles that, for more simplicity, we assume to be homogeneous, i.e., all having equal capacity Q and all kept in the same depot we indicate with 0. The distances between each pair of clients and between the depot and each client (c_{ij}, $i,j = 0,\ldots,n$) are known as well. The problem consists in delivering the goods to all clients minimizing the total distance covered by the vehicles and respecting their capacity constraints. To support the formulation, we can consider a complete graph with nodes corresponding to clients and to the depot. A feasible solution consists in a family of m ($m \leq k$) cycles having origin in 0 such that all clients belong to one cycle. We can decompose the decisions into two levels: at the first level we assign clients to vehicles. To do so we use assignment variables:

$$y_{ih} = \begin{cases} 1, & \text{if node } i \text{ belongs to the route of vehicle } h, \\ 0, & \text{otherwise.} \end{cases}$$

The constraints referring to these variables are:

$$\sum_{h=1}^{k} y_{ih} = 1 \qquad i=1,\ldots,n$$

that is, each client must belong to a route. We can also have a bound on the number of used vehicles by imposing a constraint on the number of times node 0 is assigned to a route:

$$\sum_{h=1}^{k} y_{1h} \leq m$$

Constraints on vehicle capacity can also be easily expressed by imposing that the sum of demands by the clients assigned to the same vehicle does not exceed the maximum capacity, and we do so using the constraints we saw for the knapsack problem:

$$\sum_{i=1}^{n} d_i\, y_{ih} \leq Q \quad h=1,\ldots,k$$

If now we want to measure the distance covered by each vehicle, we have to solve as many traveling salesperson problems as the number of vehicles that are used. So, we introduce variables x_{ij}^{h} in order to determine the route of vehicle h:

$$x_{ij}^{h} = \begin{cases} 1, & \text{if vehicle } h \text{ goes from node } i \text{ to node } j, \\ 0, & \text{otherwise,} \end{cases} \qquad i, j = 0,\ldots,n,\ h = 1,\ldots,k.$$

The feasibility constraints in the route choice (an incoming arc and an outgoing arc for each node and subcycle removal) are:

$$\sum_{j=1, j\neq i}^{n} x_{ij}^{h} = y_{ih} \qquad i = 0,\ldots,n,\ h=1,\ldots,k$$

$$\sum_{j=1, j\neq i}^{n} x_{ji}^{h} = y_{ih} \qquad i = 0,\ldots,n,\ h=1,\ldots,k$$

$$\sum_{i,j\in S} x_{ij}^{h} \leq |S|-1 \qquad S\subseteq\{1,\ldots,n\}\ |S|\geq 2,\ h=1,\ldots,k.$$

We can remove a group of variables and simplify the model by observing that

$$y_{ih} = \sum_{j=1, j\neq i}^{n} x_{ij}^{h}$$

The complete formulation of the vehicle routing problem becomes:

$$\min \quad \sum_{h=1}^{k}\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}^{h}$$

$$\sum_{j=1}^{n} x_{ij}^{h} = \sum_{j=1}^{n} x_{ji}^{h} = y_{ih} \qquad i = 0,\ldots,n,\ h=1,\ldots,k$$

$$\sum_{h=1}^{k}\sum_{j=1}^{n} x_{ij}^{h} = 1 \qquad i = 1,\ldots,n,$$

$$\sum_{i=1}^{n}\sum_{j=0}^{n} d_i x_{ji}^{h} \le Q \qquad h=1,\ldots,k.$$

$$x_{ij}^{h} \in \{0,1\}, \qquad i,j = 0,\ldots,n,$$

Example: frequency assignment

In a cellular telecommunication network the territory is served by various antennas, each covering a specific *cell*. Cells overlap each other in order to ensure continuity of service to users while they move. A classical problem in this field is to assign frequencies to cells so as to prevent that interfering cells (for example those that overlap) use the same frequency. A purpose can be, for example, to minimize the number of used frequencies.

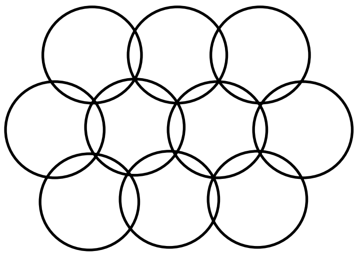

Fig. 9: cellular system

The problem makes use of a representation on a graph. Let us build an undirected graph $G= (N,A)$ where nodes correspond to the cells of the telecommunication network and between two nodes there is an arc $\{i,j\}$ if the corresponding cells interfere.

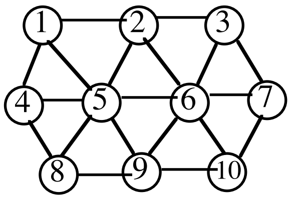

Fig. 10: graph of interferences

Having k available frequencies, in order to formulate the problem we use variables assigning nodes to frequencies:

$$x_{ih} = \begin{cases} 1, & \text{if node } i \text{ is assigned the frequency } h, \\ 0, & \text{otherwise.} \end{cases}$$

Further, we need a set of variables determining the actual use of frequencies.

$$y_{h} = \begin{cases} 1, & \text{if the frequency } h \text{ is assigned to some node,} \\ 0, & \text{otherwise.} \end{cases}$$

The problem formulation is:

$$\begin{aligned} \min \quad & \sum_{h=1}^{k} y_h \\ & x_{ih} + x_{jh} \le 1 \quad \forall\{i,j\}\in A \\ & x_{ih},\, y_h,\ \forall i\in N,\ h=1,\dots,k. \end{aligned}$$

This problem on graphs is known in the literature as *coloring* problem: the purpose is to color the nodes of a graph in such a way that adjacent nodes are differently colored, while minimizing the number of colors. A well-known case is the coloring of geographical maps.

2. Choice of the formulation and "polyhedral" methods of solution

Unlike in continuous linear programming, in ILP the choice of the formulation has a strong influence on the efficiency of the methods of solution, and often proves to be a decisive factor. Let us consider the following example.

$$\begin{aligned} \max \quad & x_1 + 0.64\, x_2 \\ & 50x_1 + 31x_2 \le 250 \\ & 3x_1 - 2\,x_2 \ge -4 \\ & x_1, x_2 \ge 0 \text{ integers.} \end{aligned}$$

In Figure 11 we indicate the feasible region provided by the integer points contained in the polyhedron defined by the linear constraints.

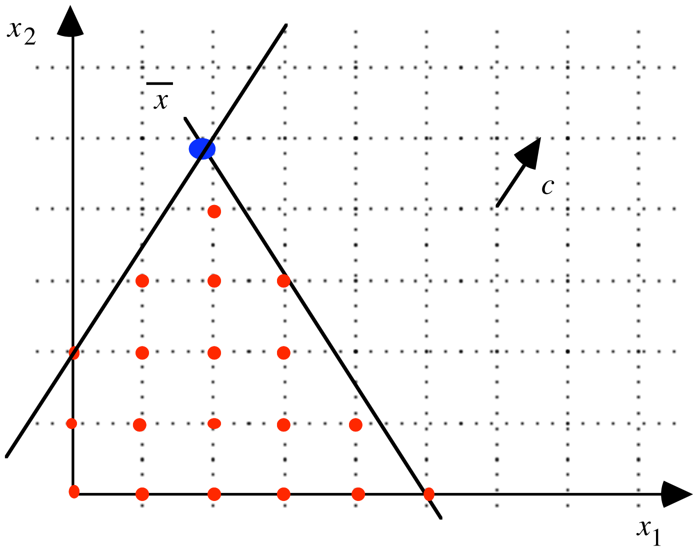

Fig. 11: feasible region and optimal solution of the continuous problem

In the figure we also indicated the solution $\bar{x}$ of the linear programming problem obtained eliminating integrity constraints from variables. Such problem can be easily solved by using the methods we saw in the preceding chapter, thus giving $\bar{x}=\begin{bmatrix}376/193\\950/193\end{bmatrix}$, with value 5.098. Since we have no integer components, such solution cannot be considered as the solution to the original ILP problem because it is unfeasible. We could try to obtain a feasible solution, possibly an optimal one, by approximating fractional values to integer values being closer to the values of the components of $\bar{x}$. In the examined case, if we look at the drawing we notice that such solution might be $x'=\begin{bmatrix}2\\4\end{bmatrix}$. Note that generating such point is not so easy in general, that is when the space of solutions has a dimension being much greater than two. Anyhow, the obtained solution is not only rather difficult to generate, but in some cases also proves to be very far from the optimal solution of the ILP problem. In the examined example the integer optimal solution is $x^*=\begin{bmatrix}5\\0\end{bmatrix}$, as it can be easily verified by comparing the values of the objective function.

Yet, we should ask ourselves what would have happened if, instead of using the above formulation with two constraints, we had used the following one, whose geometric representation appears in Figure 12:

$$\begin{array}{lll} \max & x_1 + 0.64\,x_2 & \\ & -x_1 + x_2 \le 2 & \\ & x_1 + x_2 \le 6 & \\ & 3x_1 + 2\,x_2 \le 15 & \\ & x_1, x_2 \ge 0 \text{ integers.} & \end{array}$$

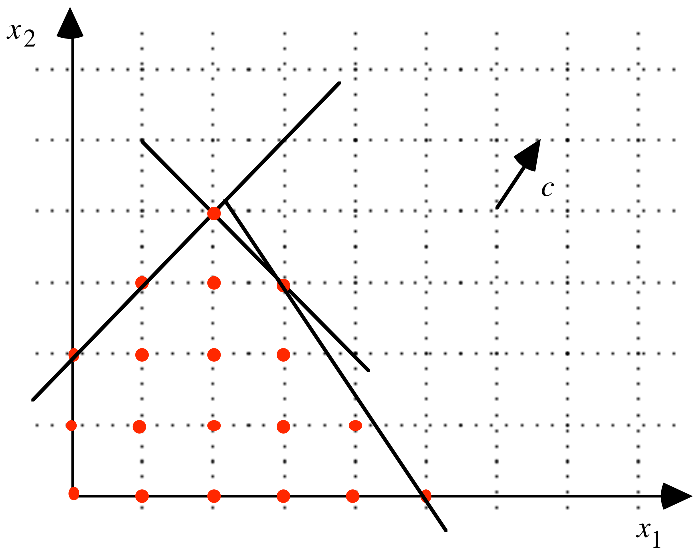

Fig. 12: geometric representation of the alternative formulation

We immediately notice that, from the point of view of the feasible region, the two formulations are equivalent; in fact they comprise the same set of integer points. However, the second formulation has all vertices corresponding with feasible solutions for the ILP problem. Remember that, if in a linear programming problem the finite optimum exists, then at least one vertex of the polyhedron is optimal; this implies that if we leave out the integrity constraints and solve the corresponding LP problem, the solution we obtain is integer, therefore it is optimal for the original problem as well.

The considerations we developed in the preceding example clearly illustrate how important it is to detect a good formulation of the problem. Let S be the set of feasible solutions of a specific discrete optimization problem; we can detect the smallest convex set comprising all points of S:

Definition 2.1
Given a finite set of points S, the *convex envelope of* S, denoted with $conv(S)$, is the smallest convex polyhedron containing all points of S.

Consider the problems:

$$P:\ \min\ cx \quad Ax = b \quad x \in \mathbf{Z}^n \qquad\qquad P':\ \min\ cx \quad x \in conv(\{Ax=b,\ x\in \mathbf{Z}^n\})$$

We rather intuitively understand that each vertex of the convex envelope is a point of S, hence the optimal solution of P', due to the linearity of the objective function, is optimal for P as well. Yet, we still do not know how to solve problem P, whereas P' can be solved by means of algorithms for linear programming. In fact, if there exists an optimal finite solution and at least one vertex of the polyhedron coincides with an optimal solution, then such vertex belongs to S. These observations might induce us to think that solving a discrete linear optimization problem or an ILP problem is easy: we obtain the formulation of the convex envelope in term of linear constraints and we apply the algorithm for solving LP problems that we know to be efficient. The difficulty, however, hides in the formulation of the convex envelope. In fact, in several problems, for example in the traveling salesperson problem, it is not possible to provide the complete formulation for instances of any size, whereas in general such formulation, even when it is possible to characterize it completely, requires an exponential number of constraints, thus making the method of solution extremely inefficient.

**2.1 Unimodularity*

There are cases in which the polyhedron $\bar{P}$:$\{Ax = b, x \geq 0\}$, from which we have removed the integrity constraints, coincides with the convex envelope $P'=conv(\{Ax=b, x\in \mathbf{Z}^n\})$. This happens when each vertex of $\bar{P}$, and therefore each of its bases, has integer components, i.e., it possesses the *integrality* property. It could be extremely helpful to characterize these problems, which reveal to be particularly easy to manage. Let us introduce some definitions that are useful to achieve this aim.

Definition 2.2
A matrix with integer components $m\times n$, A , is *unimodular* if for each non-singular square submatrix D being of maximum rank we get:

$$\det(D) = \pm 1.$$

Definition 2.3
A matrix with integer components $m\times n$, A, is *totally unimodular* (TU) if each of its non-singular square matrices is *unimodular*.

Theorem 2.4
If A and b have integer component and A is unimodular, then all basic solutions of the problem

$$\begin{aligned} \min\ & cx \\ & Ax = b \\ & x \geq 0 \end{aligned}$$

have integer components.

<u>Proof</u>
In fact, let us consider a basis B and let A_B^* be the adjoint matrix of A_B. We have

$$x_B=A_B^{-1}b = \frac{A_B^*}{\det(A_B)}b,$$

hence the basic solution has integer components. ♦

Corollary 2.5
If A and b have integer components and A is totally unimodular, then the problems:

$$\begin{aligned} \min\ & cx \\ & Ax \geq b \\ & x \geq 0 \end{aligned} \qquad\qquad \begin{aligned} \min\ & cx \\ & Ax \leq b \\ & x \geq 0 \end{aligned}$$

have all basic solutions with integer components.

<u>Proof</u>
The demonstration immediately follows by observing that if A is TU then also $[A,I\,]$, $[A,-I\,]$ are TU.♦

The characterization of totally unimodular has been the object of deeper study. Here we relate a sufficient unimodularity condition which does not only involve an elegant demonstration, but also has direct consequences on an extensive class of optimization problems, such as those on network flows.

Theorem 2.6
A matrix A with elements 0, ±1 is TU if the following conditions hold:
a) each column contains at most two non-null elements;
b) the indexes $\{1,\dots,m\}$ of the rows of A can be partitioned into two sets I_1, I_2 such that:

- if a column has two elements with the same sign, the indexes of corresponding rows do not belong to the same set;
- if a column has two elements with opposite signs, the indexes of corresponding rows belong to the same set.

Proof
Let C be a square submatrix of A of order h. The demonstration is performed by induction on index h.
If $h=1$, then according to definition we have $\det(C)=0, \pm 1$.
Let $1<h\leq n$. We have the following exhaustive cases:
i) either C contains at least one null column, hence $\det(C)=0$;
ii) or C contains at least one column with only one non-null element; in such case, by developing the determinant starting from one of these columns and by making use of the inductive hypothesis, we have $\det(C)=0, \pm 1$;
iii) or each column of C has exactly two elements different from zero; hence, for the j-th column it gives:

$$\sum_{i\in I_1} c_{ij} = \sum_{i\in I_2} c_{ij};$$

in fact, elements of the j-th column having opposite signs belong to the same set and therefore suppress each other in the sum, whereas equal elements belong to different sets. Consequently, there exists a linear combination of rows that provides the null vector:

$$\sum_{i\in I_1} C_i - \sum_{i\in I_2} C_i = 0.$$

Hence $\det(C)=0$.♦

As a consequence of the theorem 2.6, the incidence matrix of a directed graph is TU and also the incidence matrix of a bipartite undirected graph is TU. Therefore, flow problems, shortest-path problems, maximum-flow problems, bipartite assignment and transportation problems have integer basic solutions. Remember that we had informally examined this property when studying flow problems on a network and solution algorithms. Observe that, in general, the matrix of an undirected graph is not TU. In fact, an undirected graph is not bipartite if it contains at least one odd cycle. Let C be the incidence matrix of an odd cycle. It can be immediately verified that $\det(C) = \pm 2$.

2.2 "Polyhedral" or cutting-plane methods

Resuming the idea of referring to a LP case by formulating the problem in terms of convex envelope, it should be noted that it is not necessary to describe the polyhedron of the convex envelope completely, we can rather concentrate on the neighborhood of the optimal solution. *Polyhedral* or *cutting-plane* methods do nothing but iteratively refine the polyhedron of the convex envelope where we think the optimal solution is situated. Consider the following integer linear programming problem:

$$\begin{aligned} P:\ \min\ & cx \\ & Ax \leq b \\ & x\in \mathbf{Z}^n_+ \end{aligned}$$

as well as the problem in which we leave out the integrity constraints:

$$\begin{aligned}\overline{P}:\ \min\ & cx\\ & Ax \le b\\ & x \ge 0\end{aligned}$$

$\overline{P}$ is a linear programming problem that we suppose to admit finite optimal solution $\bar{x}$. If $\bar{x}$ has all integer components, then it coincides with the optimal solution of the integer problem P, otherwise we can try to refine the formulation of $\overline{P}$ by adding appropriate constraints.

Definition 2.7
A *valid inequality* is a constraint $gx \ge \gamma$, such that:

$$gx \le \gamma, \qquad \forall x \in \{x \in \mathbf{Z}^n_+ : Ax \le b\}$$

Definition 2.8
A *cutting plane* is a valid inequality $gx \le \gamma$, such that:

$$g\bar{x} > \gamma.$$

Therefore, if we detect a cutting plane for problem $\overline{P}$ and its fractional optimal solution $\bar{x}$, we can add such inequality to the formulation and iterate the procedure while progressively approaching the integer optimal solution. The process is summarized by the following procedure:

```
Procedure cutting_plane_method:
   begin
      optimal:= false;
      repeat
         Solve($\overline{P}$,$\bar{x}$);
         if $\bar{x}$ has all integer components then optimal:= true
            else begin
                  Generate_cut ($g$, $\gamma$);
                  Add_cut($\overline{P}$,$g$, $\gamma$)
               end
      until optimal
   end.
```

Fig. 13: cutting-plane method

Consequently, it is essential to find a method allowing to generate cutting planes that are as effective as possible, i.e., near to the polyhedron of the convex envelope. Now let us describe a method (that we owe to Chvátal) useful to derive valid inequalities which, however, are not necessarily cutting planes for the fractional solution.
Consider the integer programming problem

$$(P) \qquad \min\{cx : x \in F\},$$

where $F = \{x: Ax \le b,\ x \in \mathbf{Z}^n_+\}$, with A being a matrix $m \times n$ having non-negative integer components, b being a column vector having m integer components and d having non-negative integer components.

Theorem 2.9
Given a vector $u=[u_i] \ge 0$ $i=1,\ldots,m$, each inequality $gx \le \gamma$ with

$$g_j = \lfloor \sum_{i=1}^{m} u_i A_{ij} \rfloor,$$

$$(2.1) \qquad \gamma = \lfloor \sum_{i=1}^{m} u_i b_i \rfloor,$$

is a valid inequality for P.

Proof

We call $\bar{F} = \{x\text{: } Ax \le b,\ x \in \mathbf{R}^n_+\}$ and we consider the constraint:

$$(2.2) \qquad \sum_{j=1}^{n} \left(\sum_{i=1}^{m} u_i A_{ij}\right) x_j \le \sum_{i=1}^{m} u_i b_i \,.$$

Since it was obtained as a non-negative linear combination (with multipliers u_i) of the constraints $Ax \le b$, such constraint is satisfied by each $x \in \bar{F}$, therefore also by each $x \in F$.
Now let us consider the constraint

$$(2.3) \qquad \sum_{j=1}^{n} g_j x_j \le \sum_{i=1}^{m} u_i b_i \,.$$

Because $x \ge 0$, the examined constraint is less stringent in comparison with (2.2), since original coefficients were replaced with their lower integer part; hence it is satisfied by each $x \in F$. Because for $x \in F$ the left-hand side of (2.3) takes an integer value, the result is that the constraint

$$\sum_{j=1}^{n} g_j x_j \le \gamma$$

is satisfied by each $x \in F$, hence it is a valid inequality for P. ♦

Example: application of Chvátal inequalities
Consider the following problem:

$$\begin{array}{ll} \min & -x_1 - x_2 - x_3 \\ & x_1 + x_2 \le 1 \\ & x_2 + x_3 \le 1 \\ & +x_1 + x_3 \le 1 \\ & x_i \ge 0,\ x_i \in \mathbf{Z},\ i=1,\dots,3 \end{array}$$

This formulation corresponds to the problem of determining a maximum cardinality matching in a complete graph with cardinality 3. The optimal solution of the problem is obviously one and corresponds to the choice of any arc (x_1 or x_2 or x_3 equal to 1). Otherwise, if we solve the linear programming problem that leaves out the integrity constraints, we obtain 3/2 with each variable being set at value 1/2. If we take $u_1 = u_2 = u_3 = 1/2$ as multipliers, then the Chvátal inequality is:

$$\lfloor 1/2 + 1/2 \rfloor x_1 + \lfloor 1/2 + 1/2 \rfloor x_2 + \lfloor 1/2 + 1/2 \rfloor x_3 \le \lfloor 1/2 + 1/2 + 1/2 \rfloor$$

that is

$$x_1 + x_2 + x_3 \le 1.$$

Adding the inequality to the problem and leaving out the integrity constraints we obtain the integer optimal solution.

Obviously, the generation of Chvátal valid inequalities can be iterated; in other words, we can add the valid inequality to the set of constraints and from the latter we can form a new valid inequality and so forth.
We will call this procedure **Generate_Inequality**. The fundamental achievement brought about by Chvátal is that we need a finite number of calls of **Generate_Inequality** to obtain the inequalities necessary to define $conv(F)$. For details about this important result, see []

Unfortunately, the above mentioned result does not provide constructive tools, that is it gives no information about how to determine a valid inequality that proves to be a cut as well. In the early

Seventies, Gomory proposed a general method, apparently infallible, of generating cuts for any integer linear programming problem starting from the fractional optimal solution of the problem in which integrity constraints were left out.
Consider the problem P

$$\begin{aligned} \min \quad & cx \\ & Ax = b \\ & x \geq 0 \\ & x \in \mathbf{Z}^n \end{aligned}$$

and the problem $\overline{P}$ in which we leave out the integrity constraints

$$\begin{aligned} \min \quad & cx \\ & Ax = b \\ & x \geq 0. \end{aligned}$$

Let $\overline{x} = \begin{bmatrix} \overline{x}_B \\ \overline{x}_N \end{bmatrix}$ be an optimal basic solution of $\overline{P}$, hence $\overline{x}_B, = A_B^{-1}b$ and $\overline{x}_N = 0$. Writing in full the constraints of problem $\overline{P}$ and pointing out the components in the base and those outside the base, we have:

$$A_B x_B + A_N x_N = b$$

therefore, if we derive the expression of x_B in full we have:

$$(2.4) \qquad x_B = A_B^{-1}b - A_B^{-1}A_N\, x_N\,.$$

We call $\overline{b} = A_B^{-1}b$ and $\overline{A} = A_B^{-1}A_N$. If $\overline{x}_B$ has all integer components, then solution $\overline{x}$ is optimal also for P. Otherwise, it will exist a component h such that $\overline{x}_h$ has a non-integer value; let t be the constraint corresponding to component h in the system (2.4), i.e.:

$$(2.5) \qquad x_h + \sum_{j \in N} \overline{a}_{tj} x_j = \overline{b}_t = \overline{x}_h.$$

Observe that (2.5) is a linear combination of the problem constraints with multipliers given by the t-th row of matrix A_B^{-1}, therefore it is a valid constraint for the problem. Consider now the cutting off of coefficients on the left: we can generate the following valid inequality:

$$(2.6) \qquad x_h + \sum_{j \in N} \lfloor \overline{a}_{tj} \rfloor x_j \leq \overline{b}_t$$

Since we are dealing with solutions having integer components, we can state that the left-hand side of (2.6) must be integer, hence we can reinforce the inequality in the following way:

$$(2.7) \qquad x_h + \sum_{j \in N} \lfloor \overline{a}_{tj} \rfloor x_j \leq \lfloor \overline{b}_t \rfloor.$$

Now we just have to check that (2.7) is a cut, i.e., that it is not satisfied by $\overline{x}$. Given that $\overline{x}_N = 0$, (2.7) is reduced to:

$$\overline{x}_h \leq \lfloor \overline{b}_t \rfloor$$

which is obviously violated since $\overline{x}_h = \overline{b}_t$.

<u>Example: application of Gomory cuts (1)</u>
Consider the problem:

$$\begin{array}{ll} \min & -x_1 - x_2 \\ & -3x_1 + 12x_2 + x_3 = 30 \\ & 6x_1 - 3x_2 + \quad x_4 = 8 \\ & x_i \in \mathbf{Z}_+ \end{array}$$

The optimal solution of the continuous problem is $\bar{x}_B = \begin{bmatrix} 62/21 \\ 68/21 \end{bmatrix}$ where $B = \{1,2\}$ and $A_B^{-1} = \begin{bmatrix} 1/21 & 4/21 \\ 2/21 & 1/21 \end{bmatrix}$. By applying (2.5) to the two fractional components of the solution we have:

$$x_1 + \frac{1}{21}x_3 + \frac{4}{21}x_4 = \frac{62}{21}$$
$$x_2 + \frac{2}{21}x_3 + \frac{1}{21}x_4 = \frac{68}{21}$$

therefore, Gomory cuts become:

$$x_1 \le 2, \qquad x_2 \le 3.$$

Exercise
Considering x_3 and x_4 as slack variables, graphically represent the problem and Gomory's cuts which were derived from the example.

<u>Example: application of Gomory cuts (2)</u>
Given the ILP problem

$$\begin{array}{ll} \max & -0.1x_1 + x_2 \\ & -4x_1 + 4x_2 \le 30 \\ & x_1 + 2x_2 \le 8 \\ & x_i \in \mathbf{Z}_+ \ i=1,2 \end{array}$$

The optimal solution can be graphically determined and lies in the intersection of the two constraints: x_1=3/2 and x_2=11/4. In order to approach the problem with the method of Gomory cuts, we introduce slack variables so as to turn constraints into inequalities. The problem, transformed in this way and with relaxed integrity constraints, is the following:

$$\begin{array}{ll} \max & -0.1x_1 + x_2 \\ & -4x_1 + 4x_2 + x_3 = 30 \\ & x_1 + 2x_2 + x_4 = 8 \\ & x_i \ge 0 \ i=1,,\dots,4 \end{array}$$

The base B is {1,2} whereas N={3,4}. The different matrices that are involved in the computations are:

$$A_B = \begin{bmatrix} -4 & 4 \\ 1 & 2 \end{bmatrix}, \qquad A_N = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$A_B^{-1} = A_B^{-1}A_N = \begin{bmatrix} -1/6 & 1/3 \\ 1/12 & 1/3 \end{bmatrix}$$

The Chvátal-Gomory cut referring to the first fractional variable is:
$$x_1 + \lfloor -1/6 \rfloor x_3 + \lfloor 1/3 \rfloor x_4 \le \lfloor 3/2 \rfloor$$
hence
$$x_1 - x_3 \le 1$$
replacing x_3 by the expression derived from the first equation of the constraint system:
$$x_3 = 5 + 4x_1 - 4x_2$$
we get
$$-3x_1 + 4x_2 \le 6.$$

The Chvátal-Gomory cut referring to the second fractional variable is:

$$x_2 + \lfloor 1/12 \rfloor x_3 + \lfloor 1/3 \rfloor x_4 \leq \lfloor 11/4 \rfloor$$

hence

$$x_2 \leq 2.$$

Exercise

Graphically represent the problem, the fractional solution and the obtained cuts. Derive the new solution after the introduction of the cuts.

The techniques illustrated above are wholly general and can be applied to any integer linear programming problem. There are also methods examining the characteristics of the polyhedra of the different problems more closely and generating *ad hoc* inequalities for specific cases. Let us take the example of a knapsack problem:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} c_i x_i \\ & \sum_{i=1}^{n} a_i x_i \leq b \\ & x \in \{0,1\}^n \end{aligned}$$

Detecting a subset of indexes C for which

$$\sum_{i \in C} a_i > b$$

we can state that all objects in C cannot be present at the same time in a feasible solution, hence the inequality

$$\sum_{i \in C} x_i \leq |C| - 1$$

is valid. In a similar and at times quite sophisticated way, it is possible to obtain valid inequalities for many other problems, such as the traveling salesperson problem, the vehicle routing problem, graph coloring, etc.

3. Implicit enumeration methods

Instead of using polyhedral methods, discrete optimization problems can be approached using algorithms of the enumerative type. The concept on which this class of algorithms is based is rather simple: we have to systematically enumerate all feasible solutions of the problem, evaluate the objective function and choose the best one. Just to give an example, let us consider the following instance of the knapsack problem:

$$\begin{aligned} \max \quad & 4x_1 + x_2 + 3x_3 + x_4 \\ & 5x_1 + 4x_2 + 3x_3 + x_4 \leq 8 \\ & \quad x_i \in \{0,1\},\ i=1,\dots,4. \end{aligned}$$

The set of solutions of this problem can be *enumerated* using the *decision tree* shown in Figure 14. Let us consider the problem's variables (objects to be put into the knapsack) one by one, and let us "open" a new ramification of the decision tree for each possible value that can be assigned to the examined variable (object that are put into the knapsack or that are discarded). Each node of the decision tree corresponds therefore to a subproblem in which some variables have values that are fixed in advance and can be established according to the route connecting the node to the root of the tree. For example, node D corresponds to the subproblem in which we set $x_1 = 0$ e $x_2 = 1$:

$$\begin{aligned} 1 + \max \quad & 3x_3 + x_4 \\ & 3x_3 + x_4 \leq 8 - 4 \\ & x_i \in \{0,1\},\ i=3, 4. \end{aligned}$$

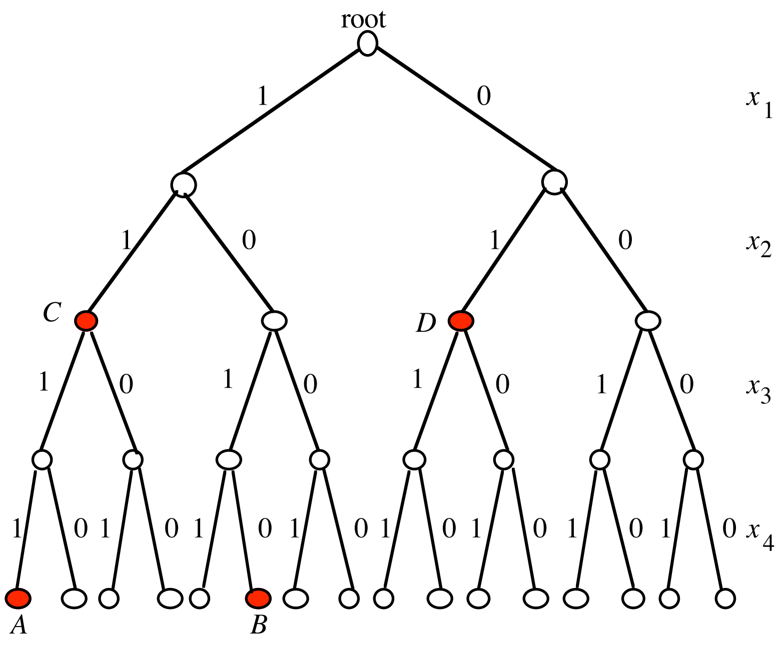

Fig. 14: decision tree for a knapsack problem 0-1 with 4 objects

Nodes of tree level *i* are 2^i because in the examined case the values we assign to each variable are 2. Paths from the root to the leaves detect the attributions of values {0,1} to variables and therefore also some solutions (not necessarily feasible) for the considered problem. For example, the path from root to leaf *A* detects an unfeasible solution of the considered knapsack problem (the global weight amounts to 13 and the capacity is 8), whereas the path from root to leaf *B* detects a feasible solution having value 7. Therefore, the aim is to detect among all the leaves corresponding to feasible solutions the one having the best value.
Observe how the nodes of the decision tree increase exponentially with the problem size, thus an algorithm actually proceeding to the explicit enumeration of all nodes reveals to be not much efficient. Therefore, we have to use "tricks" allowing to remove useless portions of the tree, thus reducing the number of nodes to evaluate.
A starting criterion should enable us to recognize as soon as possible when a subtree does not contain any feasible solution. In the case of Figure 14, once arrived at node *C* we realize we have already exceeded the knapsack's capacity and it is no use proceeding to the enumeration of the subtree having root *C*. In this way, entire subtrees can be discarded and considered as implicitly visited: then we speak of *implicit enumeration algorithms*. If we proceed to the reduction according to this criterion, what is left is the tree shown in Figure 15.

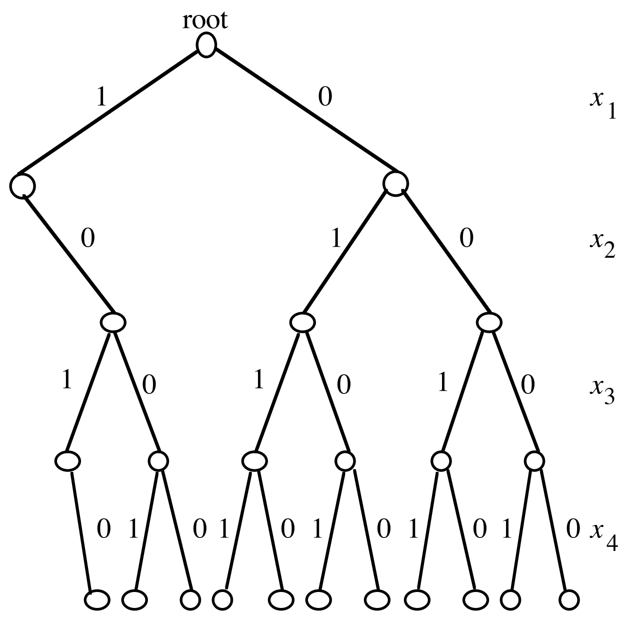

Fig. 15: decision tree that has been reduced according to the feasibility criterion

A second criterion allowing to reduce the extent of the enumeration is related to the value of solutions. Suppose we have an available feasible solution with value v. For example, in the knapsack case we could find a feasible solution by using the algorithm we saw in Chapter 1 (burglar problem): $x_3=x_4=1$, $x_1=0$, $x_2=1$, $v=5$. If, when developing the decision tree, we realize that a subtree does not contain solutions having value greater than 5, then it is no use to proceed to the enumeration of the subtree, even if it contains feasible solutions. The most critical point is how to estimate the best solution value of a subtree without making use of the enumeration. In the examined example a very rough overestimation of the best solution value could be the one that chooses the objects to be put into the knapsack while completely leaving out the capacity constraint. If the overestimation computed at a tree node is smaller than or equal to the value of the available solution, then we can avoid continuing the enumeration of the subtree. According to this criterion, the enumeration tree is further reduced, as shown in Figure 16. Beside each node we indicate the estimation value. Note that on the leaves the estimation coincides with the solution value.

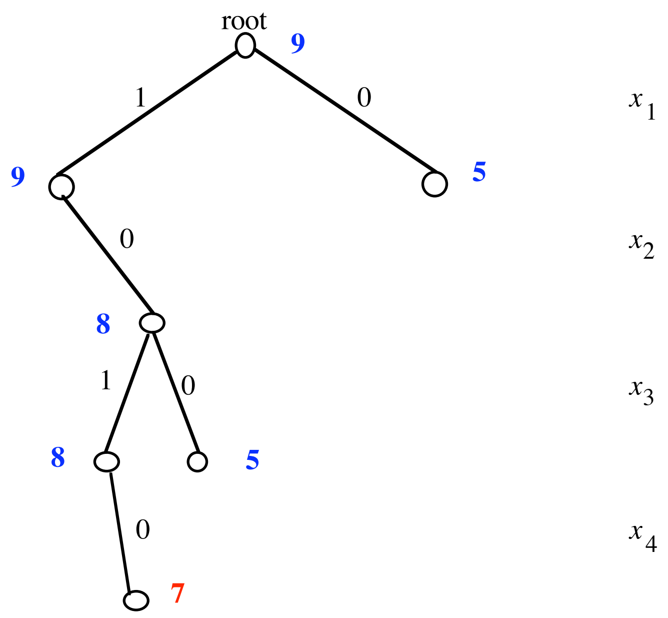

Fig. 16: decision tree reduced according to the criterion of overestimation of the optimal solution

The example developed above provides the main ideas of an implicit enumeration algorithm, which is known as *Branch&Bound* and can be summarized in the few following code lines: the procedure takes as input a problem P in maximization form as well as the value of a feasible solution v.

```
Procedure Branch_and_Bound(P,v);
   begin
      b=Bound(P,x,unfeasible);
      if x is integer then return (b);
      else if unfeasible return(-∞);
        else if b>v then
        begin
          Branch(P,x,P_1,P_2,…,P_k)
          for i=1,…,k do begin
            t:= Branch_and_Bound(P_i,v)
            if t > v then v:= t;
          end
      end
   end.
```

Fig. 17: Branch&Bound procedure

Inside the algorithm we make use of two procedures. The function **Bound**(P,x,unfeasible) provides a higher estimation of the optimal solution value of subproblem P (*upper bound*), returning, in addition to the estimation value, a solution x and a logical value having value **true** if subproblem P admits no solutions. If unfeasible=true, it is no use proceeding to the enumeration of the subtree corresponding to

subproblem P because it contains no solutions. Otherwise, if **Bound**(P,x,unfeasible) returns a solution x having integer components, therefore being feasible, then it is equally useless to proceed to the enumeration of the subtree corresponding to subproblem P, because we already have at our disposal the value of its optimal solution. The other procedure being used decomposes the problem P into the subproblems $P_1, \ldots, P_k$. performing the operation of "branching" the enumeration tree. The procedure **Branch** takes as input the solution x provided by the "bound", since in general it provides valuable indications of how P should be efficiently partitioned.
The basic ingredients for a Branch&Bound algorithm are now evident: 1) the estimation technique of the "bound" which must be computationally efficient and as accurate as possible in order to reduce the extent of the tree, 2) the problem decomposition, 3) the criterion by which the enumeration tree is visited.

3.1 Estimation of the optimal solution: relaxations

Let us consider the optimization problem P : $\max\{c(x): x \in F\}$.

Definition 3.1
We call P': $\max \{c'(x): x \in F'\}$ a *relaxation* of P if

$$F \subseteq F',$$
$$x \in F \;\Rightarrow\; c'(x) \geq c(x).$$

According to the definition of relaxation, we can easily verify the following results:

Theorem 3.2
The value of the relaxation of a maximization problem represents an overestimation of the optimal solution value:

$$z(P) = \max\{c(x): x \in F\} \leq z(P') = \min\{c'(x): x \in F'\}.$$

Theorem 3.3
Let $x^* \in F'$ be an optimal solution for the relaxation P', i.e., $z(P') = c'(x^*)$, further let $x^* \in F$ and $c(x^*)=c'(x^*)$, then x^* is optimal solution for P too.

In defining a relaxation of a problem we generally take into account the fact that the optimal solution of a relaxation should be simpler to compute than the optimal solution of the original problem. Later in this chapter we will consider just a few of the main relaxation techniques, i.e., the *constraint removal* and the *continuous relaxation.*

Consider a problem P whose feasible region is given by the intersection of two sets of constraints:

$$(P) \qquad \min\{f(x) : x \in X_1 \cap X_2\}.$$

A relaxation of P is the problem

$$(PR) \qquad \min\{f(x) : x \in X_1\},$$

which is obtained from P by removing the constraint that x must belong to the set X_2.
A relaxation of this type is meaningful when the constraints defining the set $X_1 \cap X_2$ are difficult to deal with, whereas those merely defining the set X_1 are tractable.

<u>Example: relaxation of the traveling salesperson problem</u>
Consider the traveling salesperson problem on digraph we discussed in the examples of paragraph 1. A relaxation which is common to the two formulations TSP1 and TSP2 is the assignment problem:

$$\text{TSPR:} \; \min\{ \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}x_{ij} : x \in X_1\}.$$

Solutions of this relaxation can be not only Hamiltonian cycles, but also sets of disjoint cycles containing all the graph nodes (cycle cover of the graph nodes).

The continuous relaxation may be defined as a particular case of constraint removal. In fact, in the continuous relaxation of an integer linear programming problem the integrity constraints of variables are removed. It is given the combinatorial optimization problem:

$$(P) \qquad \min\{f(x): x \in X, x \in \{0,1\}^n\},$$

with X being the convex subset of $\boldsymbol{R}^n$.
We call *continuous relaxation* of P the following problem

$$(Pc) \qquad \min\{f(x): x \in X, x \in [0,1]^n\}.$$

If P is an integer programming problem

$$(P) \qquad \min\{f(x): x \in X,\ x \in \boldsymbol{Z}^n\},$$

then the continuous relaxation of P is given by:

$$(Pc) \qquad \min\{f(x): x \in X\}.$$

In both cases, if $f(x)$ is linear and X is a polyhedron, Pc is a linear programming problem.

Example: location of continuous relaxation plants
Let us examine a plant location problem similar to the one we discussed in Chapter 1. It is given a set of candidate sites S in which service centers can be installed (in Chapter 1 we had to do with cellular antennas). For each candidate site $j \in S$ we know an activation cost $f_j > 0$ and a maximum capacity u_j. It is also given a set of clients C to be assigned to the activated centers. For each possible client i - center j assignment we know a utility coefficient $c_{ij} > 0$. The problem consists in activating a subset of centers and in assigning them the clients so as to respect capacity constraints and minimize the difference between cost and utility. A possible formulation uses 0-1 activation variables y_j and assignment variables x_{ij}:

$$\begin{array}{lll}
\min & \sum_{j\in S} f_j y_j - \sum_{i\in C}\sum_{j\in S} c_{ij}x_{ij} & \\
& \sum_{j\in S} x_{ij} = 1 & \forall i\in C \\
& \sum_{i\in C} x_{ij} \le u_j y_j & \forall j\in S \\
& x_{ij}, y_j \in \{0,1\} & \forall i\in C, \forall j\in S.
\end{array}$$

The continuous relaxation of the location problem is obtained by replacing the constraints according to which the variables x and y belong to the set $\{0,1\}$, with the constraints:

$$0 \le x_{ij} \le 1,\ 0 \le y_j \le 1 \qquad \forall i\in C, \forall j\in S.$$

It clearly appears that this results in a linear programming problem that we can efficiently solve by using the simplex algorithm. Yet, we can push the analysis further and observe how the continuous relaxation can actually be related to a minimum cost flow problem. The graph is defined by a set of nodes N corresponding to clients in C, to centers in S and to a *sink* node t. At each client node there is a flow supply of one unit, sink t demands $|C|$ flow units, whereas center nodes are transshipment nodes. We have arcs connecting all client nodes to all center nodes having capacity 1, and all arcs connecting each center node j to t having capacity u_j. Arc costs are equal to $-c_{ij}$ for arcs from client i to center j, and are equal to f_j/u_j for arcs from center j to t. Figure 18 shows a representation of the network.

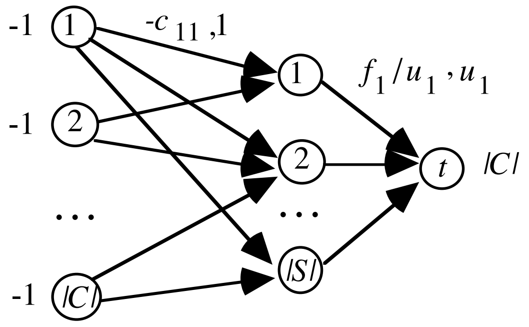

Fig. 18: formulation of the continuous relaxation of the location problem

The problem can be quite efficiently solved using a minimum cost flow algorithm. Variables x_{ij} correspond to flows on arcs (i,j) whereas y_j are obtained by dividing the flow on arc (j,t) by u_j.

***Exercise**

For the location problem we can make use of another relaxation that originates a series of problems we already know. Consider the removal of the assignment constraints ($\sum_{j\in S} x_{ij} = 1$, $\forall i\in C$). Analyze the resulting problems and provide a solution algorithm.

Observe how different but equivalent models can generate different continuous relaxations being not all equally effective.
For example, consider the following logical constraint:

$y = 0 \Rightarrow x_1 = 0$ and $x_2 = 0$, with y, x_1 and x_2 being binary variables.

Such constraint can be expressed equally well by any of the following models:

M_1: $x_1 + x_2 \leq 2y$, $(x_1, x_2, y)\in\{0,1\}^3$;

M_2: $x_1 \leq y$, $x_2 \leq y$, $(x_1, x_2, y)\in\{0,1\}^3$.

The continuous relaxations are the following, respectively:

M'_1: $x_1 + x_2 \leq 2y$, $(x_1, x_2, y)\in[0,1]^3$;

M'_2: $x_1 \leq y$, $x_2 \leq y$, $(x_1, x_2, y)\in[0,1]^3$.

it can be easily verified that $M'_1 \supset M'_2$; in fact, it clearly appears that each point of M'_2 belongs to M'_1 as well, but not the contrary. For example, the points $(1, 0, \frac{1}{2})$ and $(0, 1, \frac{1}{2})$ belong to M'_1 but not to M'_2.
It follows that, in general, the second relaxation is preferable to the first, even if it needs one more constraint in case we want relaxations that are as *near* as possible to the original problem.

Example: efficient computation of the continuous relaxation for the knapsack problem
Consider a knapsack problem with n objects:

$$
\begin{aligned}
(P)\text{: } \max & \sum_{i=1}^{n} c_i x_i \\
& \sum_{i=1}^{n} a_i x_i \leq b \\
& x_i \in \{0,1\} \qquad i=1,\ldots,n.
\end{aligned}
$$

The continuous relaxation $\overline{P}$ of P is obtained by transforming the constraints $x_i \in \{0,1\}, i=1,\ldots,n$ into $0\leq x_i \leq 1$, $i=1,\ldots,n$.

$$(\bar{P}): \max \sum_{i=1}^{n} c_i x_i$$
$$\sum_{i=1}^{n} a_i x_i \leq b$$
$$0 \leq x_i \leq 1, \qquad i=1,\ldots,n.$$

Supposing, for more simplicity, that indexes of objects are already ranked in non-ascending order, with ratio (c_i/a_i), then the optimal solution of $\bar{P}$ can be computed according to the following rule: let k be the index such that:

$$\sum_{i=1}^{k-1} a_i < b$$
$$\sum_{i=1}^{k} a_i \geq b$$

then

$$\bar{x}_i = \begin{cases} 1, & i=1,\ldots,k-1 \\ \dfrac{b - \sum_{j=1}^{k-1} a_j}{a_k}, & i=k, \\ 0 & i=k+1,\ldots,n. \end{cases}$$

Note that the solution, computed in this way, has at most one variable with fractional value $\bar{x}_k$, whereas all the others have value 0 or 1. We can prove that the solution, determined in this way, is actually optimal by using the complementary slackness theorem. The dual of the problem $(\bar{P})$ is:

$$(\bar{D}): \min \lambda b + \sum_{i=1}^{n} \mu_i$$
$$\lambda a_i + \mu_i \geq c_i \qquad i=1,\ldots,n$$
$$\lambda, \mu_i \geq 0, \qquad i=1,\ldots,n.$$

where the variable λ is associated with the capacity constraint in $(\bar{P})$ and variables μ_i are associated with the constraints $(x_i \leq 1)$. The complementary slackness equations are:

$$(\lambda a_i + \mu_i - c_i)\, x_i = 0 \qquad i=1,\ldots,n$$
$$\lambda\, (b - \sum_{i=1}^{n} a_i x_i) = 0$$
$$\mu_i\, (1 - x_i) = 0 \qquad i=1,\ldots,n.$$

Considering how the solution $\bar{x}$ was constructed, if we apply the complementary slackness equations we obtain the following dual solution:

$$\bar{\mu}_i = 0 \qquad i = k,\ldots,n$$
$$\bar{\lambda} = \frac{c_k}{a_k}$$
$$\bar{\mu}_i = c_i - \frac{c_k}{a_k} a_i \qquad i = 1,\ldots,k-1.$$

Exercise

Prove that the solution $(\bar{\lambda},\bar{\mu})$ is dual feasible.

Let us try to apply the construction of the optimal solution of the continuous relaxation to the following problem:

$$
\begin{aligned}
P\text{: max } & 52x_1 + 40x_2 + 38x_3 + 9x_4 + x_5 \\
& 17x_1 + 5x_2 + 13x_3 + 3x_4 + x_5 \quad \le 20 \\
& x_i \in \{0,1\},\ i=1,\dots,5.
\end{aligned}
$$

The index ordering according to the ratio c_i/a_i is: 2, 1, 4, 3, 5. Hence, the solution of the continuous relaxation is: $\bar{x}_2 = 1, \bar{x}_1 = 15/17, \bar{x}_4 = \bar{x}_3 = \bar{x}_5 = 0$, with value 85.88.

3.2 Branching rules

In an enumeration algorithm the *branching rule* allows to partition the feasible region into subsets, each defining a subproblem having a reduced size in comparison with the original problem. Formally, if we are confronted with a problem of the type

$$P\text{: max } \{cx\text{: } x \in F\},$$

a branching rule defines a partition of F into a family of subsets $\{F_1, F_2, \dots, F_k\}$, in which the union of the subsets obviously yields the original set F and, if possible, their two-by-two intersection is empty. The subproblems on which the exploration is continued are:

$$P_i\text{: max } \{cx\text{: } x \in F_i\} \qquad i=1, \dots, k.$$

When k=2 we speak of bipartite branching rule.

Another characterization of a branching rule is the strategy determining the partition choice. In the example illustrating the basic principles of implicit enumeration algorithms we adopted a bipartite branching rule. In the same example we fixed an ordering of variables that were considered in sequence at the different levels of the enumeration tree. At each node of the tree, the variable relative to the level was set (at 0 and at 1) and the tree was branched. This extremely static enumeration strategy can be improved taking into account the characteristics of the problem to be solved and the solution of the continuous relaxation. First of all, it is not necessary that the branching is performed on the same variable at nodes being at the same level of the tree.

The choice of the variable on which to perform the branching is usually related to the result of the problem relaxation. Let us consider an integer linear programming problem with binary variables

$$
\begin{aligned}
\max\ & cx \\
& Ax \le b \\
& x \in \{0,1\}^n
\end{aligned}
$$

and let $\bar{x}$ be the optimal solution of its continuous relaxation that we suppose to be unfeasible for the original problem. Since we have to branch the enumeration tree, we are free to choose any variable. If we choose a variable x_i for which $\bar{x}_i$ is equal to 0, we have two subproblems $P_0 = \{\max cx\text{: } Ax \le b, x \in \{0,1\}^n, x_i = 0\}$ and $P_1 = \{\max cx\text{: } Ax \le b, x \in \{0,1\}^n, x_i = 1\}$. Note that the optimal solution of the continuous relaxation of P_0 is equal to $\bar{x}$, because setting the variable x_i =0 had no influence on the solution obtained previously. A similar reasoning can be made for problem P_1 when we set at 1 a variable for which $\bar{x}_i$ is equal to 1. Consequently, in such cases we would be able to avoid the computation of the continuous relaxation in one of the two branches of the tree, but we also risk to "seize" little information from the solution of the continuous relaxation. The commonly adopted idea is to choose a variable x_i for which $\bar{x}_i$ takes fractional values. This implies that at both branches of the enumeration tree we have a change of the continuous relaxation. When it is possible to choose among more than one fractional variables, we usually give priority to the "more fractional" one, that is (in the case of problems with binary variables) the one whose value in the continuous relaxation is closer to 0.5.

Example: branching for the binary knapsack problem

Consider the binary knapsack problem of which we previously computed the continuous relaxation. The variable on which to perform the branching is the only one with fractional value, hence x_1. We obtain the two subproblems:

$$\begin{array}{lll} P_0: \max & 40x_2 + 38x_3 + 9x_4 + x_5 & \\ & 5x_2 + 13x_3 + 3x_4 + x_5 & \leq 20 \\ & x_i \in \{0,1\}, \quad i=2,\ldots,5. & \end{array}$$

$$\begin{array}{lll} P_1: 52 + \max & 40x_2 + 38x_3 + 9x_4 + x_5 & \\ & 5x_2 + 13x_3 + 3x_4 + x_5 & \leq 3 \\ & x_i \in \{0,1\}, \quad i=2,\ldots,5 & \end{array}$$

whose continuous relaxations are respectively: $\bar{x}_2 = 1, \bar{x}_4 = 1, \bar{x}_3 = 12/13, \bar{x}_5 = 0$, with value 84.08 and $\bar{x}_2 = 3/5, \bar{x}_4 = \bar{x}_3 = \bar{x}_5 = 0$ with value 76. Note that, wishing to go on performing the branching, in P_0 the variable of reference is x_3, whereas in P_1 it is x_2, and produces the tree represented in Figure 19.

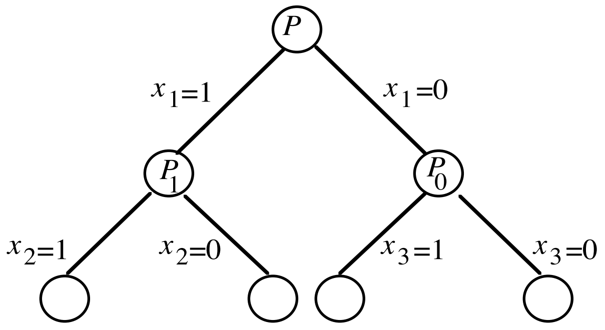

Fig. 19: branching up to the second level of the enumeration tree

In the more general case of problems with integer variables

$$\begin{array}{ll} P: \max & cx \\ & Ax \leq b \\ & x \in \mathbf{Z}^n \end{array}$$

we can hold on to the binary branching rule. Consider a variable x_i for which the solution of the continuous relaxation $\bar{x}_i$ takes a fractional value. The decomposition into two subproblems can be the following:

$$P_L = \{\max cx: Ax \leq b, x \in \mathbf{Z}^n, x_i \leq \lfloor \bar{x}_i \rfloor\}$$
$$P_U = \{\max cx: Ax \leq b, x \in \mathbf{Z}^n, x_i \geq \lceil \bar{x}_i \rceil\}.$$

Note that in this case we set no variables but just introduced constraints. The introduction of constraints allowed to remove all values of the variable x_i comprised between $\lfloor \bar{x}_i \rfloor$ and $\lceil \bar{x}_i \rceil$.

Example: bipartite branching for the ILP general case

Let us consider the following integer linear programming problem:

$$\begin{array}{lll} P: \max & -x_1 + x_2 & \\ & x_1 + x_2 & \leq 5 \\ & -3x_1 + 2x_2 & \leq 1 \\ & 2x_1 & \leq 7 \\ & x_i \in \mathbf{Z}_+, i=1, 2. & \end{array}$$

Figure 20 shows the geometric representation of P and the solution of its linear relaxation, which is given by $\bar{x}_1 = 9/5, \bar{x}_2 = 16/5$.

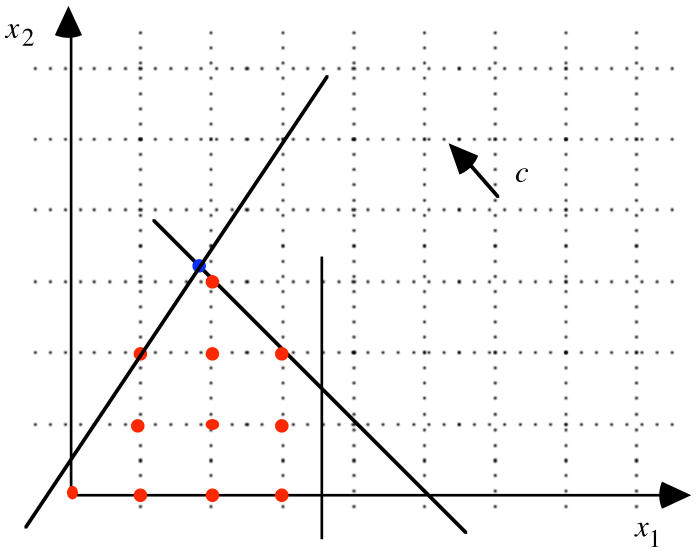

Fig. 20: geometric representation and solution of the continuous relaxation

If we choose to perform the variable x_1 we obtain the two subproblems:

$$
\begin{array}{lll}
P_L: \max -x_1 + x_2 & & \\
\quad x_1 + x_2 & \leq & 5 \\
\quad -3x_1 + 2x_2 & \leq & 1 \\
\quad 2x_1 & \leq & 1 \\
\quad x_i \in \mathbf{Z}_+,\ i=1, 2. & &
\end{array}
$$

$$
\begin{array}{lll}
P_U: \max -x_1 + x_2 & & \\
\quad x_1 + x_2 & \leq & 5 \\
\quad -3x_1 + 2x_2 & \leq & 1 \\
\quad 2x_1 & \leq & 7 \\
\quad 2x_1 & \geq & 2 \\
\quad x_i \in \mathbf{Z}_+,\ i=1, 2. & &
\end{array}
$$

In Figure 21 the feasible regions of the two subproblems are pointed out.

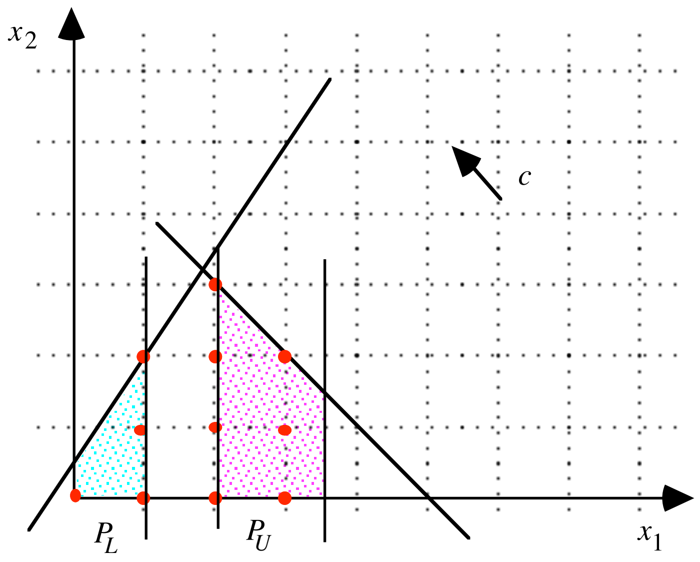

Fig. 21: geometric representation and solution of the continuous relaxation

Exercise
Develop the enumeration tree for problem P evaluating the solution of continuous relaxations and making use of the geometric representation shown in Figure 21.

When developing an enumeration tree for some problems, it might be interesting not to hold on to a bipartite branching rule. For example, in an ILP problem of the type

$$\begin{aligned} P:\ \max\ & cx \\ & Ax \leq b \\ & x \in \{1,\dots,k\}^n \end{aligned}$$

(i.e., a problem in which variables take integer values in a bounded set), once we detect the variable on which to perform the branching (x_i) it might be helpful to carry out the following decomposition:

$$P_1 = \{\max cx\text{: } Ax \leq b,\ x \in \{1,\dots,k\}^n,\ x_i = 1\}$$
$$\dots$$
$$P_k = \{\max cx\text{: } Ax \leq b,\ x \in \{1,\dots,k\}^n,\ x_i = k\}\ ,$$

that is to set the value of the branching variable in turn at all possible values situated in its feasibility set. In this case we speak of *k-partite branching*.
A k-partite branching rule can also be adopted in certain cases of problems with binary variables, as shown by the following example.

<u>Example: branching for the traveling salesperson problem</u>
Let G=(N,A) be an undirected complete graph with weights w_{ij}≥0 associated with arcs; the purpose is to determine the Hamiltonian cycle with least total weight. In Chapter 2 we saw how this problem can be formulated using binary decision variables associated with arcs. So we might be induced to form an enumeration tree on the basis of a binary branching rule. Yet, it should be noted that the variables are $\frac{n(n-1)}{2}$, hence the leaves of the complete enumeration tree will be $2^{n(n-1)/2}$, whereas the number of feasible solutions, i.e., the number of Hamiltonian cycles of a complete graph, is (n-1), that is a number being much smaller than the leaves of the tree. This means that a very large amount of leaves will correspond to unfeasible solutions, and taking into account their removal might be a heavy task.
An alternative branching rule can be of the k-partite type. Let us consider the root node of the enumeration tree and let us suppose we are at node 1 of the graph and we have to start the search. We have (n-1) alternatives, that is as many as the nodes still to be visited. So we can open (n-1) branches of the tree, one for each node following the first. At the following levels of the tree the choices are reduced (one less for each level). until we come to level (n-2) where of course there is no choice. There is a correspondence 1:1 between the leaves of the tree and the permutations of n nodes starting with node 1, thus with the feasible solutions of the traveling salesperson problem.

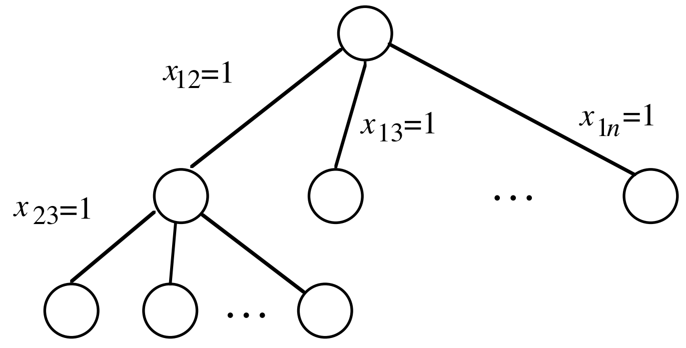

Fig. 22: a *k*-partite branching rule for the traveling salesperson problem

Note that when we choose the node following node *i* in the cycle (let it be *j*), we actually set the variable x_{ij} at 1. While making this choice, we implicitly set all variables x_{ih} at 0, because we cannot have any other nodes following *i* in addition to the one we chose.
The branching strategy we have just proposed is static and in no way related to the information drawn from the computation of the problem relaxation. Let us now propose a second *k*-partite branching rule based on the information drawn from the relaxation in which subcycle removal constraints are left out (*TSPR*). If the solution of the relaxation is unfeasible, this means that there exists a cycle not containing all the graph nodes. For more simplicity, let us suppose that the solution of the relaxation induces a cycle having three nodes $\{i - j - h\}$. A branching rule taking into account this piece of information partitions the set of solutions by trying to "split" such cycle, thus preventing that the arcs composing the subcycle are simultaneously present in the solution. The subproblems that are generated are:

P_1: $\{\min cx: x\in X, x_{ij}=0\}$
P_2: $\{\min cx: x\in X, x_{ij}=1, x_{jh}=0,\}$
P_3: $\{\min cx: x\in X, x_{ij}=1, x_{jh}=1, x_{hi}=0\}$.

Exercise
Prove that the branching rule proposed for the traveling salesperson problem partitions the feasible region correctly, without excluding any Hamiltonian cycle.

3.3 Enumeration tree search strategies

A further element characterizing a Branch and Bound algorithm is the criterion according to which we choose the next node of the enumeration tree to be explored. In general, two fundamental criteria can be distinguished: the *best first* and the *depth first*.
The simple Branch and Bound procedure we summarized in Figure 17, due to its recursive nature, makes the choice of immediately and sequentially exploring the nodes generated by the branching operation. During the execution of the algorithm two cases may take place at a node of the enumeration tree. In the first case the bound of the node is greater than the best solution value that was found (in case of a maximization problem), so a further exploration is needed. This means that the subproblem is decomposed and the procedure goes on exploring a node having greater depth. In the second case the bound provides a feasible solution, sanctions the unfeasibility of the subproblem or verifies that the possible solutions being feasible for the subproblem are of no interest for solving the general problem. In that case, the recursive procedure terminates without further decomposing the problem and the algorithm goes on exploring another node at a higher level, thus performing a so-called *backtracking* operation. As it clearly appears, this implementation of the algorithm performs a deep search of the tree (depth first), giving priority to the rapid attainment of a leaf of the tree and therefore to the rapid generation of a feasible solution, even though nothing guarantees that the generated solution is of good quality. A further advantage of this type of enumeration tree search strategy is that the memory requirements of the algorithm execution are very small. In fact, during the execution we will have at most *k* "open" nodes for each level of the tree, where *k* is the greatest number of nodes generated by the branching rule. On the other hand, the main disadvantage derives from the fact that, if no good starting solution is available then, thanks to the bound rule, the subproblem removal from the enumeration tree becomes more improbable, thus the number of nodes to enumerate can be very high.
In order to lead the exploration of the enumeration tree towards the generation of solutions of good quality, and thus make the subproblem removal rule more effective, we might choose to explore first

the nodes providing a better bound (hence the definition "best first" criterion), which is obviously higher in case of maximization problems. This type of implementation of the algorithm cannot be carried out in a simple recursive form; on the contrary, it implies the introduction of an appropriate data structure memorizing the nodes still to be explored and from which at each iteration we select the node with highest bound value. This criterion allows to explore first the portions of the tree that are more "interesting" and that more probably contain the optimal solution. But unfortunately, due to the characteristics of the relaxations, the best bounds are provided by nodes being on a higher level in the enumeration tree. This implies that the number of "open" nodes can be very high (in principle, it is exponential in the number of tree levels) and that the memory requirements of the algorithm execution are heavy.
In the literature there are several algorithms of the "hybrid" type as well, which combine the two tree search criteria. For example, by setting an upper bound to the number of nodes being simultaneously open, it is possible to proceed to a best first search until such bound is achieved and then to continue with the depth first criterion, which does not affect the use of memory by the algorithm.

Exercise
Describe the pseudocode of the Branch and Bound algorithm formally, according to the best first criterion. What data structure can be used to memorize the tree nodes, taking into account the selection operations that must be performed?

*4. Notes on computational complexity

So far we have reviewed several optimization problems and we have studied algorithms for solving them. We mainly focused our attention on aspects concerning the correctness of algorithms, that is on the optimality guarantee of the detected solutions. Another fundamental aspect of algorithmic studies concerns the efficiency of the proposed algorithms, that is to say an evaluation of their performances, typically in terms of execution time or memory space.
In order to present the considerations related to algorithm efficiency with method, we need to introduce a few basic definitions. In the following pages the word *problem* will mean a question expressed in general terms and whose answer depends on a certain number of *parameters* and *variables*. A problem is usually defined by means of:
- a description of its parameters, which are generally left undetermined;
- a description of properties having to characterize the answer or the wanted *solution*.

For example, the traveling salesperson problem is specified in the following way: "On an undirected complete graph, where weights are associated with arcs, find the Hamiltonian cycle having the least total weight". The problem parameters are the number of graph nodes (no need to further specify, since the graph is complete) and the matrix of arc weights. The characterization of solutions is given by the specification of Hamiltonian cycles, and the solution we look for is the one in which the sum of the arcs composing the cycle is minimum.

An *instance* of a given problem P is that particular question obtained by specifying particular values for all parameters of P.
Therefore, in the traveling salesperson case, an instance is obtained by specifying the value of the number of nodes n and providing as input $n(n\text{-}1)/2$ weights to be associated with the graph arcs.

Formally, an *algorithm* for solving a given problem P can be defined as a finite sequence of instructions which, if applied to any instance p of P, stops after a finite number of steps (i.e., of elementary computations) providing a solution of the instance p or indicating that the instance p has no feasible solutions.

4.1 Computational models and complexity measures

Previously in the course, notably in Chapter 2, we measured the efficiency of an algorithm in a rather informal way; but in order to study algorithms from the viewpoint of their efficiency, i.e., their *computational complexity*, we need to define a computational model.
Classical computational models are the Turing Machine (historically, the first one to be proposed), the R.A.M. (Random Access Machine), the pointer Machine, etc.
A good compromise between simplicity and versatility is the R.A.M.
A R.A.M. consists of a finite program, of a finite set of registers, each of which can contain one single integer (or rational) number, and of a memory of n words, each of which has only one address (between 1 and n) and can contain one single integer (or rational) number.
In a step a R.A.M. can:
- perform one single (arithmetical or logical) operation on the content of a specified register,

- write in a specified register the content of a word whose address is in a register,
- store the content of a specified register in a word whose address is in a register.

It is a *sequential* and *deterministic* machine (the future behavior of the machine is univocally determined by its present configuration).

Given a problem P, its instance p and an algorithm A for solving P, we call *computational cost* (or *complexity*) of A applied to p a measure of the resources used by the computations that A performs on a R.A.M. in order to determine the solution of p. In principle, the resources are of two types: *engaged memory* and *computing time*. Supposing that all elementary operations have the same duration, the computing time can be expressed as the number of elementary operations performed by the algorithm. Since the most critical resource is quite often the computing time, in the following pages the computing time will be mainly used as a measure of algorithm complexity.

Given an algorithm, we need to have at our disposal a complexity measure allowing a synthetic evaluation of the good quality of the algorithm itself as well as, possibly, an easy comparison of the algorithm with alternative algorithms. It is not possible to know the complexity of A for each instance of P (the set of the instances of a problem is generally unbounded), nor would this be of any help in practice. We try therefore to express the complexity as a function $g(n)$ of the dimension, n, of the instance to which the algorithm is applied; yet, since there are usually many instances of a specific dimension, we choose $g(n)$ as cost needed to solve the most difficult instance among the instances of dimension n. This is defined *worst-case complexity*.
We obviously need to define the expression "dimension of an instance" more precisely. We call *dimension* of an instance p a measure of the number of bits needed to represent the data defining p with a “reasonably” compact coding, i.e., a measure of the length of its input. For example, in a graph with n nodes and m arcs, nodes can be represented by means of integers between 1 and n, and arcs by means of a list containing m pairs of integers (the arc connecting nodes i and j is defined by the pair (i,j)). Then, leaving out the multiplicative constants, it will be possible, as the number of nodes and arcs changes, to take as measure of the dimension of the graph coding the function $m\log n$; in fact, $\log n$ bits are sufficient to represent integers being positive and not greater than n[1]. For more simplicity, in the following pages we will leave out, beside multiplicative constants, sublinear functions as well, such as the logarithm function; in that case we say that m is the input length for a graph with m arcs. In the formulated hypotheses, the input length measure does not change if we use a basis coding $b>2$. Otherwise, if we use a unary coding, the input length in the considered example becomes nm, thus increasing significantly.
At this point, the previously introduced function $g(n)$ has been defined rigorously enough. In practice, however, it continues to be difficult to be used as a complexity measure; in fact, the evaluation of $g(n)$ for each given value of n proves to be difficult, if not practically impossible. The problem can be solved by replacing $g(n)$ by its order of magnitude; then we speak of *asymptotic complexity*.
Given a function $g(x)$, we can say that:

i) $g(x)$ is $O(f(x))$ if there exist two constants c_1 and c_2 such that for each x we have $g(x) \leq c_1 f(x)+c_2$;
ii) $g(x)$ is $\Omega(f(x))$ if $f(x)$ is $O(g(x))$;
iii) $g(x)$ is $\Theta(f(x))$ if $g(x)$ is at the same time $O(f(x))$ and $\Omega(f(x))$.

Let $g(x)$ be the number of elementary operations performed by an algorithm A being applied to the most difficult instance – among all instances having input length x – of a given problem P. We can say that the complexity of A is an $O(f(x))$ if $g(x)$ is an $O(f(x))$; similarly, we can say that the complexity of A is an $\Omega(f(x))$ or a $\Theta(f(x))$ if $g(x)$ is an $\Omega(f(x))$ or a $\Theta(f(x))$.

When we propose problem solving algorithms we always aim at getting a polynomial complexity in the input dimension. Unlike exponential algorithms, a polynomial algorithm is considered to be efficient in practice. The following tables clarify the reason of this distinction. Table 1 shows the execution times (for different complexity functions) on a computer performing one million elementary operations per second; Table 2 indicates improvements that can be obtained (in terms of dimensions of solvable instances and for different complexity functions) following computer technology improvements; with x_i we indicated the dimension of an instance today solvable in one minute for the i-th complexity function.

(1) In general, if not differently indicated, in case a problem's data are composed of n numbers, we will consider the number of bits needed for the coding in binary of the numbers themselves as being limited by $\log n$.

Function	**Dimensions of the instance** n=10	n=20	n=40	n=60
n	10^{-5} sec.	2×10^{-5} sec.	4×10^{-5} sec.	6×10^{-5} sec.
n^3	10^{-3} sec.	8×10^{-3} sec.	6.4×10^{-2} sec.	2.2×10^{-1} sec.
n^5	10^{-1} sec.	3.2 sec.	1.7 min.	13 min.
2^n	10^{-3} sec.	1 sec.	12.7 days	36.6×10^3 years
3^n	6×10^{-2} sec.	1 hour	4×10^5 years	1.3×10^{13} years

Table 1: execution times on a computer performing 10^6 operations per second according to input dimension

Complexity function	**Present-day computer**	**100 times faster**	**10000 times faster**
n	x_1	$100\ x_1$	$10000\ x_1$
n^3	x_2	$4.6x_2$	$21.5x_2$
n^5	x_3	$2.5x_3$	$6.3x_3$
2^n	x_4	$x_4 + 6.6$	$x_4 + 13.2$
3^n	x_5	$x_5 + 4.2$	$x_5 + 8.4$

Table 2: increment of input dimensions according to computer speed and within the same execution time

In Table 1 we can notice how, in case of polynomial algorithms, the execution time grows according to input dimensions but remains at acceptable levels, whereas in case of exponential algorithms such time, though still competitive for small instances, soon becomes unsustainable. This difference appears in Table 2 as well, where polynomial algorithms benefit from computing speed progress increasing by a multiplicative factor the input dimension of instances solvable in one minute, whereas in case of exponential algorithms the progress is given by an additive constant.
In Chapter 2 we examined the complexity of several algorithms, some of which – for example the graph search algorithm, the algorithms for determining the shortest-path tree, the Edmonds-Karp augmenting-path algorithm – are evidently polynomial. A clearly exponential algorithm is the Branch and Bound algorithm we already presented in this chapter: in the worst case it must explore the whole enumeration tree which is exponential in the number of variables.
Actually, in the classification of algorithms according to their computational complexity we have an intermediate range. Consider for example the negative cycle removal algorithm for a minimum cost flow problem or, for the same problem, the algorithm of successive shortest paths. In the evaluation of the complexity of these algorithms, we have a polynomial among whose parameters there is not only the input dimension, but also a term relative to data (maximum capacity or maximum cost of arcs). For example, we have a complexity of $O(nm^2C_{max}U_{max})$ for the negative cycle removal algorithm. If C_{max} and U_{max} take particularly high values, the complexity grows, and considering that C_{max} and U_{max} are codified "in binary", the complexity grows exponentially in the input dimension. Yet, if instead of having recourse to a binary coding, we "trick" and codify the input with a unary code, then the complexity is polynomial in the input dimension, thus making our algorithm fall into the class of polynomial algorithms.

An algorithm is said to be *pseudopolynomial* if it can solve one of these problems in polynomial time in the length of the input codified “in unary”.

Exercises
Classify the algorithms we saw in Chapter 2 according to the distinctions: "polynomial", "exponential" and "pseudopolynomial". In particular, into which class does the capacity scaling algorithm for maximum flow fall?

4.2 Tractable and untractable problems

In light of the complexity-based algorithm classification we introduced previously, we can try to classify problems as well according to the algorithms for solving them. Problems for which there exist algorithms whose complexity is an $O(p(x))$ with $p(x)$ being a polynomial in the input dimension x are said to be *tractable* and problems for which such an algorithm does not exist are said to be *intractable*.
The critical point of this criterion of classification lies precisely in stating that for a certain problem there exists no polynomial solution algorithm. If we cannot find an efficient algorithm for a certain problem, to what extent is this due to its real difficulty and to what extent to our little capacity? This

question, which makes the classification rather delicate, was only solved in the early Seventies with the theory of computational complexity.
In order to perform a more rigorous classification we have to refer to problems in decisional form, that is to problems implying the answers YES or NO. Problems of this kind are, for example, the satisfiability problem we saw in Chapter 1 and the graph connection problem we saw in Chapter 2. Note that an optimization problem can always be reduced to a problem in decisional form. Consider for example the problem:

P: max $\{ cx: x \in F\}$.

By introducing a parameter k, the decisional version P_k of problem P is the following:

$P_k : \{ x \in F, cx \geq k\} \neq \emptyset$?

whose answer is a YES or a NO, and wishing to certify the answer YES we just need to provide a solution $x \in F$ such that $cx \geq k$.
If an efficient algorithm for solving problem P_k (for any value of k) is available, we can efficiently solve problem P as well. In fact, we just need to perform a binary research on the optimal solution value, solving a decisional problem P_k at each step and refining the research according to the positive or negative answer of P_k. In case of positive answer we concentrate our research on values being greater than k, otherwise we seek the solution among those having value smaller than or equal to k. If $O(p(n))$ is the complexity of the algorithm solving P_k, with n as input dimension, and $(\underline{K}, \overline{K})$ are an underestimation and an overestimation of the optimal solution value of P, the complexity of this binary research algorithm is $O(\log (\underline{K} - \overline{K})\, p(n))$. Hence, if $p(n)$ is a polynomial in n, then also the binary research algorithm continues to be a polynomial algorithm.

<u>Example: set of maximum cardinality independent nodes</u>
Given an undirected graph $G=(N,A)$, we call a set of independent nodes a subset $S \subseteq N$, such that for each pair of nodes i and j belonging to the independent set, there exists no corresponding arc in A. Hence

ISP: max $\{ |S|: \forall i,j \in S, (i,j) \notin A\}$,

and the corresponding decisional version, given a bound $k>0$, is:

$ISP_k : \{ S: \forall i,j \in S, (i,j) \notin A, |S| \geq k\} \neq \emptyset$?

An important class of problems is the **NP** class, which consists of all *decisional problems certifiable in polynomial time* by a R.A.M. machine. This means that, given any instance of a decisional problem P and a solution x, there exists a polynomial algorithm that checks if the given solution is an answer YES for P.

<u>SAT example</u>
In Chapter 1 we introduced the satisfiability problem (SAT) which, as we shall see, plays a fundamental role in the theory of computational complexity. It is given a logical formula in conjunctive normal form $A=C_1 \wedge \ldots \wedge C_m$ in which each clause C_i is a disjunction of literals ($\pm P_j$) and the elementary propositions composing the formula are $P_1, \ldots, P_n$. The problem consists in finding an assignment of truth values to the elementary propositions $P_1, \ldots, P_n$ rendering the formula true. The problem is of the decisional type, the input dimension is $O(mn)$, since that the formula is composed of m clauses, each of which contains at most n literals. The certificate of an answer YES is given by the value assignment, therefore it has length $O(n)$, hence SAT belongs to **NP**.

<u>Example: connected graph</u>
Let us examine the problem of determining the connectivity of a digraph that we saw in Chapter 2. Given a digraph $G=(N,A)$ and two nodes s and t of N, the problem consists in establishing if there exists a path directed from s to t. The problem is obviously of the decisional type. Denoting by n and m the number of nodes and the number of arcs of the graph, respectively, then the input dimension is $O(n+m)$. The certificate of an answer YES is given by the path itself that connects s to t and that can be therefore represented by a sequence of at most n-1 successive arcs $(s,i_1), (i_1,i_2), \ldots, (i_r,t)$. So, we can

conclude that the problem is in **NP**. Note that in this case the answer NO is polynomially certifiable as well; in fact, the non- existence of a path between s and t is proved by the existence of a partition of the set of nodes N into two subsets N_s and N_t such that $s \in N_s$, $t \in N_t$ and all the arcs with endpoints spanning the partition have tail in N_t and head in N_s.

Example: set of independent nodes
Let us consider again the problem of the set of maximum cardinality independent nodes (*ISP*) we have just presented. A possible solution S is given by a subset of nodes, hence the input into the certification problem, given by the graph $G=(N,A)$, by the subset S and by the constant k, has dimension $O(m+n)$. In order to verify whether S really is a solution to the problem, first of all we have to verify that $|S| \geq k$ and afterwards, by sequentially examining each arc of A, we can verify whether both endpoints are in S. If not, the given solution is an answer YES. The complexity of this operation is $O(m)$, so the problem of the maximum set of independent nodes in decisional form is in **NP**.

Exercise
How can we certify an answer NO to the satisfiability problem? Is it possible to give a polynomial certificate of it?

A subset of **NP** class is the so-called **P** class of problems solvable in *polynomial time*, which contains all decisional problems for which there are polynomial complexity algorithms that solve them (*polynomial problems*).

Example: connected graph
In Chapter 2 we saw that by means of an algorithm searching the graph from node s we could provide a solution to the connection problem in $O(m)$ time. This means that the problem belongs to the **P** class.

A very important question is whether in **NP** there are problems we can guarantee as not belonging to **P** as well, that is whether **P≠NP**. For example, the SAT problem and the decisional version of the ISP problem, for which up to now no polynomial solution algorithms have been provided yet, are in **P**: are we not skillful enough to find an algorithm or are they intrinsically difficult? No one has been able to answer this question, although it is more than likely that the answer is positive, i.e., that actually **P≠NP**.

4.3 **NP***-complete problems and* **NP***-hard problems*

With the purpose of characterizing intrinsically difficult **NP** problems in some way, a specific subset of problems was introduced. This class, known as ***NP-complete problems***, represents the core of the **NP** problems that are considered to be difficult. In order to define this class formally we need to introduce the concept of polynomial reduction, which allows to relate two problems.

Though apparently quite different, many problems can be related one to the other. Given two decisional problems, P and Q, we say that P is reduced to Q in polynomial time (thus we write $P \propto Q$) if, assuming the existence of an algorithm A_Q that solves Q in constant time (i.e., independent from the input length), there exists an algorithm that solves P in polynomial time by using A_Q as a subprogram. In this case we speak of *polynomial reduction* of P to Q.

Example: reduction from Hamiltonian cycle to TSP
Let us show how the problem of the existence of an Hamiltonian cycle (*HC*) on any symmetric graph $G=(N,E)$ can be polynomially related to the traveling salesperson problem (*TSP*) on complete symmetric, weighted graphs, in its decisional version; in other words: how to determine the existence of an Hamiltonian cycle with weight smaller than or equal to a pre-set value k. Let $A_{TSP}(N,c,k)$ be an algorithm that, given a set of nodes N and a real function $c : N \times N \rightarrow \boldsymbol{R}$, returns YES if there exists a set of arcs forming an Hamiltonian cycle with cost $\leq k$, and NO otherwise. The following algorithm solves *HC* polynomially, supposing that the complexity of $A_{TSP}(N,c,k)$ is $O(1)$:

```
Procedure Hamiltonian_Cycle(N,E):
  begin
    for all i,j∈N do if (i,j)∈E then c(i,j) := 0 else c(i,j) := 1;
    return A_TSP(N,c,0)
  end.
```

Hence, we can state that $HC \propto TSP$. In practice, this means that *HC* is no more difficult to solve than *TSP*. In fact, if there existed a polynomial algorithm for *TSP*, hence *TSP*∈**P**, then also *HC*∈**P**.

Conversely, if we could prove that *HC* admits no polynomial algorithms, the same would apply to *TSP* too. These important considerations are summarized in the following property:

Property 4.1
- If $P \propto Q$ and $Q \in \mathbf{P}$, then $P \in \mathbf{P}$.
- If $P \propto Q$ and $P \notin \mathbf{P}$, then $Q \notin \mathbf{P}$;

The relation $\propto$ has the following properties:

i) is reflexive: $P \propto P$;
ii) is transitive: $P \propto Q$ and $Q \propto R \Rightarrow P \propto R$.

Precisely the transitivity of the reduction is helpful in defining the class of **NP**-complete problems and in proving in a practical way that a problem is **NP**-complete.

Definition 4.2
A problem Q is said to be **NP**-*complete* if $Q \in \mathbf{NP}$ and if for each $P \in \mathbf{NP}$ we have $P \propto Q$.

Following the definition, the task of proving the **NP**-completeness of a problem might appear impossible, because it requires countless reductions. In 1971 Cook solved this difficulty by demonstrating that any instance of a problem *P* of **NP** turns into an appropriate logical formula with polynomial length in the input dimension of *P*. We can deduce that any problem in **NP** is reduced to *SAT* in polynomial time.

Theorem 4.3 (Cook, 1971)
Each problem $P \in \mathbf{NP}$ is polynomially reduced to *SAT* (satisfiability problem).

From Cook's theorem follows that *SAT* is an **NP**-complete problem and that it constitutes a major point of reference. The theorem's proof, which is rather technical, represents a milestone in the computability theory and should not be missing from the store of knowledge of a good information scientist. For deeper interest, we refer to the proof that appears in [C. H. Papadimitriou and K. Steiglitz, 1982].

NP-complete problems have the property that if there exists a polynomial algorithm for one of them, then all problems in **NP** are necessarily solvable polynomially, so we would have **P** = **NP**. In a sense, such problems are the most difficult problems in **NP**.
Figure 23 shows the relations between **NP**, **P** and the class of **NP**-complete problems.

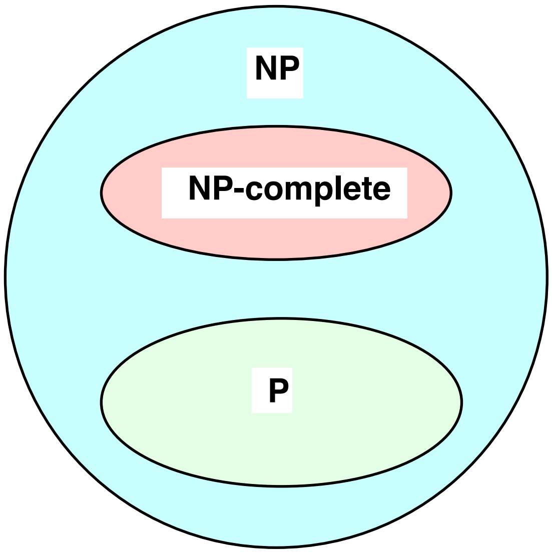

Fig. 23: scheme of problem classification

Once proved that *SAT* is **NP**-complete, we can prove that a problem *P* belongs as well to the class of **NP**-complete problems by simply reducing *SAT* to *P*, and according to the transitivity of the reduction we will obtain that any problem in **NP** is reduced to *P*. Note that at this point *P* too becomes useful for proving the **NP**-completeness of other problems. Thus, in order to prove a problem's **NP**-completeness, it generally suffices to take any **NP**-complete problem as reference and find an appropriate reduction. Let us give an example while finding a polynomial reduction from *SAT* to the problem of the set of independent nodes (*ISP*).

Theorem 4.4 [Karp 1972]
ISP is **NP**-complete.

Proof
First of all note that, as we already saw, *ISP* is in **NP** because a solution of an instance YES can be verified in polynomial time.
Consider now an instance of the *SAT* problem – i.e., the problem of determining whether a given logical formula A admits an assignment of truth values that renders it true – which, as we know, belongs to the class of **NP**-complete problems. For more simplicity, we can assume that A is in conjunctive normal form, i.e., is specified by a set of clauses $C_1 \wedge\ldots\wedge C_m$, each of which is given by the disjunction of literals (direct or negated logical variables): $C_i = \pm X_1 \vee\ldots\vee \pm X_{i_k}$. Globally, the set of logical variables is given by $\{X_1, \ldots, X_n\}$.
Now let us build an undirected graph $G=(N,E)$ corresponding with the formula A. For each clause we introduce as many nodes as the literals composing it. We also introduce arcs connecting the nodes of each clause to one another, thus forming many complete subgraphs $G_i=(N_i,E_i)$ for each clause C_i. The graph can be completed by introducing arcs E' connecting, in pairs, the nodes relative to the logical variables that appear with different signs in different clauses. Note how in the following example we can build the graph of Figure 24 from the formula: $A=(X_1\vee\neg X_2\vee X_4)\wedge(X_3\vee\neg X_4)\wedge(\neg X_1\vee X_2\vee\neg X_3)\wedge(X_2\vee\neg X_3)$.

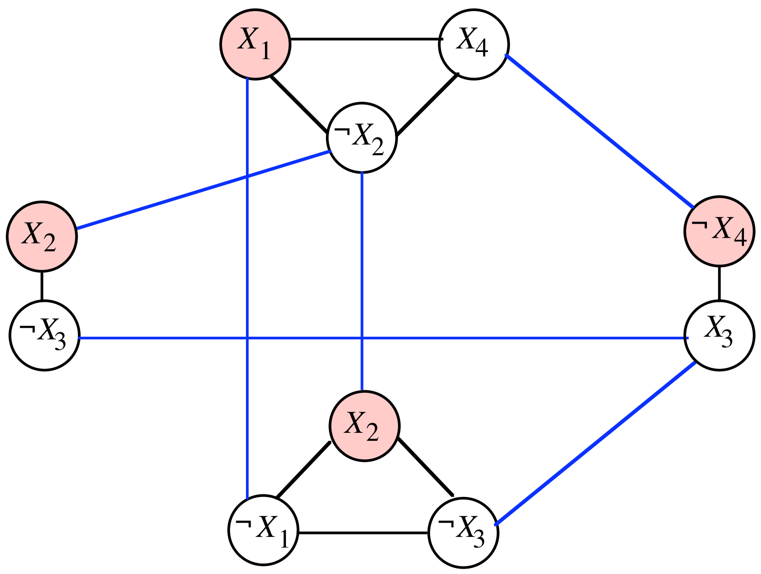

Fig. 24: instance of a set of independent nodes that is associated with a logical formula

If the graph, formed in this way, has a set of independent nodes I with cardinality m, then we have detected a set of m literals, each belonging to a different clause and compatible two by two. We can obtain an assignment of truth values as follows. We assign value **true** to variables corresponding to nodes in I if variables are direct, and we assign value **false** if variables are negated; we assign an arbitrary value to other possible variables to which no value was assigned.
Conversely, if there exists an assignment of truth values rendering A true, thus rendering each of m clauses true, then we can build a set of independent nodes by choosing a node corresponding to a true literal for each clause. ♦

In the example of Figure 24 there exists a set of four independent nodes that are associated with literals X_1 in the first clause, $\neg X_4$ in the second one, X_2 in the third and the fourth one. An assignment of truth values relative to this set of nodes is $X_1= X_2 =$ **true**, $X_4 =$ **false**. This assignment of values to variables renders formula A true for any value of X_3.

Exercise
Given an undirected graph $G=(N,E)$, let us consider the problem of determining whether in G There exists a complete subgraph (called *clique*) having cardinality k. Prove that such problem is **NP**-complete by making use of a technique similar to the one used for *ISP*.

A problem having as particular case an **NP**-complete problem has the property of being at least as difficult as **NP**-complete problems (except for a polynomial multiplicative function). A problem of this type is called **NP**-hard. It should be observed that an **NP**-hard problem can also not belong to **NP**. For example, optimality problems whose decisional version is an **NP**-complete problem are **NP**-hard. Thus, the optimization version of *ISP* is **NP**-hard.

When we speak of input length we assume that the way used to codify the data defining the problem was reasonably compact. For example, we can assume that if x is the problem length with the coding that was used and y the length with any other coding, then x is an $O(f(y))$, where f is a polynomial (possibly with degree 1).
For example, the integer number a has length $x = \log a$ if codified "in binary", whereas it has length $y = a$ if codified "in unary". Clearly, the coding "in unary" is not acceptable with respect to the criteria presented above; in fact y grows exponentially with x.
Yet, it should be noted that there exist problems that are polynomially solvable in the input length if codified "in unary", whereas they are not so with more compact codings. Such problems are in a sense easier than those requiring exponential algorithms apart from the input coding. For example, we saw that the knapsack problem admits a pseudopolynomial algorithm that makes use of the search for a longest path on an appropriate graph.
We call **NP**-complete *in the strong sense* an **NP**-complete problem for which there exist no pseudopolynomial algorithms. Similarly, we call **NP**-hard *in the strong sense* an optimization problem having as particular case an **NP**-complete problem in the strong sense.

A very useful tool for determining results pertaining to computational complexity of combinatorial problems is the compendium edited by P. Crescenzi e V. Kann:
http://www.nada.kth.se/~viggo/wwwcompendium
The Web site offers a synthetic historical memory of complexity results appearing in the literature, it is regularly updated and presents new results, new problems and an up-to-date bibliography.

5. Approximate algorithms

When confronted with instances of an **NP**-hard problem, we know that solution time is likely to be long. It frequently happens, however, that the solve time available is limited, so we have to diminish our ambitions and be satisfied with a good solution, not necessarily optimal but quickly obtained. Moreover, note that optimality is quite often an important property from the theoretical viewpoint, but its meaning is uncertain in practice. In Chapter 1 we saw that there is always some distance between reality and model. Such distance derives also from the difficulty (or quite often even the impossibility) to know exactly all the quantities that are involved; thus, the coefficients appearing in the model are encumbered by errors that in most cases are substantial: some constraints may have been left out, etc. In several cases, this contributes to make the search for the "theoretical optimality" of little significance from the practical point of view.
In such cases we make use of *approximate algorithms*, also known as *heuristics* because they utilize the knowledge of the particular problem's structure and of the characteristics of optimal solutions.
When using heuristic algorithms we should (although it is not always easy or possible) perform an analysis of the error we are making and accept the obtained solution instead of the optimal one.
Consider the problem $\min \{c(x): x \in F\}$.
Let A be an algorithm for such problem and x_A the (not necessarily optimal) solution it provides, and let:

$$z_{OPT} = \{\min c(x): x \in F\},$$

$$z_A = c(x_A).$$

We define:

absolute error $\qquad E_A = z_A - z_{OPT}$

relative error $\qquad R_A = \dfrac{z_A - z_{OPT}}{z_{OPT}}$

An algorithm A is *ε-approximate*, with $\varepsilon > 0$, if:

$$R_A \leq \varepsilon \;.$$

An *approximation scheme* is an approximate algorithm such that, for each $\varepsilon>0$, it produces a solution with $R_A \leq \varepsilon$. The complexity of such an algorithm is obviously a function of the wanted approximation ε (growing with $1/\varepsilon$).
An approximation scheme is said to be:

- *polynomial*, if its complexity is polynomial in n (input dimension);
- *fully polynomial*, if its complexity is polynomial in n and in $1/\varepsilon$.

An interesting branch of the theory of complexity concerns problem *approximability*. In case of **NP**-hard problems, this means determining particularly difficult classes which do not admit polynomial approximation schemes.

Like when dealing with the modeling of decisional problems, in the case of heuristics as well it is difficult to systematize the exposition and provide an infallible "recipe" for producing good algorithms. The best heuristic algorithms are actually the fruit of experience, at times of luck, and of knowing the specific characteristics of the different problems. Considering the difficulty of the task – which would require a complete course – and the little space available, we will just describe two very general algorithmic schemes that we already applied to several problems we saw in the previous chapters. The first one is the class of *greedy* algorithms that iteratively build feasible solutions without ever going back to choices made previously. The second one is the class of *local-search* algorithms that, starting from a feasible solution, seek possible better solutions obtained by making slight changes in the given solution. The two types of algorithm can be used in sequence by means of a greedy algorithm generating a feasible solution to be improved afterwards using a local-search algorithm.

5.1 Greedy algorithms

Before presenting the formal description of the scheme for algorithms of the greedy type, let us analyze a simple example and try to build a relevant solution relying on "common sense".

Example: constructive algorithm for the knapsack problem
Consider the instance of the knapsack problem in the burglar example of Chapter 1:

$$\begin{array}{ll} \max & 7x_1 + 2x_2 + 4x_3 + 5x_4 + 4x_5 + x_6 \\ & 5x_1 + 3x_2 + 2x_3 + 3x_4 + x_5 + x_6 \leq 8 \\ & x_1,\ x_2,\ x_3,\ x_4,\ x_5, x_6 \in \{0, 1\}. \end{array}$$

A presumably good, though generally not optimal, solution can be obtained by proceeding according to the following informal scheme. Indexes of objects are arranged in a descending unit profit order; then variables are examined following the established order: each variable is assigned value 1 if adding the object does not violate the capacity constraint, otherwise it is assigned value 0, and in fact the corresponding object is definitively discarded.
In the examined case, the steps that the described method would take are the following:

1. The object having greater unit cost is the fifth; its ratio cost/weight is equal to 4; as the object has weight 1 and the residual capacity of the knapsack at this moment is 8, we set x_5=1, thus leaving enough room in the knapsack for 7 further weight units.
2. The following object in the selected order is the third; its unit cost is 2; its weight is less than the residual capacity, so we set x_3=1; the residual capacity of the knapsack descends therefore to 5.
3. The following object is the fourth; its unit cost is 5/3; in this case too, since there is enough capacity available, we can set x_4=1; the knapsack still has a residual capacity of 2 units.
4. The following object is the first; its unit cost is 7/5 but its weight (5) is greater than the residual capacity of the knapsack (2), so the corresponding variable x_1 is set at zero.
5. The following object is the sixth, which has unit cost 1; since its weight is 1, we can set x_6=1; the knapsack still has a residual capacity of 1 unit.
6. The last object to be examined is the second; since it has weight greater than the residual capacity of the knapsack, it is discarded and the variable x_2 is set at zero.

The obtained solution is therefore (0, 0, 1, 1, 1, 1), with cost 14 and total weight 7. It clearly appears that this solution is not optimal: in fact, the solution (1, 0, 1, 0, 1, 0), with total weight 8, has cost 15. Thus, the described algorithm does not guarantee the optimality of the solution; yet, it has the advantage of being simple to implement and computationally very efficient, because its complexity – except for the ordering of objects – is linear in the input dimension.

An algorithm of this type is called *greedy* algorithm. it is a kind of algorithm that determines the solution through a sequence of partial decisions (locally optimal), without ever going back to already made decisions and modifying them. These algorithms are easy to implement and considerably efficient from the computational point of view, but, with a few important exceptions, do not ensure the optimality and at times not even the feasibility. The class of problems that can be optimally solved by greedy algorithms is known as the class of *matroids*. For a deeper knowledge of matroids we refer to Lawler's book.

In order to present these algorithms formally, we usually refer to problems represented in the form of systems of independent sets.
We detect a set E (*ground set*) that can be considered as the set of elements composing the solution of the examined problem. In the case of the knapsack problem the set E is given by the set of objects to choose. Further, we detect the set F, which is given by a family of subsets of E ($F \subseteq 2^E$), such that the property of closure by inclusion holds:

$$(A \subseteq B) \text{ e } (B \in F) \Rightarrow A \in F.$$

the set F represents the set of all partial solutions being feasible for the problem. So, we call:
F a *family of independent sets,*
(E,F) a *system of independent sets*.

A set $I \in F$ is called a *maximal set* if:

$$e \in E \setminus I \rightarrow I \cup \{e\} \notin F.$$

Given a function

$$c: 2^E \rightarrow \boldsymbol{R},$$

we call a *problem associated* with (E,F) the following optimization problem:

Determine a maximal set $I \in F$ such that:
$c(I) \geq c(Y)$, $\forall\ Y \in F$ and maximal.

Similarly, we can make a minimization problem correspond with (E,F).

It is given the optimization problem associated with the system of independent sets (E,F) with objective function c; an algorithm of the greedy type builds the set $X \in F$ (the solution of the problem), starting from the empty set and, at each step, inserting in it the most "promising" element of E among those not violating the set independence.
This type of algorithm, in its most general form, is described by the following procedure **greedy**:

```
Procedure greedy:
  begin
    $X := \emptyset$;  $S := E$;
    repeat
      $e :=$ Best($S$);  $S := S \setminus \{e\}$;
      if Ind($X,e$) then $X := X \cup \{e\}$
    until $S = \emptyset$
  end.
```

The algorithm makes use of the two subprocedures **Best** and **Ind**:

Best provides the best element of S according to a pre-set criterion (for example the element with minimum cost in the case of the minimum cost spanning tree problem, or the element with the maximum ratio of v_i/p_i in the knapsack problem).

Ind is a logical function that is defined as follows:

$$\textbf{Ind}(X,e) = \begin{cases} \mathit{true}, & \text{if } X \cup \{e\} \in F, \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

In the knapsack example considered at the beginning of the paragraph, F is the set of all subsets of elements whose total weight is less than or equal to the knapsack's capacity (i.e., the set of all feasible solutions).
Let us now analyze the complexity of the procedure **greedy**.
Let $|E| = n$; we denote by $h(n)$ the complexity of **Best** and by $k(n)$ the complexity of **Ind**. Then, the complexity of the whole procedure is $O(n(h(n)+k(n)))$. Suppose that the elements of E are ordered at the beginning, hence at each step **Best** provides the first among the remaining elements; then the complexity becomes $O(n\log n+nk(n))$.

Example: set covering
In the set covering example, let there be a set I of elements and a family $J=\{J_1,\dots,J_m\}$ of subsets of I. A cost c_j is associated with each subset I_j. The purpose is to determine a selection of subsets of the family such that each element of I appears in at least a selected subset and the sum of the costs of the choice that was made is minimized. Such problem is **NP**-hard, as it is confirmed in the summary available at the Web site we mentioned in the pages concerning problem complexity. Let us formulate the problem in terms of system of independent sets. In this case the set E is given by subsets of family J, or alternatively by the set of indexes $\{1,\dots,m\}$. The family of independent sets F is given by the set power of E, hence by the set of all possible subsets of indexes. In fact, note that the solution $X=E$ too, although probably quite inefficient, is feasible for the set covering. Wishing to specify the greedy algorithm, we just need to define the details of the functions **Best** and **Ind**. The **Best** can be led by cost c_j if we choose the minimum cost subset in S, or it can be led by the set cardinality if we give priority to larger subsets, or else a compromise between these two criteria can be found if we give priority to subsets whose cost for each element is minimum:

$$\mathbf{Best}(S) = \mathrm{argmin}\{c_j : j\in S\}$$
$$\mathbf{Best}(S) = \mathrm{argmax}\{|J_j| : j\in S\}$$
$$\mathbf{Best}(S) = \mathrm{argmin}\{\frac{c_j}{|J_j|} : j\in S\}$$

On the other hand, as any subset of indexes represents a partial feasible solution, the function **Ind**, always returns true and no filter must be applied. In this way we obtain a correct greedy algorithm. However, wishing to go more into details, we see that the algorithm, just as it is specified above, can yield "redundant" solutions, i.e., solutions in which there are some subsets merely containing elements that are already covered by other selected subsets. To overcome this difficulty we can modify the **Ind** in the following way:

$$\mathbf{Ind}(X,j) = \begin{cases} \textit{true}, & \text{if in } J_j \text{ there are elements of } I \text{ that are not contained in any } J_i \text{ for } i\in X, \\ \textit{false}, & \text{otherwise.} \end{cases}$$

This **Ind** ensures that, at each iteration, the greedy algorithm for the set covering increases the set of covered elements, thus it actually prevents redundant solutions. Note that, according to how the function **Best** was described, the criteria adopted to choose a subset are static; therefore, as it was the case in the knapsack problem, we could make use of a starting ordering of all subsets according to the particular function that was chosen. Yet, it should be observed that in the second and the third definition of the **Best** we refer to the subset cardinality. If, instead of considering the cardinality in an absolute sense, we considered the cardinality of the set of the new elements that would be added to the current solution, we could prevent redundancies, without having to specialize the function **Ind**, and at the same time we would obtain a more sophisticated criterion of choice. This "dynamic" criterion of choice would make the starting ordering useless and, at each call of the **Best**, would require a complete scansion of the set S, thus making the algorithm complexity slightly heavier.

The greedy algorithm, in its general form, can be applied to many problems without any particular implementative tricks, as we saw for the knapsack and the set covering problems. In the case of more complex problems, it can be worth while to specialize the greedy in order to make full use of the structure and of the specific features of the examined problem. Let us look at the case of the traveling salesperson problem.

Example: greedy algorithms for the traveling salesperson problem

The problem is specified by an undirected complete graph $G=(N,A)$ with costs associated with arcs $c_{ij}\geq 0$. A banal way of yielding a greedy algorithm for the traveling salesperson problem proceeds by applying the general rules described above; so, considering that a solution can be viewed as a subset of arcs, we can define $E=A$ and $F=\{H\subseteq A$: H is a Hamiltonian circuit or a subset of the latter and not necessarily connected$\}$. The **Best**(S) just returns the arc of minimum cost among those in S:

$$\textbf{Best}(S) = \text{argmin}\{c_{ij}: (i,j)\in S\}$$

and the **Ind** verifies whether adding an arc to the current solution allows to reach the end of the algorithm with a Hamiltonian circuit:

$$\textbf{Ind}(X,(i,j)) = \begin{cases} true, & \text{if } X\cup\{(i,j)\} \text{ does not contain nodes with three incident arcs or subcycles,} \\ false, & \text{otherwise.} \end{cases}$$

The defect of this algorithm is that it chooses the best arcs at once, but then must hold on to choices that may be arbitrarily bad.

Another version of greedy algorithm for the TSP slightly deviates from the general scheme described above, although it maintains the feature of building a feasible solution incrementally by making locally optimal choices.

The idea is to start from a solution already being a circuit, though not Hamiltonian, and increase it at each iteration by adding the node implying the shortest deviation. Let (i,j) be the arc of minimum cost of the graph; we can take the circuit i-j-i as starting solution. At each iteration we consider each arc (h,k) of the circuit and each other node y of the graph which is not included in the solution yet; further we compute the global cost of the deviation: $s_{hky} = c_{hy} + c_{yk} - c_{hk}$, and we choose the triplet h,k,y minimizing the value of the deviation s_{hky} and we insert the node y in the most advantageous point of the circuit, hence between node h and node k. Using the sequence of nodes of the circuit as representation of the solution X, the functions specifying the algorithm become:

$$\textbf{Best}(S,h,k) = \text{argmin}\{s_{hky} = c_{hy} + c_{yk} - c_{hk},: y\notin X,(h,k)\in X\}$$

$$\textbf{Ind}(X, y, h, k) = true$$

where the **Best** returns, beside the index of the node to be inserted, also the pair of nodes (h,k) (adjacencies of the current circuit) into which the new node must be inserted. According to this criterion of choice, the **Ind** becomes superfluous. This algorithm is known in the literature as *nearest insertion*.

Exercise

Provide an efficient data structure to implement the nearest insertion algorithm.

Countless constructive algorithms for the TSP have been proposed over the years, all referring to a greedy logic. Let us look at a further greedy algorithm that makes use of the result of one of the equally numerous relaxations of the problem. Consider the relaxation of the TSP obtained by leaving out the subcycle removal constraints. As we saw previously, the problem corresponds to an assignment on bipartite graph and its result is a circuit cover of graph G on which the TSP problem is defined. If we are lucky, we obtain only one Hamiltonian circuit corresponding to the optimal solution of the problem; if not, we will have many subcircuits. The algorithm examines these circuits and "patches them up" in pairs, until it obtains only one circuit.

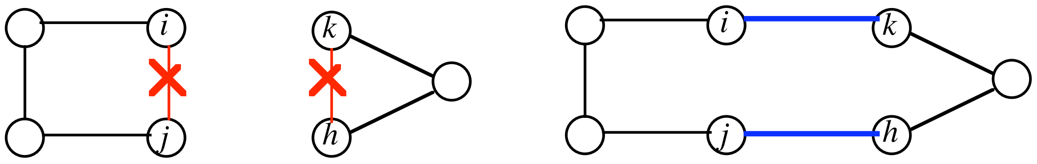

Fig. 25: the "patching" of two circuits

Figure 25 illustrates the process of patching up two circuits. We detect two arcs (i,j) and (h,k) in two different circuits which will be replaced by the arcs (i,k) and (j,h) (or (i,h) and (j,k)). The new solution will have an additional cost of $P(i,j,h,k)=c_{ik} + c_{jh} - c_{ij} - c_{hk}$. Algorithms based on this technique are called "*patching*" algorithms; at times, instead of costs, we use the reduced costs resulting from the solution of the assignment problem. It is not difficult to define the details of a greedy algorithm based on patching.

Exercise

Specify the representation of the partial solution as well as the functions **Best** and **Ind** for a greedy algorithm based on the patching technique.

Greedy algorithms are usually extremely fast, but yield only one solution. When approaching a complex problem, the time available may be enough to allow attempts at producing more than one heuristic solutions. In such situations, however, the greedy is of no help because, if called repeatedly on the same input, it always yields the same solution. A way to avoid this defect is to render the choice of the **Best** non-deterministic, so that if the algorithm is called more than once, at each execution – at least in principle – different choices are made and alternative solutions are generated. This idea forms the basis of GRASP (Greedy Randomly Adaptive Search Procedure) algorithms. Using our greedy procedure scheme, the only modification concerns the function **Best**. The new function, summarized below, takes as input the set S of elements that are candidate to enter the solution, like in the case of the traditional greedy, but also a parameter $0<\alpha<1$ establishing a deviation threshold from the value of the best choice. The **Best_GRASP** takes into consideration elements not deviating from the best in S by more than α%, randomly selects one of these and returns it as result.

```
Function Best_GRASP(S,$\alpha$)
  begin
    v := min ($c_j$: j$\in$S}
    S' := {j: $c_j \geq$ v (1-$\alpha$)}        {selection of elements being no worse than $\alpha$v in relation to the best}
    return (random_selection(S'));
  end.
```

In practice, GRASP algorithms, that are based on the simple idea we summarized above, contain many other "ingredients" and expedients, and are usually accompanied by appropriate local-search algorithms. For deeper study of the subject, see [], or look at the Web site of Mauricio Resende: http://www.research.att.com/~mgcr/grasp/index.html

5.2 *Local-search algorithms*

Local-search algorithms examine a feasible solution and iteratively try to improve it. During the course we have been dealing with various local-search algorithms, such as augmenting-path algorithms for the maximum flow problem, or the negative cycle removal algorithm for the minimum cost flow problem, or again the Primal-Dual Simplex algorithm for linear programming. These algorithms are characterized by the fact that at each step they examine a feasible solution, seeking a better one among those that are "near" to it, thus restricting the search (hence the term "local") to a *neighborhood* of the current solution. So, we can try to provide a general description of this algorithmic paradigm with the aim of using it as a guide for yielding new algorithms for a variety of problems, as we did for the greedy.

In general, a local-search algorithm is characterized by the following operations:

- we determine a starting feasible solution x^0;
- we iteratively explore a sequence of feasible solutions, $x^0, x^1,\dots, x^k,\dots$, and we stop as soon as one of the detected solutions is recognized as being an optimal or not improvable solution.

The search for the solution improving the current solution x^i is performed in a subset of feasible solutions near to x^i, which is called *neighborhood of x^i* and is denoted by $I(x^i)$. In general, when defining a neighborhood, the following rules are to be considered, recalling that F indicates the set of feasible solutions of the problem:

1) $x \in F \Rightarrow x \in I(x)$;

2) $x, y \in F \Rightarrow \exists \{x^0, x^1, ..., x^p\} \subseteq F,$ con $x^0 = x, x^k \in I(x^{k-1}), k = 1,\dots, p, x^p = y.$

The first rule implies that each solution belongs to its own neighborhood, so as to ensure a stopping condition for the algorithm. The second rule, more difficult to verify, ensures that, given any two solutions, it is possible to "migrate" from one to the other through a sequence of modifications compatible with the definition of neighborhood, even though the resulting sequence of solutions is not necessarily best-oriented. This rule, at least in principle, guarantees that it is possible to generate any solution from any starting solution. Occasionally, however, the constraints of the examined problem are so stringent that they cannot guarantee this second property for a certain neighborhood. Later in the course we will see how to overcome this difficulty.

A further essential element of this kind of algorithm is a function σ, which, given a solution x^k, provides the next solution $x^{k+1} = \sigma(x^k)$ by seeking it in the interior of $I(x^k)$. In minimization problems with objective function $c(x)$ we typically have:

$$\sigma(x^k) = \text{argmin}\{c(x): x \in I(x^k)\}$$

A general scheme of a local-search algorithm is the following.

```
Procedure Local_Search($F$,$c$,solution):
    begin
      $x$ := Feasible($F$);
      while $\sigma(x) \neq x$ do $x := \sigma(x)$;
      solution := $x$
    end.
```

We have $x=\sigma(x)$ when the algorithm recognizes the current solution as optimal x. This does not necessarily mean that the solution is really the optimal one but just that it satisfies a pre-set stopping criterion.
If the solution determined by the algorithm actually coincides with the optimal solution of the problem, we speak of *global optimum*, if not we have a *local optimum*. If the neighborhood used for a certain problem guarantees that the global optimum is always achieved, then the neighborhood is said to be *exact*.
Obviously, the greater the set $I(x)$, the higher the probability that the neighborhood is exact, but computing the $\sigma(x)$ will be more onerous as well. Pushing the reasoning to extremes, we could have $I(x)=F$, but $\sigma(x)$ would correspond to solving the original problem. On the other hand, a more bounded neighborhood probably ensures an efficient computation of the $\sigma(x)$, but can imply a convergence of the algorithm in a greater number of steps.
Let us look at the definition of a few neighborhoods, starting from the problems we encountered and for which we studied a local-search algorithm.

Example: maximum flow
Augmenting-path algorithms present the characteristic of moving from a solution x^i to a solution x^{i+1} varying the flow along a path of the residual graph. Hence, we can define:

$$I(x) = \{x' = x \oplus x_p: p \text{ path from } s \text{ to } t \text{ in } G_R(x)\},$$

where the operator $\oplus$ summarizes the operation of flow variation along path p, which implies a flow augmentation along the arcs of $A^+(x)$ in p and a diminution along those of $A^-(x)$ in p.
Algorithms differ according to the choice made by the $\sigma(x)$, which can be: any path (generic algorithm), a path with the lowest number of arcs (Edmonds-Karp algorithm), a path with capacity above a certain threshold (capacity scaling algorithm).

Example: minimum cost flow
The negative cycle removal algorithm has the features of a local-search algorithm. Unlike in the maximum flow problem, in this case the definition of neighborhood is slightly more delicate because the purpose is to preserve the feasibility of flow conservation constraints at nodes, which is possible if we perform variations by an equal amount of flow along cycles of the residual graph. Using the same notation as the preceding example, we can define:

$$I(x) = \{x' = x \oplus x_c: c \text{ cycle in } G_R(x)\}.$$

The choice made by the $\sigma(x)$ is to detect a negative cost cycle in $G_R(x)$. This choice can be further specialized; for example: the minimum cost cycle, the cycle with lowest product of cost and residual capacity, the cycle with lowest average cost (cycle cost divided by the number of arcs).

Example: Linear Programming
Also the Primal-Dual Simplex algorithm we examined for solving linear programming problems of the form max $\{cx: Ax \leq b\}$ has the features of a local-search algorithm. In fact, it starts from a feasible solution and moves towards best-oriented solutions. In the case of this algorithm the definition of neighborhood $I(x)$ of the current solution is more composite because, at least in the description

provided above, it decomposes into two phases: the search for the displacement direction and the determination of the displacement step along the given direction. Figure 26 provides a possible interpretation.

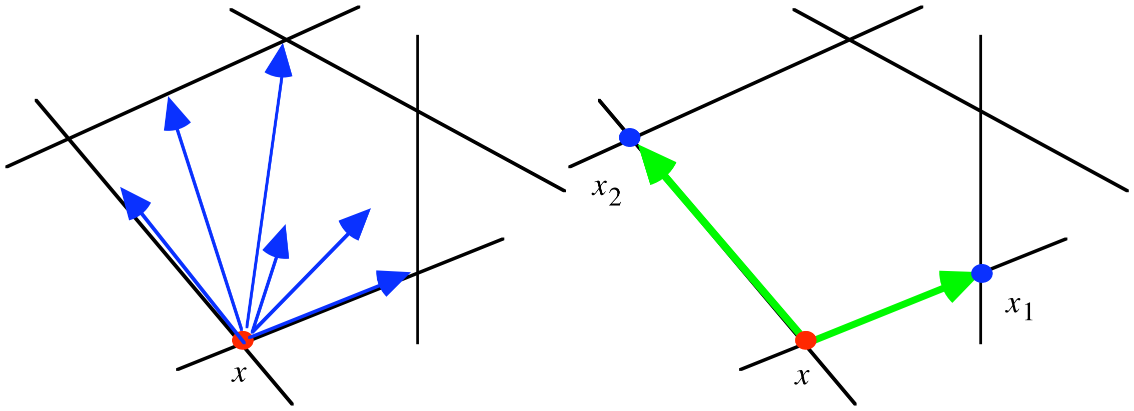

Fig. 26: neighborhood used by the Primal-Dual Simplex. Extended version and bounded version

Hence, we can formally say that:

$$I(x) = \{x' = x + \lambda\xi:\ A_I\xi \leq 0, Ax \leq b\}$$

where I, as usual, denotes the set of indexes of the constraints being active in x. If we apply the definition of neighborhood in an extended way, we see that from point x we can reach any point of the feasible region in just one step. This actually makes the original problem equivalent to the problem we solve to determine the best solution of the neighborhood by deriving the $\sigma(x)$. Therefore, we practically bound the neighborhood and, among all directions, the $\sigma(x)$ chooses one remaining perpendicular to all active constraints except one. Furthermore, the $\sigma(x)$ returns an extreme point obtained by moving along the direction until a new active constraint can be met, if in that direction the feasible region proves to be bounded. In this way, the number of solutions in the neighborhood is finite. In the case of the algorithm we presented in Chapter 3, the choice of the direction is made according to Bland's anticycle rule.

Example: SAT

As we saw, the satisfiability problem is a decisional problem, so it simply requires to determine if the feasible region (set of the assignments of truth values making the formula true) is empty or not. Wishing to execute a local-search algorithm, we realize that by determining a feasible starting solution we would solve the problem at the first step already. We can therefore turn the problem into an optimization problem by setting the number of not satisfied clauses as objective function to be minimized. It follows that now any assignment of truth values to logical variables is a feasible solution. If we have n variables, the feasible region consists of the vertices of the unit hypercube:

$$F \subseteq \{0,1\}^n$$

A possible neighborhood of a solution is defined by the subset of solutions differing from the given solution in a component, i.e., that are at a Hamming distance smaller than or equal to 1 in relation to the given solution:

$$I(x) = \{y \in F: \sum_i |x_i - y_i| \leq 1\}.$$

Figure 27 illustrates the case of $n = 3$, pointing out the set of solutions in the neighborhood of (1,0,1): $I(1,0,1) = \{(1,0,1), (1,0,0), (1,1,1), (0,0,1)\}$.

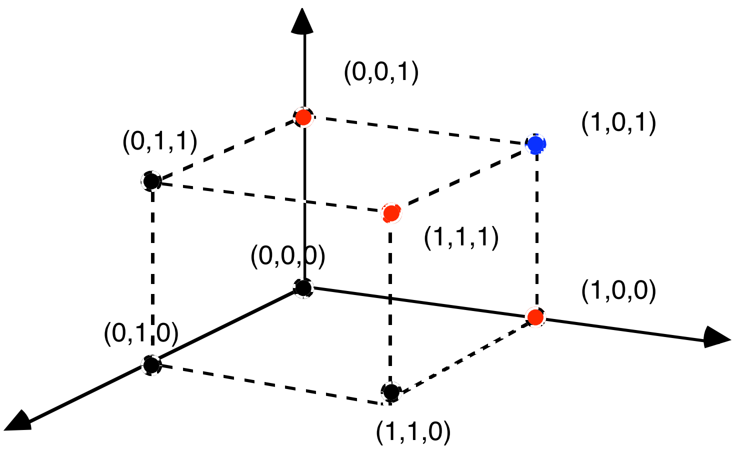

Fig. 27: neighborhood for the SAT problem

<u>Example: minimum weight spanning tree</u>
Consider the spanning tree problem defined on an undirected graph G=(N,A) with weights on arcs w_e≥0, for which in Chapter 2 we proved that the greedy algorithm ensures the optimality of the solution. Let us now try to elaborate a local-search algorithm for it. So, we have a spanning tree T – not necessarily optimal – and we wish to define a neighborhood $I(T)$. In order to define such neighborhood we draw inspiration from the properties of spanning trees. In particular, recall that if an arc $e \notin T$ is added to the tree, then a unique cycle $C(e)$ forms and in order to get a tree again we just have to remove any arc of the cycle $C(e)$. Thus, we can define a neighborhood in the following way:

$$I(T) = \{T' = T \cup \{e\} \backslash e' \ \forall e' \in C(e)\}$$

If an exchange between arc $e \in T$ and arc $e' \in C(e)$ is carried out, the weight of the new solution is given by:

$$w(T') = w(T) - w_e + w_{e'}$$

If we wish to implement the $\sigma(T)$ by detecting the best solution in the neighborhood, we must exhaustively explore all possible exchanges between arcs not belonging to T and arcs forming the cycle. The possible alternatives are therefore m-n-1 (number of arcs in $A \backslash T$) by n (the greatest number of arcs in the cycle). Since each evaluation of the weight of the new solution implies a constant number of operations, the complexity of the $\sigma(T)$ is $O(nm)$.

Theorem 5.1
$I(T)$ is an exact neighborhood.

<u>Proof</u>
The proof is based on criteria that are similar to those used to prove the optimality of Kruskal's algorithm. Let T be a local optimum and $T^* \neq T$ the minimum weight tree and suppose $w(T^*)<w(T)$. Since both trees have the same number of nodes, there must exist an arc $e \in T^* \backslash T$ for which $w(e)<w(e')$ for each $e' \in T$. If we add e to T a unique cycle C is formed in which the arc e has weight strictly smaller than the weight of the other arcs of the cycle. This contradicts the hypothesis that T is a local optimum and that there can exist a global optimum having smaller weight than the local optimum.
♦

Exercise
Define a neighborhood $I(T)$ based on the property relative to the removal of an arc from the tree and to the formation of two connected components. Prove that the neighborhood is exact in this case too.

<u>Example: the traveling salesperson problem</u>
In the case of the traveling salesperson problem, defined on an undirected complete graph G=(N,A) with costs c_{ij} on arcs, there are various neighborhoods. The simplest neighborhood is the one performing a minimum variation. We cannot exchange only one arc being in the solution for one not belonging to it, because in that case we would obtain the starting solution. In fact, if we remove an arc from the Hamiltonian circuit, there is just one way to reconstruct a feasible solution that coincides with the starting solution. Thus, the minimum variation includes the exchange of pairs of arcs being in the solution for arcs not being in the solution. Arcs cannot be chosen in whichever way; in fact, if the two

removed arcs proved to be adjacent, then, in this case too, the only way to obtain a feasible solution again would repropose the starting solution. Let H be the starting Hamiltonian circuit and (i,j) and $(h,k) \in H$ the two removed arcs. The new solution H' is obtained by exchanging (i,j) and (h,k) for (i,h) and (j,k), as shown in Figure 28.

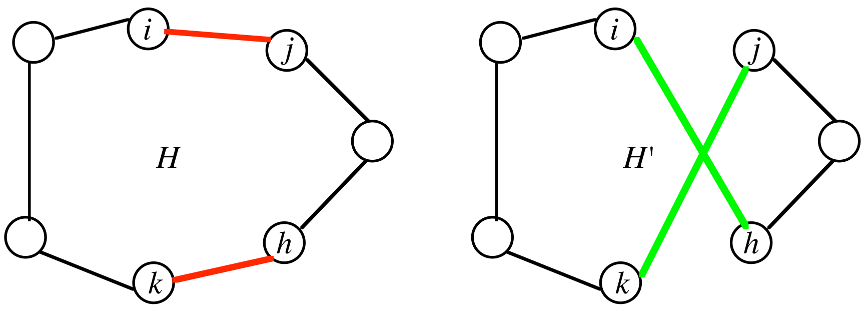

Fig. 28: variation of a Hamiltonian circuit

The neighborhood is formally defined as follows:

$$I(H) = \{H' = H \setminus \{(i,j),(h,k)\} \cup \{(i,h),(j,k)\},\ \forall (i,j),(h,k) \text{ non-adjacent arcs of } H\}.$$

Note that, strictly speaking, if $I(H)$ is defined in the above way it does not contain H itself, thus contravening the rule i) of the definition of neighborhood. This difficulty can be overcome by adding H to the set $I(H)$. The number of solutions in a neighborhood can be easily determined: it suffices to enumerate the ways of choosing two pairs of non-adjacent arcs in a Hamiltonian circuit of n arcs amounting to $n(n-3)/2$. The determination of the best solution of the neighborhood, which is needed to evaluate the $\sigma(H)$, has therefore complexity $O(n^2)$, since the cost of each solution H' of the neighborhood is evaluated by difference in relation to the cost of the reference solution H and implies a constant number of operations:

$$c(H') = C(H) - c_{ij} - c_{hk} + c_{ih} + c_{jk}.$$

Observe that, if we considered a directed version of the traveling salesperson problem with non-symmetric arc costs ($c_{ij} \neq c_{hk}$), then the new Hamiltonian cycle H' – obtained according to the definition of neighborhood $I(H)$ given above – would not respect the same direction of cycle H in all original parts. This means that, in practice, each exchange of pairs of arcs (i,j) and (h,k) corresponds to a pair of Hamiltonian (directed) cycles: the one using the directed arcs (i,h) and (j,k) traversing the left-hand portion of the cycle in the same direction as the starting cycle, whereas the right-hand portion is traversed in the opposite direction (refer to Figure 28); or the one using the directed arcs (k,j) and (h,i) traversing the left-hand section in the opposite direction and the right-hand section in the same direction as the original cycle. This means that for each exchange of arcs we cannot proceed by difference but must evaluate the cost of the new cycle by adding together the costs of the arcs composing it. Hence, the complexity of the $\sigma(x)$ becomes $O(n^3)$.

Exercise

Propose a neighborhood preserving the direction of the cycle portions that remain unchanged. Evaluate the complexity of the function $\sigma(x)$.

<u>Example: the knapsack problem</u>

For some problems the definition of local-search algorithms can be hardly systematized. Let us take, for instance, the knapsack problem $\max\{cx: ax \leq b, x_i \in \{0,1\}, i=1,\ldots,n\}$ and let us consider the solution x defined by the choice of the subset of objects $S \subseteq \{1,\ldots,n\}$, that is $x_i=1$ for each $i \in S$. A neighborhood of solution S could be defined by removing from S a subset of objects K and replacing it with another subset of objects H utilizing the capacity made available in the knapsack. We formally have:

$$I(S) = \{S' = S \setminus K \cup H: K \subseteq S, H \subseteq \{1,\ldots,n\} \setminus S \setminus K\}$$

Note that the definition of $I(S)$ is extremely general and the set K might coincide with the whole S. In that case a function $\sigma(S)$ searching for the best solution of the neighborhood coincides with the original problem and the local-search paradigm is of no help. Then we can bound the cardinality of set K in order to reduce the degrees of freedom of the $\sigma(S)$. Hence, let k be the maximum cardinality allowed for the set of removed objects K. In order to know how to best replace the removed objects, the following problem should be solved:

$$\begin{aligned} P(S)\text{: } \max & \sum_{i\in O} c_i z_i \\ & \sum_{i\in O} a_i z_i \le b - \sum_{i\in S\setminus K} a_i \\ & z_i \in \{0,1\} \qquad \forall i\in O \end{aligned}$$

where, for the sake of brevity, we denoted by O the set of available objects, i.e., $O=\{1,\ldots,n\}\setminus S\setminus K$. The determination of $\sigma(S)$ implies therefore the solution of a knapsack problem, though on a reduced number of objects and with smaller capacity compared with the original one. This means that the instance of the knapsack problem could be reduced enough to be optimally solved using the algorithm based on the search for a longest path on an acyclic graph that we saw in Chapter 2. Alternatively, if the instance is large, and knowing that it must be solved for each possible set K, we might envisage not solving the subproblem optimally by making use of an algorithm of the greedy type. Consequently, we would have a heuristic evaluation of the $\sigma(S)$.

Exercise

Define a neighborhood based on the idea of adding a set of objects to the current solution S while removing other objects, so as not to exceed the maximum capacity of the knapsack. Specify the problem that must be solved to evaluate the $\sigma(S)$.

If a local-search algorithm quickly converges on a local optimum, there might remain time available to explore other zones of the feasible region and possibly to generate better solutions. This could be done using a *restart* procedure that generates a new feasible solution (by means of a GRASP or a random generation procedure) and applying the local search again. Other algorithms that try to exceed local optima are the *Simulated Annealing* and the *Tabu Search*, which can be viewed as variants of the local search.
The Simulated Annealing tries to reproduce the process of crystallization of chemical compounds that, if cooled very slowly, converge on minimum energy configurations. Instead of exhaustively exploring the neighborhood, we choose a random solution in the neighborhood. If the picked out solution is better than the current one, the variation is accepted; if the new solution is worse than the current one, the variation is accepted according to a probability law that depends on the "temperature" (i.e., a parameter that decreases as the number of iterations grows) and on the size of the worsening.
The Tabu Search, in its simplest version, is totally deterministic. It is a sort of local search in which at each iteration the $\sigma(x)$ returns the best solution of the neighborhood. Such solution is always accepted, even when it involves a worsening. As this choice would easily induce cycling within few solutions, we retain a list of *Tabu* solutions that are removed from the neighborhood in order to prevent endless cycles. The practical implementation of the Tabu Search implies a series of peculiarities that may be rather sophisticated.
For further details on variants of local-search algorithms we refer the reader to the book....