Graphs and Network Flows

1 Models on graphs and network flows

An extensive class of problems having a considerable practical importance can be modeled by using graphs or network flows. In this chapter we will develop models with the help of such tools and, moreover, we will analyze the algorithms used for solving the different problems we will encounter. A graph *G* is generally defined by a set of nodes *N* and a set of arcs *A*. An arc is defined by a pair of nodes. Let us review a few problems admitting an easy representation through graphs.

Telecommunications network design
The university computing center is to be decentralized to several buildings across the campus; a computer will be installed at each different center. The installation cost of a data transmission link between two centers depends on the distance to be covered. Wishing to guarantee the data transfer between the different machines, the systems analysts have to plan the communications network trying to curb spending as much as possible.

Assignment of exercises to students
The exercises prepared for the Operations Research examination are as many as the candidates. For each exercise, the degree of difficulty perceived by each student is known. The purpose is to assign a different exercise to every different student, in such a way that no two students are assigned the same exercise and the maximum degree of difficulty is minimized (or the minimum degree of difficulty is maximized!).

The postman problem
A postman must deliver the mail to the inhabitants of houses placed on both sides of a given set of streets. In order to carry out the delivery, the postman starts from the post office, covers each street at least once, then returns to the starting point. The problem is to find the sequence of streets (with possible repetitions) the postman has to cover in order to make the total route as short as possible.

Route of the Giro d'Italia
The mayors of several Italian towns offer a certain amount of money to the organizers of the Giro d'Italia in order to have a stage arrival of the Giro in their town; a smaller amount of money is offered if the Giro just crosses a town. The rules according to which the Giro route is defined imply that both departure and arrival must take place in Milan and that the riders, for reasons of public order, may not pass through a town which is a Giro stage more than once. The organizers of the Giro have to determine departure and arrival of each stage as well as the route of each stage in such a way that no stage is longer than 200 km and the global amount of prizes is maximized.

Car renewal
A traveling salesperson has to plan the purchase and/or the possible replacement of the car he/she needs for his/her job during the next 5 years. Actually, at the beginning of every year he/she has to decide whether to replace the old car with a new one or to face the coming year and the upkeep costs with the old car.

2 Notation and definitions

A graph G is generally defined by a set of nodes N and a set of arcs A. An arc is defined by a pair of nodes that are called the *endpoints* of the arc.
If the pairs of nodes defining the arcs are undirected, the graph is said to be *undirected*. Figure 1 shows the representation of an undirected graph $G=(N,A)$, $N=\{1,2,3,4,5\}$, $A=\{\{1,2\}, \{1,4\}, \{1,5\}, \{2,3\}, \{2,4\}, \{3,4\}, \{4,5\}\}$.

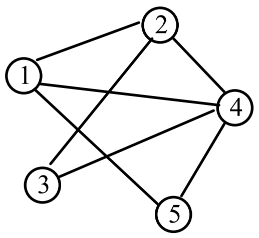

Fig. 1: undirected graph

Otherwise, if the pairs of nodes defining the arcs are directed, the graph is called a *directed graph*, or *digraph*. Figure 2 shows the representation of a directed graph $G=(N,A)$, $N=\{1,2,3,4,5\}$, $A=\{(1,2), (1,4), (1,5), (2,3), (2,4), (4,2), (4,3), (4,5), (5,4)\}$.

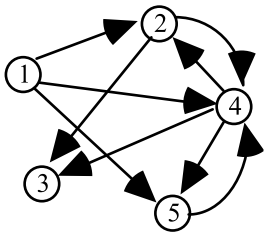

Fig. 2: directed graph

Consider a directed arc $a=(i,j)$; node i is called *tail of arc a*, whereas node j is called *head of arc a*.
A graph is *simple* if it has no multiple arcs (i.e., arcs having all the same head and the same tail) and no self-loops (i.e., arcs with head identical to tail); in particular, a digraph $G=(N,A)$ is simple if for each pair of arcs of A, $a=(i,j)$ and $a'=(h,k)$, $i\neq h$ or $j\neq k$; further $i\neq j$, and $h\neq k$.
Now let us introduce some definitions that refer to directed graphs but, wherever the orientation is unimportant, can also be extended to the case of undirected graphs.
Consider a digraph $G=(N,A)$:
Set of successors:

$$FN(i) = \{j\in N: (i,j)\in A\}.$$

Set of predecessors:

$$PN(j) = \{i\in N: (i,j)\in A\}.$$

Example: in Figure 2 $FN(1)=\{2,4,5\}$, $FN(2)=\{3,4\}$, $FN(3)=\emptyset$, $FN(4)=\{2,3,5\}$, $FN(5)=\{4\}$; $PN(1)=\emptyset$, $PN(2)=\{1,4\}$, $PN(3)=\{2,4\}$, $PN(4)=\{1,2,5\}$, $PN(5)=\{1,4\}$.
Set of adjacent nodes:

$$AD(i) = FN(i) \cup PN(i).$$

Forward star:

$$FS(i) = \{(i,j) \in A\}.$$

Backward star:

$$BS(j) = \{(i,j) \in A\}.$$

Star:

$$S(i) = FS(i) \cup BS(i).$$

Example: in Figure 2 $FS(1)=\{(1,2),(1,4),(1,5)\}$, $FS(2)=\{(2,3),(2,4)\}$, $FS(3)=\emptyset$, $FS(4)=\{(4,2),(4,3),(4,5)\}$, $FS(5)=\{(5,4)\}$; $BS(1)=\emptyset$, $BS(2)=\{(1,2),(4,2)\}$, $BS(3)=\{(2,3),(4,3)\}$, $BS(4)=\{(1,4),(2,4),(5,4)\}$, $BS(5)=\{(1,5),(4,5)\}$.

The *out-degree* of a node i is given by the cardinality of its forward star (or, equivalently, of the set of successors), whereas the *in-degree* is given by the cardinality of the backward star (or of the set of predecessors). The *degree* is given by the sum of out-degree and in-degree.
The *subgraph of G induced* by $V \subseteq N$ is graph $G' = (V,A')$, where the set of arcs $A'=\{(i,j) \in A: i, j \in V\}$. Figure 3a shows the subgraph of G induced by $V=\{1,3,4,5\}$. The *partial graph of G defined by $B \subseteq A$* is graph $G' = (N,B)$. Figure 3b shows the partial graph defined by $B=\{(1,2),(1,5),(2,4),(4,2),(4,5),(5,4)\}$. The two definitions can be combined. Figure 3c shows the partial subgraph of G induced by V defined by B.

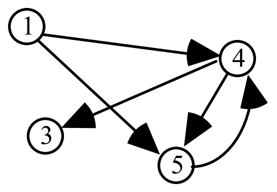

Fig. 3a

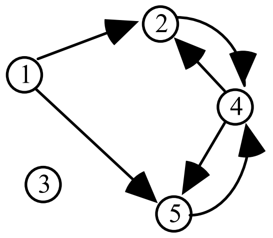

Fig. 3b

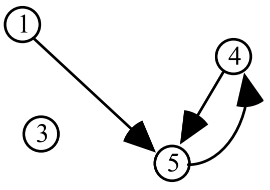

Fig. 3c

Now let us define graphs having particular structures and introduce some important and basic concepts.
Chain: a chain in a digraph $G=(N,A)$ is a partial subgraph of G defined by $B=\{a_1,a_2,a_{r-1}\}$ and represented as a finite sequence of adjacent nodes and arcs, i_1, a_1, $i_2,\ldots$, i_{r-1}, a_{r-1}, i_r such that either $a_k=(i_k,i_{k+1})$ or $a_k=(i_{k+1},i_k)$, for $k=1,\ldots,r-1$. When an orientation is established from i_1 to i_r, arcs of type (i_k,i_{k+1}) are said to be *in accordance with the orientation* and arcs of type (i_{k+1},i_k) are said to be *not in accordance with the orientation*. For more simplicity, from now on we will indicate a chain as a sequence of arcs and nodes.
Closed chain: it is a chain in which $i_1=i_r$. A closed chain is also called a *circuit*.
Example: for the graph of Figure 2 a chain is given by (1,2),(4,2),(4,5),(1,5),(1,2). A closed chain is given by (1,2),(4,2),(1,4).

Path: a path in a digraph $G=(N,A)$ is a partial graph of G defined by $B=\{a_1,a_2,a_{r-1}\}$ represented as a sequence of adjacent nodes and arcs, i_1, a_1, $i_2,\ldots$, i_{r-1}, a_{r-1}, i_r such that $a_k=(i_k,i_{k+1})$ for $k=1,\ldots,r-1$. For more simplicity, from now on we will indicate a path as a sequence of nodes or as a sequence of arcs. *Closed path*: It is a path in which $i_1=i_r$.

<u>Example:</u> for the graph of Figure 2 a path is given by 2,4,5,4,3. A closed path is given by 2,4,5,4,2.

Simple path: a simple path is made of a path having no repeated arcs. *Elementary path*: an elementary path is made of a path having no repeated nodes.

<u>Example</u>: for the graph of Figure 2 a simple path is given by 2,4,5,4,3. An elementary path is given by 1,2,4,5.

Cycle: it is a closed path in which the only repeated node is the first one. In other words, a cycle is made of an elementary path from i_1 to i_r and of the arc (i_r, i_1). A graph which does not contain cycles is said to be *acyclic*.

<u>Example</u>: for the graph of Figure 2 a cycle is given by 2,4,2.

Hamiltonian cycle: the Hamiltonian cycle of a digraph $G=(N,A)$ is a cycle whose nodes are all nodes in N, thus exactly containing $|N|$ arcs. Note that not always there exists a Hamiltonian cycle in a graph; for example, in the graph of Figure 2 it is not possible to detect any Hamiltonian cycle.

Eulerian cycle: the Eulerian cycle of a digraph $G=(N,A)$ is a closed simple path exactly having $|A|$ arcs. Just like in the case of the Hamiltonian cycle, not all graphs admit an Eulerian cycle. A necessary and sufficient condition for the existence of an Eulerian cycle is given by: $|FS(i)|=|BS(i)|$, $\forall i\in N$.

Connected graph: a digraph $G=(N,A)$ is connected if, for each pair of nodes i and $j\in N$, there exists a chain from i to j.

Strongly connected graph: a digraph $G=(N,A)$ is strongly connected if, for each pair of nodes i and $j\in N$, there exists a path from i to j.

<u>Example</u>: the graph of Figure 2 is connected but not strongly connected; it clearly appears, for instance, that there is no path starting from node 3 (in fact $FS(3)=\emptyset$), further there is no path arriving at node 1 (in fact $BS(1)=\emptyset$).

Connected component: a connected component of a digraph $G=(N,A)$ is a maximal connected subgraph $G'=(N',A')$, i.e., such that for each $i\in N\setminus N'$ there exists neither the arc (i,j) nor the arc (j,i) with $j\in N'$. Similarly we define a *strongly connected component*.

<u>Example</u>: in Figure 3c it is possible to detect 2 connected components {3} and {1,4,5}. In Figure 2 there are 3 strongly connected components: {1}, {3} and {2,4,5}.

Cut: a graph's cut is defined by a partition of the set of nodes N into two subgraphs N' and N'' with $N=N'\cup N''$ and $N'\cap N''=\emptyset$. The *arcs of the cut* (N',N'') are defined by the set of arcs with tail in N' and head in N'' (arcs in accordance with the cut) and by the set of arcs with tail in N'' and head in N' (arcs not in accordance with the cut). Removing the arcs of the cut, an initially connected graph is partitioned into at least two connected components.

<u>Example</u>: a cut of the graph of Figure 2 is given by $N'=\{1,4\}$ and $N''=\{2,3,5\}$; the arcs of the cut are (1,2), (1,5), (2,4), (4,2), (4,3), (4,5), (5,4).

Bipartite graph: a graph is bipartite if the set of nodes N can be partitioned into two subsets N_1 and N_2 such that for each arc (i,j) of A either $i \in N_1$ and $j \in N_2$ or $i \in N_2$ and $j \in N_1$.
Tree: a tree is a connected graph which does not contain closed chains.
Spanning tree of a graph: a tree T is a spanning tree for graph G if all nodes of G are head or tail of arcs of T.
Example: a spanning tree for the graph of Figure 2 is given by {(1,4), (2,4), (4,3),(5,4)}.
Matching: a matching is a partial graph such that no two arcs are incident on the same node. When each node has one incident arc, the matching is called a *perfect matching.*

Let us shortly reconsider the problems mentioned above, trying to interpret them in terms of graphs and find out what characteristics the solutions must possess.

In the problem concerning a telecommunications network design we can use a graph in which nodes represent the centers to be connected and arcs represent the communication lines that can be activated. The graph is undirected and, since we have to guarantee the connection between all centers, a solution is given by a spanning tree; in particular, we have to seek the one with minimum cost. But if the design must take into consideration the network's robustness in case of breakdowns as well, we soon realize that a tree has a defect: in case of breaking of a communication line, the network appears to be unconnected. A possible way of dealing with this problem is to require the network biconnection, that is between each pair of nodes there must exist two chains disjoint on nodes and arcs. It is quite evident that, given such requirements, what we are looking for is a ring connecting all nodes, i.e., a Hamiltonian circuit.

In the case of exercise assignment to students we can use a bipartite graph in which nodes are partitioned into two subsets: one subset representing students, the other representing exercises. Arcs go from students-subset nodes to exercises-subset nodes; the weights associated with them is that of the perceived difficulty. The problem consists in finding a perfect matching in which the arc having maximum weight is minimized (bottleneck objective function).

In the postman case we can use a graph in which nodes represent crossroads and arcs represent streets. For more simplicity, we assume that the postman delivers the mail at the same time on the even-numbered side and on the odd-numbered side of the street, hence the graph is undirected. If the graph thus obtained is Eulerian, almost nothing can be decided; in fact it is only possible to traverse all arcs once and then return to the starting point. We leave behind as a possible exercise the detection of the Eulerian circuit. Otherwise, if the graph is not Eulerian – hence there exist odd-degree nodes (but why is the number of such nodes even?) – then it is necessary to determine which streets the postman has to cover more than once in order to be able to carry out his task, obviously minimizing the route length. Later we shall see how this problem can be solved by combining different algorithms on graphs.

In the Giro route problem, assuming that optimization criteria are actually adopted, we can represent the road network with a graph: nodes are towns candidate for being intermediate winning lines or stages and arcs are roads connecting the different places being suitable for the passage of the Giro caravan. Unlike the postman problem, in this case the graph should be directed in order to take into consideration the upward or downward direction in which roads can be run. Since start and arrival are to be in the same place, we have to look for a cycle on the graph being not necessarily Hamiltonian, but nevertheless provided with additional constraints referring to stage length, meters of climbing, etc. On the other hand, the objective function has to consider the prizes totalized when passing through the different places. Since additional constraints are frequently not strictly binding – and taking them into account might be complicated – it is reasonable to consider them as secondary objectives and include them in a unique utility function.

In the car renewal case it might surprise to use a graph in order to determine the optimal solution. Actually, we can use a graph in which nodes represent the beginning of the different years of planning – that is when the decision about the car renewal must be made – and arcs represent the decisions as such. So we obtain a directed arc which directly connects node *i* to node *i*+*k* if it is decided to buy the car in the year *i* and resell it after *k* years. This choice implies an exactly quantifiable cost: the purchase cost in the year *i*, the upkeep costs in the *k* following years, possibly growing with the car aging, from which we subtract the proceeds of the car sale after *k* years of usage. Therefore, in order to solve the problem we have to detect a car renewal strategy ensuring that exactly one car is available all the time and minimizing the cost during the planning period of time. As we shall see, the optimal solution of this problem consists in a shortest path from the node representing the planning beginning to the node representing the end of the traveling salesperson's activities.

2.1 *Representation of graphs*

To represent graphs – apart from the graphical representation of nodes and arcs – we can adopt a variety of techniques depending on the type of use. Some of them are more suited to a mathematical type of use, others to an algorithmic use.

Adjacency matrix

We make use of a square matrix *AD* with a row/column corresponding to each node of graph $G=(N,A)$. The matrix coefficient ad_{ij} is equal to 1 if and only if arc $(i,j)\in A$. If the graph is undirected the matrix *AD* is symmetric, or else it could only be memorized in the triangular upper or lower part.

For the digraph of Figure 2 we have the following matrix:

0	1	0	1	1
0	0	1	1	0
0	0	0	0	0

0	1	1	0	1
0	0	0	1	0

Note that the non-zero elements of each row detect the corresponding forward star, whereas those of the columns correspond to the backward stars.

Node-arc incidence matrix

Given a digraph $G=(N,A)$, the node-arc incidence matrix E has a row for each node and a column for each arc of the graph and is built in the following way:

$$e_{ia} = \begin{cases} -1 & \text{if } a=(i,j)\in A \\ 1 & \text{if } a=(j,i) \\ 0 & \text{else} \end{cases}$$

The incidence matrix is particularly useful in mathematical formulations.

The incidence matrix of the graph of Figure 2 is given by:

	(1,2)	(1,4)	(1,5)	(2,3)	(2,4)	(4,2)	(4,3)	(4,5)	(5,4)
1	-1	-1	-1	0	0	0	0	0	0
2	1	0	0	-1	-1	1	0	0	0
3	0	0	0	1	0	0	1	0	0
4	0	1	0	0	1	-1	-1	-1	1
5	0	0	1	0	0	0	0	1	-1

Lists

Matricial representations are not particularly adequate to efficient implementations of algorithms, especially if graphs are sparse. In this case we have recourse to a representation of backward stars and forward stars by means of lists. Each arc is thus represented by a record containing the following fields: pointer to arc attributes (cost, capacity, weight, flow, etc.), pointer to head node, pointer to tail node, pointer to next arc in the forward star of tail node, pointer to next node in the backward star of head node. If we intend to alter the degree by adding or canceling arcs, then it might be profitable to provide double lists, further to place also the pointers at the predecessor in the backward and the forward star.

3 Undirected graphs: spanning tree and Hamiltonian circuit

Let us examine the problem of the telecommunications network design in detail, which – as we saw – can be represented by means of a weighted undirected graph $G=(N,A)$ in which nodes correspond to the centers to be connected and arcs to the communication lines that can be installed. The weight $w_e > 0$ associated with each arc $e=(i,j)$ is the cost implied by installing the line between the centers and the endpoints of the arc. Figure 4 shows a graph with 7 nodes and 12 arcs.

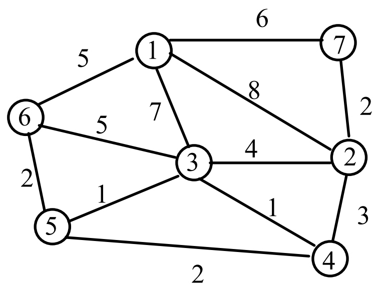

Fig. 4: weighted undirected graph

3.1 Spanning tree: formulation and a solution algorithm

Remember that in the case of a spanning tree the solution requirements imply that between each pair of nodes there must exist a (unique) chain. Before we continue describing a possible solution algorithm, let us try to formulate the problem in terms of variables, objective function and constraints. In order to come to a representation of a solution *T*, it should be noted that a tree is composed of arcs, so we can introduce logical variables associated with arcs having the following meaning:

$$x_e = \begin{cases} 1 & \text{if } e \in T \\ 0 & \text{else} \end{cases}$$

for each arc $e \in A$. The objective function can be easily expressed:

$$\min \sum_{e \in A} w_e x_e$$

Expressing connection constraints mathematically is much more interesting than one can do with words. Such constraints are based on the following observation: in a spanning tree, as well as in any connected subgraph, if we consider a proper subset of nodes $S \subset N$ containing at least one node, it must be possible to establish a connection with the rest of the graph by at least one arc. So the resulting constraints are:

$$\sum_{e \in \delta(S)} x_e \geq 1 \qquad \forall S \subset N, |S| \geq 1,$$

where with $\delta(S)$ we denote the set of arcs having one endpoint in S and the other in $N \backslash S$: $\delta(S) = \{(i,j) \in A: i \in S, j \in N \backslash S\}$. Summing up, the formulation of the spanning tree problem is the following:

$$\begin{aligned} \min & \sum_{e \in A} w_e x_e \\ & \sum_{e \in \delta(S)} x_e \geq 1 \qquad \forall S \subset N, |S| \geq 1, \\ & x_e \in \{0,1\} \qquad \forall\, e \in A. \end{aligned}$$

Note how the above formulation contains no explicit constraint concerning the uniqueness of the chain between each pair of nodes. In fact, such constraint is implied by the objective function and by the fact that arc weights are positive. A further observation concerns the model dimension: the number of variables is equal to the number of arcs, but the number of constraints is equal to the number of possible ways of choosing a subset of nodes, thus it is exponential in the number of nodes. Using the tools we have seen so far, such formulation appears rather useless. Nonetheless, it is possible to describe an algorithm which is extremely efficient to determine the optimal solution of the problem.

Exercise
The spanning tree model described above does not provide a feasible solution in case of weights not restricted to positive values. Why? How can we complete the model so as to make it correct in this more general case as well?

The algorithm we are considering iteratively builds a solution starting from the empty one, and at each new step adds a new arc to the set of those that had been chosen. Two criteria must be established to guarantee the correct functioning of the algorithm: i) the order according to which we should select the arcs to be put in the solution; ii) the criterion according to which we decide to add an arc to those already existing.

Concerning the order, we are guided by arc weights. So we begin by considering the arcs with lighter weight and end with the heavier ones. An arc is added to the current solution only if the set of arcs, so augmented, can be part of a feasible solution. In the considered case this means that the arc added to the solution must not form cycles with the chosen arcs. Otherwise, we would come to a redundant solution. In conclusion, the algorithm, known in the literature as Kruskal's algorithm, is the following:

```
Procedure Kruskal ($G=(N,A)$, $T$):
    begin
      $T := \emptyset$;  $S := A$;
      repeat
        $e :=$ argmin$\{w_a, a \in S\}$;  $S := S \setminus \{e\}$;
        if $T \cup \{e\}$ does not contain cycles then $T := T \cup \{e\}$
      until $S = \emptyset$
    end.
```

During the algorithm execution, *S* contains the set of arcs candidate for entering the solution and which were not examined yet; at the end *T* contains the solution. The function **argmin**$\{w_a, a \in S\}$ restores the element with minimum weight among those in *S*. In the following table we describe the evolution of the algorithm applied to the graph of Figure 4.

e	*S*
(3,4)	{(3,4)}
(3,5)	{(3,4), (3,5)}
(4,5)	{(3,4), (3,5)}
(2,7)	{(3,4), (3,5), (2,7)}
(5,6)	{(3,4), (3,5), (2,7), (5,6)}
(2,4)	{(3,4), (3,5), (2,7), (5,6), (2,4)}
(2,3)	{(3,4), (3,5), (2,7), (5,6), (2,4)}
(1,6)	{(3,4), (3,5), (2,7), (5,6), (2,4), (1,6)}

(3,6)	{(3,4), (3,5), (2,7), (5,6), (2,4), (1,6)}
(1,7)	{(3,4), (3,5), (2,7), (5,6), (2,4), (1,6)}
(1,3)	{(3,4), (3,5), (2,7), (5,6), (2,4), (1,6)}
(1,2)	{(3,4), (3,5), (2,7), (5,6), (2,4), (1,6)}

Table 1: evolution of Kruskal's algorithm applied to the graph of Figure 4

Note that the algorithm might stop before having considered all arcs. Actually, in our example the last four iterations are useless because we already completed the spanning tree by adding the arc (1,6). The algorithm can be sped up by modifying the stopping condition: in fact, if we have a graph *G* connected to *n* nodes, then we know that a spanning tree contains exactly *n*-1 arcs. Hence, the stopping condition becomes:

```
    until $S = \emptyset$ or $|T|=|N|-1$
```

Theorem

Kruskal's algorithm provides the optimal solution of the spanning tree problem.

Proof

The demonstration is by contradiction. Let T^* be the minimum weight spanning tree, further suppose that Kruskal's algorithm provides a tree T different from T^* and with a total weight greater than the optimal one. This means that there exists an arc e of the graph belonging to T^* but not to T. By adding arc e to tree T a cycle C is formed. According to the functioning of Kruskal's algorithm $w_e \geq w_{e'}$ for each $e' \in C$. If now we remove arc e from T^* we obtain two connected components, and at least one arc $e' \in C$ and $\notin T^*$ belongs to the cut which has formed. If now we replace arc e by arc e' in the solution T^* we obtain a spanning tree with a total weight smaller than or equal to the one of T^*, thus contradicting the hypothesis.

♦

Exercise

Find an appropriate data structure as well as tricks enabling to efficiently implement Kruskal's algorithm. Special attention deserves the method by which cycles are found.

Exercise

Consider the problem of memorizing a set of records in a compact form, no matter if film frames or genetic codes. Instead of entirely memorizing each record, it is possible just to register the differences between one record and the other. Formulate the problem of storing the data minimizing the required memory in terms of minimum cost spanning tree.

3.2 Minimum cost Hamiltonian circuit

The problem of the minimum cost Hamiltonian circuit is one of the most studied combinatorial optimization problems and, not without reason, it has several applications. In the literature it is also known as Traveling Salesperson Problem - TSP. (For historical information and other curiosities, look at http://…….). The problem is specified by an undirected graph $G=(N,A)$ which is weighted like in the spanning tree case. A frequent hypothesis is also that the graph is *complete*, i.e., between each pair of nodes there exists the arc connecting them. It is always possible to refer to this case by adding to the graph the missing arcs and assigning them a very heavy weight. A simple application has already been mentioned in relation to the telecommunications network design: wishing to make a solution stronger with respect to breakdowns, we impose that there must always exist two disjoint chains between each pair of nodes. The simplest solution having this characteristic is a Hamiltonian circuit. Note that, if compared with the spanning tree, the Hamiltonian circuit has only one more arc. A possible formulation takes its starting point precisely from the spanning tree problem formulation. So, calling *C* a feasible solution (Hamiltonian circuit), we introduce the following variables:

$$x_e = \begin{cases} 1 & \text{if } e \in C \\ 0 & \text{else} \end{cases}$$

It must be noted that in a Hamiltonian circuit there are exactly two arcs incident on each node. So, an initial constraint might be:

$$\sum_{e \in S(i)} x_e = 2 \qquad \forall\, i \in N.$$

These constraints are not sufficient because they can be satisfied not only by Hamiltonian circuits, but also by all solutions visiting all graph nodes with exactly one cycle (not necessarily Hamiltonian); such solutions are also called *cycle cover*. For example, in the graph of Figure 4 a solution of this kind is: $C_1=\{(1,2),(2,7),(1,7)\}$, $C_2=\{(3,6),(3,4),(4,5),(5,6)\}$. It is thus necessary to add constraints in order to restrict the set of feasible solutions to cycle covers formed by a unique cycle. This is attained by using the constraints already seen for the spanning tree, which imposed the connection of the set of obtained arcs.

$$\sum_{e \in \delta(S)} x_e \geq 1 \qquad \forall S \subset N,\ |S| \geq 1,$$

Hence, a possible formulation of the Traveling Salesperson Problem is:

$$\begin{aligned} \min \sum_{e \in A} w_e x_e & \\ \sum_{e \in S(i)} x_e = 2 & \qquad \forall\, i \in N \\ \sum_{e \in \delta(S)} x_e \geq 1 & \qquad \forall S \subset N,\ |S| \geq 1, \\ x_e \in \{0,1\} & \qquad \forall\, e \in A. \end{aligned}$$

An alternative formulation of the problem takes into consideration another way of avoiding cycles not containing all nodes. We can impose a constraint that, considered any proper subset of *N* containing at least two nodes, can avoid the existence of a *subcircuit*. This means that the number of arcs connecting the subset nodes must be smaller than or equal to the cardinality of the subset minus 1. Hence, the formulation is:

$$\min \sum_{e\in A} w_e x_e$$
$$\sum_{e\in S(i)} x_e=2 \qquad \forall\, i \in N$$
$$\sum_{e=(i,j):\, i,j\in S} x_e \le |S| -1 \quad \forall S\subset N,\ |S|\ge 2,$$
$$x_e \in \{0,1\} \qquad \forall\, e \in A.$$

Like in the case of the spanning tree problem, the formulations we described contain an exponential number of constraints. Unfortunately, unlike the previous problem, the Traveling Salesperson Problem is not equally easy to be solved, as we shall study more deeply later on.

4 Graph connection: search algorithm

Consider a digraph $G=(N,A)$ with s and $t \in N$ and pose the problem of finding whether there exists a path connecting s to t. If such path exists, it is obviously easy to provide a "synthetic justification" (*certificate*) of its existence: in fact, we just have to describe the elementary path connecting s to t. Conversely, if such a path does not exist, it is not possible to justify the answer in a synthetic way while remaining in the space of paths; actually, we might enumerate all paths going out from s, but their number could prove to be exponential. The following theorem provides some help; it points out the relation existing between a graph's space of paths and space of cuts.

Property: path or cut

Let there be a digraph $G=(N,A)$ with s and $t \in N$. Only one of the following statements is valid:

i) there exists a path directed from s to t;

ii) there exists a cut (N_s, N_t) such that $s \in N_s$ and $t\in N_t$, and there exist no arcs being in accordance with the cut, that is to say all arcs $(i,j)\in A$ crossing the cut have $i\in N_t$ and $j\in N_s$.

<u>Proof</u>

The demonstration is constructive and is based on the possibility to search the graph starting from s. Let us proceed searching the graph starting from s and traversing the arcs according to their orientation. There appear two cases being mutually exclusive:

i) during the search for sequences of traversable arcs it is possible to reach node t , thus falling back into the case i) of the proposition: the path from s to t can be traced by going up the predecessors;

ii) the graph's search starting from s stops without having defined the predecessor of node t. We call N_t all nodes having a null predecessor, while we let N_s be the set of all nodes having predecessor other than zero. Obviously $N=N_s\cup N_t$, and $N_s\cap N_t=\emptyset$. According to the functioning of the algorithm, there cannot exist arcs (i,j) with $i\in N_s$ and $j\in N_t$, otherwise j would belong to N_s. ♦

A simple but extremely important algorithm is the one performing the search of digraph G starting from a root node s. The algorithm restores a vector of predecessors. The predecessor $P[i]$ of a node i provides the immediately preceding node i in the path directed from s to i. If at the end of the algorithm a node has predecessor equal to zero, this means that the node is not reachable from r.

```
Procedure Search (G=(N,A),s,P):
    begin
      for i := 1 to n do P[i] :=0; P[s] := s; Q := {s};      {initialization}
      repeat
        select i from Q; Q:=Q\{i};      {selection and removal of a node from Q}
        for each (i,j) ∈ FS(i)  do
          if P[j] = 0 then begin P[j] := i;  Q := Q ∪ {j} end;
      until Q = ø
    end.
```

Let us analyze the algorithm's complexity: denoting by n the number of nodes and by m the number of arcs of the graph, we notice that the search procedure examines each node at most once, and that for each node it examines the forward star able to contain at most n arcs. From this first analysis it emerges that the algorithm performs at most n selections from set Q and at most $O(n^2)$ controls and updates of the predecessor. But if we go more into details and globally estimate what is being developed by the cycle, we notice that each arc of the graph is examined at most once. In fact, an arc belongs to only one forward star. This means that the algorithm performs at most n selections from Q and at most m controls and updates of the predecessors. Assuming an implementation of set Q with an unordered list or with another structure simple to manage, the complexity of the search algorithm is $O(m)$.

5 Shortest paths

Let there be a weighted digraph $G=(N,A)$ with weights c_{ij} associated with arcs $(i,j)\in A$. The weights c_{ij} are also called lengths of the arcs. A path's length is given by the sum of arc lengths composing it. Now let us pose the problem of seeking the shortest path from a node s to a node t. Let us assume that in the graph there are no cycles with negative length. In fact, the existence of a negative cycle would make it profitable to choose a path containing the cycle and traversing it an infinite number of times, thus producing an infinite sequence of arcs with unbounded total length. In order to avoid this inconvenience we should require that the path we are looking for is elementary, i.e., that it does not traverse the same node more than once, which is far from simple. Later we shall see how to find cycles with negative length in a graph. To formulate the problem we use logical variables related to arcs, which are elementary components of a path p:

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j)\in p \\ 0 & \text{else} \end{cases}$$

The problem's constraints are:

i) the path must start from node s:

$$\sum_{(s,j)\in FS(s)} x_{sj} = 1$$

ii) the path must arrive at node t:

$$\sum_{(j,t)\in BS(t)} x_{jt} = 1$$

ii) if the path traverses an intermediate node *i*, other than *s* and *t*, it cannot stop:

$$\sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} = 0, \forall i \neq s, t.$$

Note how the latter class of constraints is valid for all nodes other than *s* and *t*. In fact, if a node *i* is not part of the path, the sum of incoming arcs is equal to zero and must be equal to the sum of outgoing arcs. Supposing the absence of negative cycles, there is no need to restrict variables x_{ij} to take values 0 or 1, so we just have to impose non-negativity; the problem's formulation can be summarized as follows:

$$\begin{array}{lll} \min & \sum_{(i,j)\in A} c_{ij} x_{ij} & \\ & \sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} = b_i & \forall i\in N \\ & x_{ij} \in \{0,1\} & \forall\ (i,j)\in A, \end{array}$$

where b_i = -1 if i=s, b_i = 1 if i=t, and b_i = 0 else. It is to be noted that in this formulation we have a constraint for each node and a variable for each arc.

5.1 The "rope" problem

The problem of searching for a shortest path from a node *s* to a node *t* of an undirected graph can also be represented and solved by using a non-elastic rope. For each arc (*i*,*j*) of the graph we cut a piece of rope with length c_{ij}. The pieces of rope are knotted according to the definition of the stars of the graph nodes, so that the knots of the tangle correspond to the nodes of the graph. In order to determine the shortest path from a node *s* to a node *t* it is sufficient to take the corresponding knots in the tangle and try to move them apart as much as possible without breaking the threads. The tight pieces of rope correspond to the arcs in the shortest path. The problem can also be described mathematically, using variables π_i (called potential variables) associated with the nodes representing their position, i.e., their height above earth level:

$$\begin{array}{lll} \max & \pi_t - \pi_s & \\ & -\pi_i + \pi_j \leq c_{ij} & \forall\ (i,j)\in A. \end{array}$$

Therefore, the problem is the same as maximizing the difference of potential between node *s* and node *t* so that the difference of potential between the endpoints of each arc do not exceed the length of the arc itself (which is the same as imposing that the threads in the tangle do not break).
Note that if an arc (*i*,*j*) belongs to the shortest path, then the difference of potential between *i* and *j* must be equal to c_{ij}. Further, if we assign potential 0 to node *s*, then the optimal potential of the single nodes belonging to the path from *s* to *t* will be given by the length of the shortest path connecting them to *s*. The formulation can be applied to the directed case as well.

Exercise

Using the modeling language we saw in chapter 1, formulate the problem of the search for a shortest path from 1 to 7 for the graph of Figure 5 and determine the optimal solution. Formulate as well the corresponding rope problem and verify that its optimal solution is equivalent to the one found with the first formulation. Observe the relations between the arcs chosen by the solution as well as the difference between the potentials of the arc endpoints. Note that these two problems have the same input file.

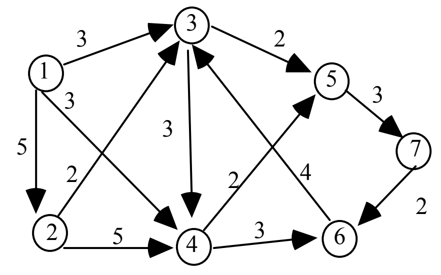

Fig. 5: problem of the shortest path from 1 to 7

5.2 *Application: purchase planning*

A building company uses steel beams of a standard section but of different lengths $L_1<L_2<\ldots<L_n$. The demand for each type of beam i is given by D_i. If not available, a beam of length L_i can be obtained cutting a beam of greater length (rejects are not reused). The cost involved by the storehouse set-up in order to store beams of length L_i is given by K_i and is independent from the number of beams; the cost implied by the purchase of a beam of length L_i is of C_i. The problem of buying the needed beams and of setting up the storehouse in order to store them, so as to minimize the total cost, can be formulated as a problem of search for a minimum length path on an appropriate graph $G=(N,A)$. The set of nodes N is given by $\{0,1,\ldots,n\}$, whereas the set of arcs $A=\{(i,j)\text{: } i<j,\ i,j\in N\}$. The generic arc $(i,j)\in A$ represents the choice of buying and storing beams of length L_j in order to obtain beams of length $L_{i+1},\ldots, L_j$. Then, the length of arc (i,j) is given by $c_{ij}= K_j + C_j(\sum_{h=i+1}^{j} D_h)$. A path from node 0 to node n represents a solution, hence the shortest path provides the optimal solution.

5.3 *Application: car purchase*

A young man looking for a job is offered a 5-year contract as a traveling salesman. Such job implies having a car, so before being employed the young man has to buy a car and afterwards he has to keep it in good working order or, if necessary, replace it within the contract term. The model he is interested in costs 12,000 €. The value of the car on the second-hand market is of 7,000 € if it is resold after one year, 6,000 € after 2 years, 2,000 after 3 years, 1,000 after 4 years and 0 after 5 years. The yearly upkeep cost is of 2,000 € if the car is new, 4,000 after one year's usage, 5,000 after 2 years, 9,000 after 3 years and 11,000 after 4 years. The young man has to define a car purchase strategy so as to minimize the total cost (purchase plus upkeep). The problem can be formulated in terms of search for a path on a graph. The graph has 6 nodes, each node i represents the beginning of the i-th year. Between each pair of nodes i and j (with $i<j$) there exists an arc representing the car purchase at the beginning of the year i and its sale at the end of the year j-1. The arc length is set equal to the operation's cost. Any path from 1 to 6 represents a feasible

strategy for purchase and upkeep. The minimum length path corresponds to the minimum cost strategy. Note that also in this case the graph contains no cycles.

5.4 Shortest-path tree

By slightly modifying the model of the search for the shortest path between two nodes, it is possible to formulate the problem of the search for the set of shortest paths from a node *r* (*root*) to all other nodes of the graph. This set of shortest paths is called *shortest-path tree with root r*. It is called a tree because the set of shortest paths does not contain cycles. In fact, it should be noted that if the shortest path between *r* and a node *s* traverses *q*, then the subpath from *r* to *q* is minimum (otherwise, the path between *r* and *s* would not be minimum either). The mathematical formulation is obtained by placing one on the other and by summing one to the other the constraints of the problem of the search for the shortest path from *r* to every other node, so the result is the following:

$$\begin{array}{lll} \min & \sum\limits_{(i,j)\in A} c_{ij}\, x_{ij} & \\ & \sum\limits_{(j,i)\in BS(i)} x_{ji} - \sum\limits_{(i,j)\in FS(i)} x_{ij} = b_i & \forall i\in N \\ & x_{ij} \geq 0 & \forall\ (i,j)\in A, \end{array}$$

where b_i = -(n-1) if i=r, b_i = 1 else.

Also in the case of the shortest-path tree we can formulate a corresponding “rope” problem, with variables associated with nodes and constraints associated to the graph arcs.

$$\begin{array}{lll} \max & \sum\limits_{i\in N/\{r\}} \pi_i - (n\text{-}1)\, \pi_r & \\ & -\pi_i + \pi_j \leq c_{ij} & \forall\ (i,j)\in A. \end{array}$$

The same properties we saw in the case of the shortest path from *s* to *t* continue to hold, so also in this case the optimal potentials π, having fixed π_r=0, correspond to the lengths of the shortest paths from root *r* to every other node.

Exercise

Formulate and solve for the graph of Figure 5 the problem of the shortest-path tree with root 1 using both formulations we saw.

5.5 Shortest-path tree algorithm on acyclic graphs

Let there be an acyclic weighted digraph G=(N,A). Note that both application examples seen above give rise to a graph of this type. In order to verify whether a graph is acyclic, we just need to check if it is possible to renumber the graph nodes so that if $(i,j)\in A$ then $i<j$. When nodes satisfy this property the graph is said to be *topologically numbered*. In order to renumber an acyclic graph according to a topological numbering we can apply the following simple recursive algorithm, which restores in vector φ the new node numbering:

```
Procedure Topological_numbering($N$,$A$,$\varphi$,$x$)
  begin
    if $\exists\ i\in N$: BS(i)=ø then $\varphi(i)=x$; Topological_numbering($N\setminus\{i\}$, $A\setminus$FS($i$),$x$+1)
    else if $N\neq$ø then return("Non-acyclic graph")
  end.
```

The algorithm assigns the first available number to a node having no incoming arcs, if one exists. Then it proceeds to renumber the rest of the graph starting from the next number. Note how the algorithm can be also used to verify if a graph is acyclic.

Exercise
Evaluate the complexity of the topological numbering algorithm.

From now on we assume *G* to be topologically numbered. Wishing to find the shortest-path tree with root 1, we can proceed by induction. We associate with each node *i* a label *d*[*i*] providing the length of the shortest path from 1 to *i*. A possible algorithm proceeds according to the following inductive rule:

$$d[1] = 0;$$
$$d[k] = \min \{d[i] + c_{ik},\ i=1,\dots,k\text{-}1\},\ k=2,\dots,n.$$

Practically, at each new iteration we go and visit the backward star of each node. Considering the nodes in ascending order, we have the guarantee that the labels of the nodes preceding the current node have already been determined.

Exercise
Formalize the algorithm based on the inductive rule seen above and evaluate its complexity.

A further and equivalent method visits the forward stars of each node. In this case, node labels are gradually updated as a better route is found. At the beginning, except for node 1 whose label is definitively fixed at value 0, labels are set equal to a very high value, such that any path from 1 to the node in question is shorter than that value. The algorithm is the following:

```
Procedure SPT_Acyclic (P,d)
  begin
    P[1]:=1; d[1]:=0; for i:=2 to n do begin P[i]:=1; d[i]:=M end;        {initialization}
    for i:=1 to n-1 do
      for each (i,j)∈FS(i) do
        if d[i]+$c_{ij}$ < d[j] then begin P[j]:=i; d[j]:=d[i]+$c_{ij}$ end
  end.
```

In addition to the label, the algorithm also records how paths are. This is performed by means of the predecessor associated with each node, as we saw in the case of a graph's search algorithm. The graph arcs are examined only once, and for each arc the number of performed operations is constant. This means that, like in the search case, the algorithm's complexity is *O*(*m*), where *m* is the cardinality of the set of arcs.

5.6 Acyclic graphs and dynamic programming

The determination of shortest or longest paths on an acyclic graph is the ground for solving optimization problems with the technique of *dynamic programming*. This technique decomposes

the determination of a feasible solution into a sequence of decisions. Moreover, the solution cost is computed as the sum of the costs of the single decisions. Then a so-called *state graph* is constructed, in which nodes represent a summary of the decisions made in the previous states. Arcs represent the possible transitions from a state to the other. For a deeper analysis of dynamic programming and its applications, see Bertsekas' book[1].
Below is an example of the application of dynamic programming to the solution of a knapsack problem whose generalization can be used for solving other problems we examine in this course. Consider an instance of the knapsack problem:

$$\max \sum_{i=1}^{n} c_i x_i$$
$$\sum_{i=1}^{n} a_i x_i \leq b$$
$$x_i \in \{0,1\},\ i=1,\ldots,n.$$

For more simplicity, we consider the hypothesis that the knapsack capacity b as well as all coefficients a_i are non-negative integers. The decisions into which a solution can be decomposed concern the single objects that can be introduced or not into the knapsack and that we consider according to the sequence imposed by their numbering. The nodes of the state graph are characterized by a pair of information (i,K) where i is the index of the last object considered by the sequence of decisions and K is the total weight that is present in the knapsack. It is quite evident that if we merely intend to consider feasible states, we will just have nodes (i,K) with $i=1,\ldots, n$ and $K=0,\ldots, b$. Further, we add the states (0,0) and $(n+1,b)$ representing the beginning and the end of our decision process. Arcs corresponding to feasible transitions are of two types: from node $(i\text{-}1,K)$ to node (i,K) – which corresponds to the decision of not choosing the object i – and from node $(i\text{-}1,K)$ to node $(i,K+a_i)$, with $K+a_i \leq b$ – which corresponds to the decision of introducing the object i into the knapsack. To the transitions of the first type we can assign value 0, whereas to those of the second type we assign value c_i. To the state graph we add the transitions from nodes (n,K) to node $(n+1,b)$ with value 0. The constructed graph is evidently acyclic. It should be noted that each node represents a possible knapsack state, but several partial solutions can be associated with one state: it is only by determining one of the possible paths from node (0,0) to the node in question that we can find a solution. Any path in the state graph connecting the initial node (0,0) to the final node $(n+1,b)$ corresponds to a feasible solution. Among all solutions we wish to determine the one of maximum value, which obviously corresponds to the path of maximum length. The determination of the longest path on an acyclic graph can be made by adapting the SPT_Acyclic algorithm.

[1] Bertsekas, D. *Dynamic Programming and Optimal Control*. Belmont, MA: Athena Scientifica, 1995.

Dynamic programming enables to efficiently approach several problems. As we may notice, in the two examples concerning the purchase planning and the car purchase the graphs used to support the models are nothing but particular cases of the state graph.

Exercise

Modify the SPT_Acyclic algorithm so as to determine the longest path tree, further apply it to the knapsack example we saw in the preceding chapter.

5.7 Project planning and critical-path method

Path computation on a graph is helpful also in planning a project's activities. A complex project can be decomposed into a number of activities, each characterized by a needed execution time. Activities are mutually related by precedence constraints, which actually govern the development of the project. Let us take a simple example concerning a bicycle's assemblage. Activities are summed up in Table 2, which also includes execution times (in number of minutes).

Activity	Description	Time
A	Assemblage of hubs, spokes, rims and wheel centering	120
B	Gear and fork	60
C	Handlebar and saddle	15
D	Brakes	45
E	Gear lever and derailleur	30
F	Central movement, pedal cranks and pedals	60
G	Mudguards and wheels	40
H	Chain assemblage and gear adjustment	30
I	Assemblage of controls and cables	40
J	Lights	20

Table 2: project activities and times in minutes

If enough staff is available, activities can be carried out simultaneously, provided that the precedence constraints are respected. In the considered example, the precedence constraints are the following: (A→G), (B→C, D), (B, G→I, J), (E, I, G→H), (F→E), (D→I), where $i \rightarrow j$ means that activity i must be complete before activity j begins. The problem of project planning optimization consists in establishing an ordering of the different activities as well as a time location with the objective of minimizing the time between the beginning of the first activity and the end of the last. In order to simplify and to render the formulation uniform, two dummy activities are introduced: the activity "beginning", which precedes all activities that can be carried out as first and therefore never appear on the right side in precedence constraints, and the activity "end" which follows all activities that never appear on the left side in precedence constraints. So, in the exemplified case we have the following new precedences: ("beginning"→A, B, F) and (C,H, J→"end").

To formulate the problem we introduce variables related to activities: for each activity i the variable π_i provides the beginning instant of the activity. The objective function is the following:

$$\min \pi_{end} - \pi_{beginning.}$$

When fixing the beginning times of the different activities, we just have to look at the precedence constraints and operate in such a way that if an activity i precedes another activity j , the end time of i is less than or at most equal to the beginning time of j. Hence, denoting by p_i the execution time of activity i, the constraints are:

$\pi_i + p_i \leq \pi_j$ for each precedence ($i \rightarrow j$).

If constraints are rewritten in the following way

$-\pi_i + \pi_j \geq p_i$

we can observe similarities with the constraints and the formulation of the "rope" problem. So we can try to construct a graph *G* in which nodes correspond to activities and arcs to precedence constraints between pairs of activities. Arc lengths can be set equal to the duration of the activity represented by the tail node of the arc. According to the formulation given above, in this graph our purpose is that of trying to move the "end" node as near as possible to the "beginning" node, unlike we did in the "rope" problem case. Constraints are reversed too; in fact, in the planning problem, while moving the nodes nearer we have to respect the minimum distances given by arc lengths (activity duration).

Starting from these considerations, we try to discover if the solution of the project planning problem as well can be interpreted in terms of path on the graph. We easily realize that the optimal solution of the planning problem corresponds to the path of maximum length from the "beginning" node to the "end" node in graph *G*. Such path is called *critical path* because a delay of any activity belonging to the critical path would lead to an identical delay of the whole project, which is not necessarily true for other activities not belonging to that path.

Exercise

Formulate the problem of the search for the longest path from "beginning" to "end" and remark differences and similarities compared with the shortest-path problem.

Observe that graph *G* is acyclic; in fact, if there was a cycle, this would mean that the precedence constraints are cyclic, which would make the project execution impossible. This implies that we can compute the longest path quite efficiently by adapting the SPT_Acyclic algorithm to the maximum case and determine the solution of the project planning problem.

Exercise

Construct the graph relative to the bicycle assemblage project and compute the critical path.

Exercise

Given the critical path, how can beginning times of the different activities be reconstructed? In case of activities not belonging to the critical path, the beginning time may take different values: how can the feasible range of these values be computed? Apply it to the bicycle assemblage case.

5.8 Dijkstra's algorithm

Let us examine some properties of the SPT_Acyclic algorithm which was previously discussed in relation to the search for a path on an acyclic graph. During the algorithm's execution we have:

i) value $d[i]$ is greater than or equal to the length of the shortest path from r to i;

ii) at each iteration, the optimal value $d[i]$ (definitive) is found for at least one node;

iii) when the $FS(i)$ is examined, value $d[i]$ is optimal.

Particularly, the latter property enables the SPT_Acyclic algorithm to be extremely efficient.

We are now going to present an algorithm which, under appropriate hypotheses, behaves like SPT_Acyclic on non-acyclic graphs as well. The algorithm uses a set *Q* into which are introduced nodes whose forward star must be explored. Initially *Q* contains only the root node. Every time a node label is updated, the node, if not already there, is introduced into *Q*. The key instruction of the algorithm is the selection from *Q* of the node from which the graph search is to be continued. In the case of the SPT_Acyclic algorithm nodes were examined according to the topological ordering given by the numbering. In order to have the certainty of a similar behavior, and in particular the certainty that an examined node will never be introduced into *Q* again, the algorithm examines each time the node *i* having the smallest *d*[*i*] label.

```
Procedure SPT_Dijkstra (r,P,d)
   begin
      for i:=1 to n do begin P[i]:=r; d[i]:=M end; d[r]:=0; Q:={r};            {initialization}
      repeat
         select i from Q such that d[i]=min{d[h]: h∈Q};  {smallest label node is selected}
         Q:=Q\{i};
         for each (i,j)∈FS(i) do
            if d[i]+$c_{ij}$ < d[j] then
            begin P[j]:=i; d[j]:=d[i]+$c_{ij}$; if j∉Q then Q:=Q∪{j} end
      until Q=∅
   end.
```

Considering the hypothesis that arc lengths are all greater than or equal to zero ($c_{ij}\geq 0$), when a node is selected from *Q* it has no possibility to be introduced again into it because it is not possible to find a shorter path connecting it to the root. This implies that the properties i), ii) and iii) already seen for the SPT_Acyclic algorithm hold.

Now let us try to analyze the algorithm's complexity, denoting, as usual, by *n* the number of nodes and by *m* the number of arcs of the graph. In the initialization phase $O(n)$ elementary operations are performed; according to the property seen above, we will have to perform at most *n* selections from set *Q* and at most *m* controls on arcs, label updates and insertions into *Q*. Therefore, everything depends on the data structures used to implement the set *Q*. If an unordered list is used, the selection of the smallest label node implies a scanning of the entire list, so its complexity is $O(n)$, whereas the insertion is made into $O(1)$. Hence, the global complexity is $O(n+n^2+m) = O(n^2)$.

If we use instead a priority queue, for instance a *binary heap*[1], the selection complexity is $O(1)$, whereas removal, insertion and update are $O(\log n)$. Globally, we will have $O(n+n\log n+m\log n) = O(m\log n)$.

The priority queue implementation is advantageous when $m\log n<n^2$, that is when the number of arcs is less than $n^2/\log n$.

Example of execution of Dijkstra's algorithm

Let us consider the following weighted digraph and apply the algorithm SPT_Dijkstra starting from root node 1. Before we begin the execution, it should be observed that the graph is not acyclic

[1] For details on priority queues, see: Cormen, T. H., Leiserson, C. E. and Rivest, R. L. *Introduction to Algorithms*. Cambridge, MA: MIT Press; New York: McGraw-Hill, 1990.

(except for node 1, there are arcs arriving at and departing from each node, so it is not possible to apply a topological numbering). Moreover, all arc lengths are positive.

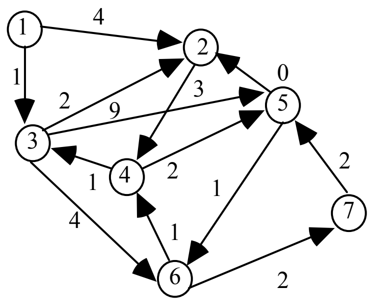

Fig. 6: graph on which to apply SPT_Dijkstra with r=1

Initially *Q*={1}, then we select 1 from *Q* and visit its forward star. Labels of nodes 2 and 3 have value 1 and 4, respectively, and both of them are introduced into set *Q* (*Q*={2,3}). Since the smallest label is 1, we select 3 from *Q* and render its label definitive (Fig. 7).

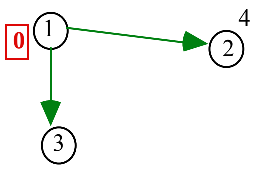

Fig.7: node labels visited at the beginning of the second iteration; definitive labels are pointed out.

By visiting the forward star of node 3 we can improve the labels of nodes 2 (whose value had not been definitively fixed yet), 5 and 6; node 2 is already present in *Q*, whereas 5 and 6 are introduced for the first time, hence *Q*={2,5,6} and the smallest label node is 2, which is selected from *Q* and whose label is fixed (Fig. 8).

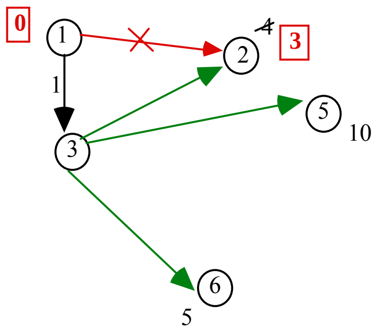

Fig. 8: labels at the beginning of the third iteration; variation at the predecessor of node 2

Visiting the forward star of 2 we can only improve the label of node 4 (which is introduced into *Q*); in fact, node 6, although it can be reached from 2, has a label smaller than 3+5. *Q*={4,5,6} and the smallest label node is 6, which is subsequently selected from *Q*.

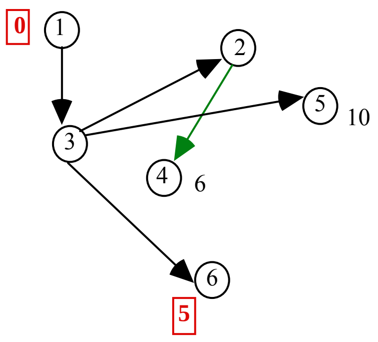

Fig. 9: labels at the beginning of the fourth iteration

Visiting the forward star of 6 we improve the label of 7, but not that of 4, which remains unchanged. $Q=\{4,5,7\}$. The smallest label node is 4, which is removed from Q.

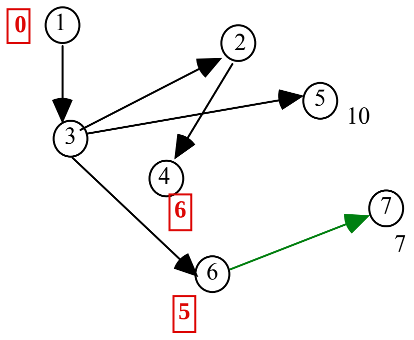

Fig. 10: labels at the beginning of the fifth iteration

Visiting the forward star of 4 we can improve the label of 5, so we update its pointer. $Q=\{5,7\}$ and the smallest label node is 7. Visiting its forward star we do not improve any label, and the same occurs for node 5 at the next iteration. The final situation is represented in Figure 11.

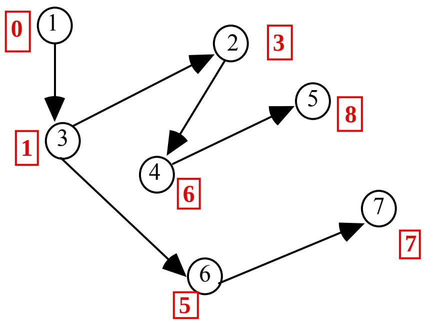

Fig. 11: shortest-path tree with root 1

5.9 *SPT_L (label correcting) algorithm*

If the graph is not acyclic and some of the lengths associated with arcs are negative, the properties seen above do not hold any more. In particular, it is not necessarily true that a node, once selected from Q, cannot enter it again. It can be proved that, in some pathological cases, Dijkstra's algorithm has a particularly inefficient behavior and its complexity is $O(2^n)$, thus exponential.
In trying to provide an efficient algorithm, we keep nevertheless an algorithmic scheme similar to those previously seen. In particular, we use node labels $d[i]$ such that at each new step of the algorithm they represent a higher estimate of the optimal potential value ($d[i]\geq\pi_i$). Referring again to the "rope" problem formulation, the necessary optimality conditions are (*Bellman's conditions*):

$$d[j] \leq d[i] + c_{ij}.$$

If we consider any path from r to j, $P(r,j)$, then for optimal labels we have

$$d[j] \leq \sum_{(i,h)\in P(r,j)} c_{ih},$$

so $d[j]$ is exactly the length of the shortest path from r to j.

Therefore, a possible algorithm does nothing but check arc by arc that Bellman's conditions are valid, and if not it updates the label of the arc's head node so as to render the conditions valid with equality. The algorithm's general scheme is the following:

```
Procedure SPT_L (r,P,d)
  begin
    for i:=1 to n do begin P[i]:=r; d[i]:=M end; d[r]:=0;          {initialization}
    while ∃(i,j)∈A such that d[i]+$c_{ij}$ < d[j] do
    begin                     {update of the label not satisfying Bellman's conditions)
      P[j]:=i; d[j]:=d[i]+$c_{ij}$;
    end
  end.
```

If no particular criteria are adopted for the selection of the arcs to be controlled, the algorithm may require an exponential number of iterations.

Complexity

In order to evaluate the complexity of the preceding algorithm, in which no particular criterion is adopted for the choice of the arc to be controlled, it suffices to consider the value of labels. We assume that the graph lengths are integer. Defining $C = \max\{|c_{ij}|: (i,j)\in A\}$, we can state that for each node i, $-(n-1)C \leq d[i] \leq (n-1)C$. Given that at each new iteration a decrease of at least one unit for one node takes place, and that in order to determine an arc violating Bellman's conditions – in the worst case – the scanning of the entire node list is necessary, the algorithm's complexity is $O(2(n-1)Cnm)$, hence $O(n^2mC)$.

Exercise

Construct a graph and find a sequence in which to control the arcs such that the SPT_L algorithm requires $O(2^n)$ iterations.

Polynomial variant

By appropriately choosing a criterion to be adopted in searching for arcs on which to perform the test of Bellman's conditions, the SPT_L algorithm can become polynomial. Let us consider the arcs in any fixed order. The criterion according to which we seek an arc violating Bellman's conditions scans the ordered list of arcs in a cyclic way; when an arc not satisfying the conditions is found, we correct the head node label and we continue the scanning from the next arc on the list. The completion of an entire scanning is called a *phase*. The complexity measure is based on the estimate of how many phases there can be at most before all arcs satisfy Bellman's conditions.

Phases can be at most $(n-1)$; the demonstration is based on the meaning of labels $d[i]$. At the end of phase k, label $d[i]$ provides the *length of the shortest path from root r to i which uses at most k arcs*. The demonstration is by induction.

For k=1 labels are corrected only at node heads in FS(r), whereas for all other nodes labels remain unchanged at M, since they cannot be reached from the root. Hence, what was stated is valid.
Supposing that it is valid for $k \geq 1$, we wish to demonstrate that at the end of phase (k+1) the label $d[j]$ of each node has as value the length of the shortest path from root r to j which uses at most (k+1) arcs. Consider a node j connected to r by a path (r - i_1 - i_2 - … - i_k - j), which proves to be the shorter path of k+1 arcs between r and j, and there exist no shorter ones using less arcs. Note that the path (r - i_1 - i_2 - … - i_k) proves to be the shorter one between r and i_k , and at the end of phase k, due to inductive hypothesis, $d[i_k]$ must be equal to the length of this path. Therefore, when in phase k+1 arc (i_k, j) is examined, we set $d[j]$ equal to the length of path (r - i_1 - i_2 - … - i_k - j), thus confirming the thesis.
As a consequence, given that a simple path may contain at most (n-1) arcs, the resulting complexity of the algorithm is $O(nm)$.

Implementing aspects

It should be noted that when $d[j]$ is decreased, all arcs (j,k) in $FS[j]$ may potentially violate Bellman's conditions, in fact $d[j]+c_{jk}$ might have become smaller than $d[k]$. Considering this observation, we can lead the search for arcs violating Bellman's conditions by memorizing in a set Q the nodes of which we corrected the labels and which potentially have arcs with violated conditions in their forward star. The algorithm becomes:

```
Procedure SPT_L (r,P,d)
   begin
        for i:=1 to n do begin P[i]:=r; d[i]:=M end; d[r]:=0;        {initialization}
      repeat
        select i from Q; Q:=Q\{i};
        for each (i,j)∈FS(i) do
          if d[i]+$c_{ij}$ < d[j] then
          begin
             P[j]:=i; d[j]:=d[i]+$c_{ij}$; Q:=Q∪{j}
          end
      until Q=∅
   end.
```

When insertions into and selections from Q are performed according to a FIFO policy (then Q is a queue), the algorithm's global complexity is $O(nm)$, given that the polynomial variant of the scanning of arcs is implemented, as if they were ordered in the list preserving the consecutive forward stars. In the scanning, the queue Q allows to "skip" those forward stars being certainly not concerned by the modification of labels, thus producing improvements in the algorithm's execution times, although in the worst case such improvements have no direct influence on complexity.

Example: execution of the SPT_L algorithm with FIFO policy

Let us consider the graph of Figure 12 and apply the SPT_L algorithm starting from root 1.

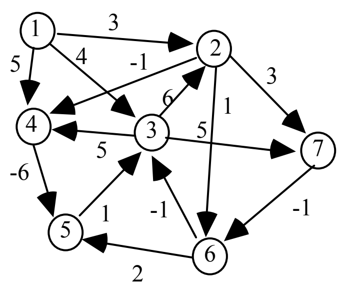

Fig. 12: graph on which to apply SPT_L with r=1

We observe that the graph is not acyclic and has arcs of negative length, therefore it is not appropriate to apply SPT_Acyclic nor even SPT_Dijkstra. Initially *Q*={1}, we select node 1 from set *Q* and visit its forward star. Since at the beginning all nodes have label *M*, we can correct the labels of 2, 3 and 4 (Fig. 13).

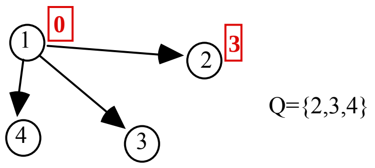

Fig. 13: tree and set *Q* at the end of the first iteration

At the next iteration node 2 is selected from *Q*, as it was introduced before the others, and its forward star is visited. The operation leads to the correction of the labels of nodes 4 (already present in *Q*), 6 and 7 (Fig. 14).

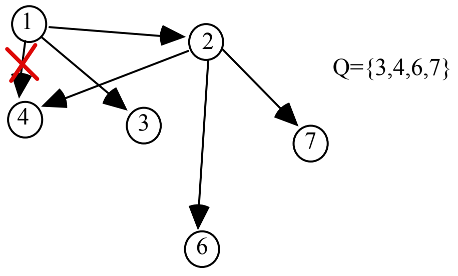

Fig. 14: tree and set *Q* at the end of the third iteration

Visiting the forward star of node 3 we improve the label of node 6 (Fig. 15).

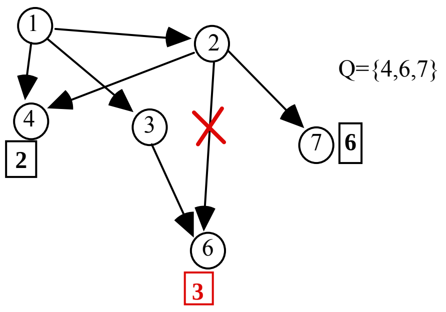

Fig. 15: tree and set *Q* at the end of the fourth iteration

Visiting the forward star of node 4 we improve the label of node 5, which is introduced into *Q* (Fig. 16).

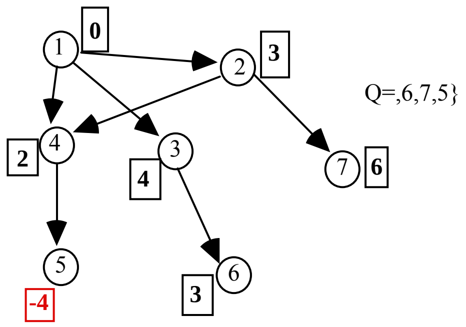

Fig. 16: tree and set *Q* at the end of the fifth iteration

Subsequently node 6 is selected, from which no label can be improved. The same occurs for node 7. Then we select node 5 and while visiting its forward star we improve the label of node 3, which is introduced into *Q*. Note that this is occurring for the second time. Note also that now the labels of all nodes descending from 3 (i.e., 6) have improvable labels, although not necessarily implying a modification of the predecessors (Fig. 17).

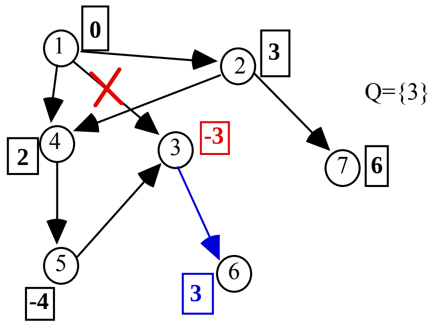

Fig. 17: tree and set *Q* at the end of the eighth iteration

Visiting the forward star of node 3 implies the improvement of the labels of nodes 6, 7, which are introduced into *Q* (Fig. 18). Next iterations do not lead to further modifications.

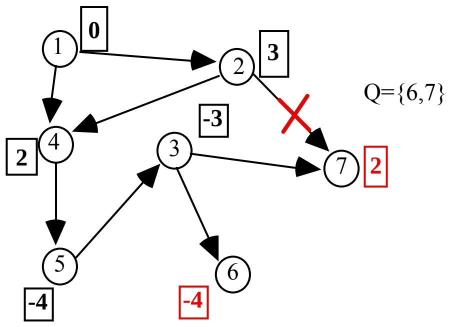

Fig. 18: tree and set *Q* at the end of the ninth iteration

5.10 Detection of negative length cycles

There are different ways to recognize negative length cycles. If such cycles are present, all the algorithms we saw previously do not stop and go on cycling indefinitely.

Let C be the maximum absolute value of arc lengths; evidently, if there are no negative length arcs, it will never be possible to have labels $d[j]<-(n-1)C$. As soon as, in the iterations of the algorithm, such eventuality takes place, we can stop. The negative length cycle can be reconstructed by backtracking along predecessors P starting from j, until j is not reached again.
Negative length cycles can also be detected in a more efficient way. By using the SPT_L algorithm implemented with a queue, we know that a node can be introduced in the rear no more than n-1 times, if there are no negative cycles. Therefore, as soon as a node is introduced into Q for the nth time, this means that a negative cycle is present. So, with a complexity of $O(nm)$ we detect negative length cycles. Nodes still being in queue Q after n-1 iterations of the algorithm are part of negative cycles. In order to reconstruct the cycles, it suffices to go up the predecessors of the nodes present in Q.

Exercise
Applying the procedure explained above, discover if in the graph of Figure 12 – in which the arc (3,2) has length 5 instead of 6 – there exist negative length cycles.

*5.11 *Root change of the shortest-path tree*

Suppose we have to compute the shortest-path trees for more than one root ($r_1, r_2, \ldots, r_p$). So, for each root we should apply the shortest-path algorithm. However, after having computed the first tree relative to root r_1, we can try to take advantage of the information relative to the detected solution in order to speed up the computation of the other trees. A possible way to do this takes into account the *reduced costs* of arcs with respect to the detected solution. Let $d[i]$ be the optimal labels relative to the shortest-path tree with root r_1, then the reduced cost of arc (i,j) is given by:

$$\bar{c}_{ij} = c_{ij} + d[i] - d[j].$$

We compare the cost with the reduced cost of a path $P_{hk} = (h, i_1, i_2, \ldots, i_s, k)$:

$$C(P_{hk}) = \sum_{(i,j)\in P_{hk}} c_{ij}$$

$$\bar{C}(P_{hk}) = \sum_{(i,j)\in P_{hk}} (c_{ij} + d[i] - d[j]) = c_{ij} + d[h] - d[k].$$

Hence, a path's reduced cost is equal to the path cost, with the exception of a constant which depends on the initial node and on the final node but is independent from intermediate nodes. This means that if a path is shortest with respect to reduced costs, it is so also with respect to original costs. The advantage of using reduced costs consists in the fact that reduced costs have value greater than or equal to zero (why?), so it is possible and more advantageous to apply Dijkstra's algorithm. The complexity of computing the shortest-path trees for roots ($r_1, r_2, \ldots, r_p$) on an ordinary-cost graph is given by $O(mn)$ for the computation of the first tree plus p times the application of the most efficient version of Dijkstra's algorithm.

Exercise
Starting from the shortest-path tree of the graph of Figure 12, as represented in Figure 18, determine the shortest-path tree with root 3.

**5.12 Floyd-Warshall algorithm: shortest path between each pair of nodes*

If we have to compute the shortest path between each pair of nodes, we can refer to the procedure we briefly described in the preceding paragraph, having a worst-case complexity of $O(m\ n + n\ D(n,m))$, where $D(n,m)$ is the complexity of the employed Dijkstra's algorithm that varies according to the data structure being used. Alternatively, we can refer to Floyd-Warshall algorithm which, using no particular data structures (two $n \times n$ matrices are enough), computes the required paths with constant complexity. The algorithm's idea is based on a simple recursive rule. If we denote by $d^k[i,j]$ the length of the path from i to j using as intermediate nodes $\{1,\ldots, k\text{-}1\}$, then what follows is valid:

$$d^{k+1}[i,j] = \min\{d^k[i,j],\ d^k[i,k]+d^k[k,j]\}.$$

The shortest path from i to j using $\{1,\ldots,k\}$ as intermediate nodes is given by the shorter of the two following paths: the shortest path from i to j using $\{1,\ldots,k\text{-}1\}$ as intermediate nodes and the path obtained putting together the shorter path from i to k and the shorter path from k to j, both using $\{1,\ldots,k\text{-}1\}$ as intermediate nodes. The recursion rule is completed by the initial defined value only if between i and j there exists a direct connection:

$$d^1[i,j] = \begin{cases} c_{ij} & \text{if } (i,j) \in A \\ \infty & \text{if } (i,j) \in A \end{cases} .$$

In order to record the paths we make use of predecessors, just like in the tree case, but in this case we need a matrix instead of a vector. The algorithm is summarized as follows:

```
Procedure Floyd_Warshall(c,p,d);
   begin
      for i:=1 to n
         for j:=1 to n
            if i=j then begin d[i,i]:=0; p[i,i]:=ø end
            else if (i,j)∈A then begin d[i,j]:=$c_{ij}$; p[i,j]:=i end
               else d[i,j]:=M;
      for k:=1 to n
         for i:=1 to n
            for j:=1 to n
               if d[i,j]>d[i,k]+d[k,j] then
                  begin
                     d[i,j]:=d[i,k]+d[k,j]; p[i,j]:=p[k,j];
                  end
   end.
```

It is uninteresting to analyze the algorithm's computational complexity: since the dominant operation is composed of three nested cycles which scan all nodes, the complexity is $O(n^3)$. The identification of negative cycles is immediate as well: if at the end of the algorithm $d[i,i]$ is less than 0, it means that node i belongs to a negative cycle.

6 Maximum flow

Given a digraph $G=(N,A)$ with u_{ij} capacities associated with arcs, the maximum flow problem consists in finding the maximum amount of flow that can be sent from node $s\in N$ (called *source*) to node $t\in N$ (called *sink*), while respecting arc capacities as well as the fact that flow must not be dispersed at intermediate nodes. The problem's variables are the quantities of flow sent to every single arc: x_{ij}, for each $(i,j) \in A$. The problem can be formulated as follows:

$$
\begin{aligned}
\max \sum_{(s,j)} x_{sj} & \\
0 \le x_{ij} \le u_{ij} & \qquad \forall (i,j)\in A \\
-\sum_{(i,j)\in FS(i)} x_{ij} + \sum_{(j,i)\in BS(i)} x_{ji} = 0 & \qquad \forall i\in N\setminus\{s,t\}.
\end{aligned}
$$

The first group of constraints – associated with arcs – are called *capacity constraints*. The second group of constraints – associated with all graph nodes except s and t – are the so-called *flow conservation constraints*, which impose that at each intermediate node the incoming flow must be equal to the outgoing flow, thus guaranteeing the absence of dispersions. The objective function computes the total flow going out from node s. Note that, due to flow conservation, we might alternatively have considered the sum of flow coming into t.

Wishing to write the problem more uniformly, we can introduce a dummy arc (t,s) with capacity $u_{ts}=M$, where M is an appropriately large value such that M is greater than the optimal solution value. The maximum flow problem can be equivalently rewritten as follows:

$$
\begin{aligned}
v = \max x_{ts} & \\
0 \le x_{ij} \le u_{ij} & \qquad \forall (i,j)\in A' \\
-\sum_{(i,j)\in FS(i)} x_{ij} + \sum_{(j,i)\in BS(i)} x_{ji} = 0 & \qquad \forall i\in N.
\end{aligned}
$$

where $A'=A\cup\{(t,s)\}$.

6.1 *Example of maximum flow problem*

Consider a car repair shop with M workers (all having identical working capacities) and a set of cars $J=\{1,\ldots,n\}$ to be repaired. Each car to be repaired j is characterized by the triplet (p_j,r_j,d_j), where p_j is the time needed to repair it, r_j is the time at which the car owner brings it to the shop and d_j is the time at which he comes to collect it (due date). Repairs can be interrupted and resumed (pre-emption) without loss of time, but they must be performed by one worker at a time. The problem is to determine the assignment of repair-jobs to workers so that cars can be returned within the fixed times. The problem can be solved by searching for a maximum flow on an appropriate graph. Let us consider an example in which $M=3$; the data concerning repair work in J appear in the following table:

J	1	2	3	4
p_j	1.5	1.25	2.1	3.6
r_j	3	1	3	5
d_j	5	4	7	9

Table 3: repair execution times, car arrival and car leaving times

In order to construct the graph, we arrange the "events" (car arrival or car leaving) in ascending order and eliminate repetitions:

1, 3, 4, 5, 7, 9.

Between two consecutive "events" we have time intervals of uninterrupted work (T_{kl}):

T_{13}(2 hours), T_{34}(1 hour), T_{45}(1 hour), T_{57}(2 hours), T_{79}(2 hours).

During interval T_{kl} it is possible to repair cars arrived before k ($r_j \leq k$) and collected after l ($d_j \geq l$). The graph is constructed as follows:

- the set of nodes includes a source s, a sink t, a node for each car $j \in J$ and a node for each interval T_{kl};
- there are arcs connecting s to each node $j \in J$, such arcs have capacity equal to repair time p_j; each node $j \in J$ is itself connected to all intervals T_{kl} during which the car can be repaired, the capacity of such arcs is given by (l-k); finally we have arcs connecting nodes T_{kl} to sink t, the capacity of such nodes is given by the working capacity of the shop during that interval, that is $M(l-k)$.

The flow on the arcs of a graph constructed in this way represents the amount of work that was accomplished.

If the maximum flow that can be sent from s to t is equal to the sum of repair times, then it is possible to repair all cars within the fixed times.

6.2 *Properties of flows and cuts*

A partition of the nodes into two subsets (N_s,N_t) such that $s \in N_s$, and $t \in N_t$, is called *s-t cut*. The arcs crossing the cut (having one endpoint in N_s and the other in N_t) are themselves partitioned into two subsets: the set of *direct arcs* $A^+(N_s,N_t)=\{(i,j) \in A\colon i \in N_s, j \in N_t\}$, and the set of *inverse arcs* $A^-(N_s,N_t)=\{(i,j) \in A\colon i \in N_t, j \in N_s\}$.

Given a cut (N_s,N_t) the *quantity of flow crossing the cut* is given by the quantity of flow traversing direct arcs minus the quantity of flow traversing inverse arcs:

$$x(N_s,N_t) = \sum_{(i,j) \in A^+(Ns,Nt)} x_{ij} - \sum_{(i,j) \in A^-(Ns,Nt)} x_{ij}.$$

A *cut's capacity* (N_s,N_t) is given by the sum of capacities of direct arcs:

$$U(N_s,N_t) = \sum_{(i,j) \in A^+(Ns,Nt)} u_{ij}.$$

Obviously, for each feasible flow, as well as for each *s*-*t* cut, it occurs that the flow crossing the cut is smaller than or equal to the capacity of the cut itself. Given a feasible flow $\bar{x}$ with value $\bar{v}$, it can be easily verified that, due to flow conservation, for each cut (N_s,N_t)

$$\bar{x}(N_s,N_t) = \bar{v}\ .$$

It is intuitively clear that the cut most limiting the flow transfer from *s* to *t* is the one of minimum capacity. We can also state that if a flow *x* is such that there exists a cut with capacity equal to the quantity of flow crossing it, such flow is optimal. A question spontaneously arising now is whether the minimum cut capacity value is not exactly equal to the maximum flow value. This can be constructively demonstrated by thinking of a possible algorithm for solving the maximum flow problem.

6.3 *Augmenting path algorithm*

We try to solve the problem incrementally: given a feasible flow, we test if it is improvable, i.e., if there exists a way of routing more flow from *s* to *t*. In order to discover if such flow augmentation is practicable, we introduce the *residual graph of a flow* $\bar{x}$. Given a graph $G=(N,A)$ with capacity on the arcs *u* and a feasible flow $\bar{x}$, we define the *residual graph* $G_R(\bar{x}) = (N,A(\bar{x}))$, where arcs are defined as follows:

$$A(\bar{x}) = A^+(\bar{x}) \cup A^-(\bar{x}),$$

$$A^+(\bar{x}) = \{(i,j): (i,j)\in A,\ \bar{x}_{ij}<u_{ij}\},$$

$$A^-(\bar{x}) = \{(i,j): (j,i)\in A,\ \bar{x}_{ji}>0\}.$$

Example: residual graph

Consider the graph of Figure 19, where a feasible flow is shown as well.

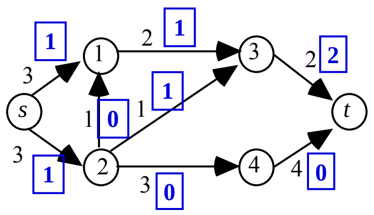

Fig. 19: network flow: in the squares are the flow values on the different arcs

We can construct the residual graph by testing, arc by arc, if the flow can be augmented (by introducing an arc of $A^+(\bar{x})$) and/or diminished (by introducing an arc of $A^-(\bar{x})$) (Fig. 20).

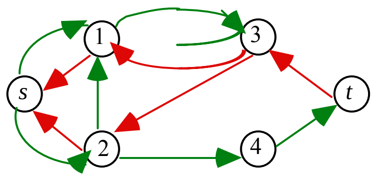

Fig. 20: residual graph in which the arcs of $A^+(\bar{x})$ are distinguished from those of $A^-(\bar{x})$

Note that the arcs of the residual graph correspond to the arcs of the original graph, on which it is possible to apply a flow variation. The flow variation corresponding to an arc (i,j) of $A^+(\bar{x})$ is positive, i.e., it is possible to augment the flow $\bar{x}_{ij}$; the *residual capacity* of arc $(i,j)\in A$ with respect to this variation is given by $(u_{ij}-\bar{x}_{ij})$, since the maximum capacity cannot be exceeded. The flow variation corresponding to an arc (i,j) of $A^-(\bar{x})$ is negative, i.e., it is possible to diminish the flow $\bar{x}_{ji}$; as no negative flow can be obtained, the residual capacity of arc $(j,i)\in A$ with respect to this variation is given by $\bar{x}_{ij}$.

Given a feasible flow $\bar{x}$, consider the residual graph $G_R(\bar{x})$. Searching for a path from s to t, corresponding to a possible flow augmentation, two exclusive cases may occur:

i) there exists a path from s to t in $G_R(\bar{x})$;

ii) there exists a cut (N_s,N_t) in $G_R(\bar{x})$ such that $A^-(\bar{x})(N_s,N_t) = \emptyset$.

In the first case the flow $\bar{x}$is not optimal and it can be increased by a strictly positive quantity, whereas in the second case $\bar{x}$is optimal.

Searching for the path from s to t, we apply the search algorithm to the residual graph $G_R(\bar{x})$ starting from node s. The search algorithm ends either marking t (case i) or letting t – and possibly other nodes – be non-marked (case ii).

In the first case we found an *augmenting path* (on which the flow can be varied without violating the capacity constraints). Let P_{st} be such path. Now we define the maximum quantity of flow θ that can be sent on the detected path and that is defined by the minimum of residual capacities of the corresponding arcs on the original graph:

$$\theta = \min\{\{u_{ij}-\bar{x}_{ij}: (i,j)\in A^+(\bar{x})\cap P_{st}\}, \{\bar{x}_{ji}: (i,j)\in A^-(\bar{x})\cap P_{st}\}\}.$$

Note that, due to how the residual graph was constructed, θ>0. The flow variation, also called *flow augmentation* operation, is performed according to the following rules:

$$x'_{ij} = \begin{cases} \bar{x}_{ij} + \theta & \text{if } (i,j)\in A^+(\bar{x})\cap P_{st}\} \\ \bar{x}_{ij} - \theta & \text{if } (j,i)\in A^-(\bar{x})\cap P_{st}\} \\ \bar{x}_{ij} & \text{else} \end{cases}$$

The value of the new flow x' equals the value of $\bar{x}$increased by θ, hence the flow we started from was not optimal.

Otherwise, in the second case we can construct a cut by placing in N_s the set of nodes marked during the search of $G_R(\bar{x})$ and in N_t all the other ones. Let us consider the cut arcs in the original graph G. According to construction, if $(i,j)\in A^+(N_s,N_t)$, then $\bar{x}_{ij} = u_{ij}$; conversely, if $(i,j)\in A^-(N_s,N_t)$, then $\bar{x}_{ij} = 0$. If not, in $G_R(\bar{x})$ there would exist an arc (i,j), so j would have been marked and would not be in the set N_t. So, we have:

$$\bar{x}(N_s,N_t) \quad \sum_{(i,j)\in A^+(N_s,N_t)} \bar{x}_{ij} - \sum_{(i,j)\in A^+(N_s,N_t)} \bar{x}_{ij}. = \sum_{(i,j)\in A^+(N_s,N_t)} u_{ij}. = U(N_s,N_t).$$

This proves the equivalence of the maximum flow value and the capacity of the minimum cut. Now it becomes easier to provide an algorithm for solving the maximum flow problem, in which –

starting from an initial feasible flow (for example the null flow) – we search for an augmenting path using the residual graph updated at each flow variation. The algorithm having this form is known as augmenting path algorithm or Ford-Fulkerson algorithm.

```
Procedure Augmenting_paths (G,x);
   begin
      x:=0;
      repeat
         Gr:=Residual_graph(x);
         Search (Gr,s);
         if P[t]≠0 then Augment_flow(P_st,x)
      until P[t]=0;
   end.
```

The procedure "Augment_flow" sends the maximum quantity of flow θ from *s* to *t* on the path which was found during the search, according to the rule stated above.

Example: execution of the augmenting path algorithm

Let us continue the example of Figures 19-20: the graph search starting from node *s* finds a first path *s*, 2, 4, *t*. θ=min{3-1, 3-0, 4-0}=2 because it is merely composed of arcs of $A^+(\bar{x})$. The new flow is obtained by summing θ=2 to the flow already existing on the path arcs. Hence $\bar{x}_{s2}=3$, $\bar{x}_{24}=\bar{x}_{4t}=2$. The new residual graph is shown in Figure 21.

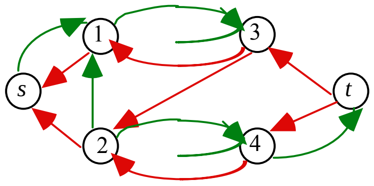

Fig. 21: the residual graph after the first flow update

By applying again the search starting from node *s* we find a second path: *s*, 1, 3, 2, 4, *t*. Note that in this case we use an arc of $A^-(\bar{x})$ (the arc (3,2)). θ=min{3-1, 2-1, 1, 3-2,4-2}=1. The flow update implies an augmentation of 1 unit of flow on the arcs (*s*,1), (1,3), (2,4), (4,*t*) and, on the other hand, a diminution of 1 unit of flow on the arc (2,3). The new residual graph is represented in Figure 22; on such graph it is not possible to find again a path from *s* to *t*, in fact there is a cut $(N_s, N_t)=(\{s, 1\}, \{2, 3, 4, t\})$.

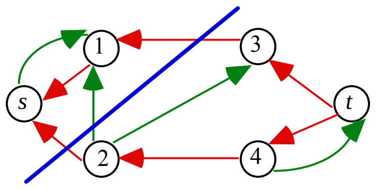

Fig. 22: the residual graph after the second flow update and the cut (N_s, N_t)

The flow updated after the two performed iterations is represented in Figure 23. It should be noted that in the detected cut the flow on the arcs of $A^+(N_s,N_t)=\{(s,1), (1,3)\}$ is placed at the upper capacity, whereas on the arcs of $A^-(N_s,N_t)=\{(2,1)\}$ it is at zero, further the flow crossing the cut equals the capacity of the cut itself.

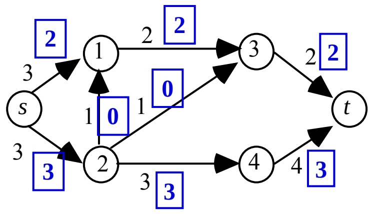

Fig. 23: flow updated after two iterations of flow augmentation

<u>* Edmonds-Karp algorithm</u>

If the augmenting path is not appropriately chosen, pathological cases may take place, causing a huge increase in the number of iterations being necessary to determine the optimal solution.

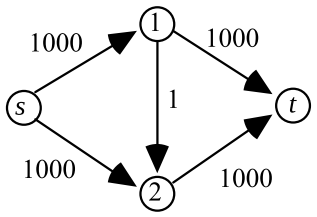

Fig. 24: example of a pathological case

The example of Figure 24 clearly shows that the maximum flow is 2,000; this solution can be determined with just two iterations of the augmenting path algorithm. If at each new step the augmenting path algorithm alternately considers the arcs (1,2) and (2,1) of the residual graph, the maximum flow is determined by performing 2,000 iterations, each being able to augment the flow of only one unit. In order to avoid such cases, the paths on which the flow is augmented must be appropriately chosen.

Actually, the Edmonds-Karp algorithm is a peculiar form of the augmenting path algorithm and it ensures the polynomiality of the number of iterations. The only trick to be adopted concerns the choice of the augmenting path. At each iteration we choose the augmenting path of minimum length, that is the one using the lowest number of arcs. The search for such path can occur by performing the Breadth First Search, thus by implementing the set Q of the search algorithm with a queue. This trick does not overload the complexity of the search algorithm, which remains $O(m)$.

Choosing the augmenting path with the lowest number of arcs results in considering paths with length that grows with the number of iterations of the algorithm. Iteration by iteration, the configuration of the residual graph on which we seek the augmenting paths evolves because in the operation "Augment_flow" at least one arc of the path reaches either the maximum capacity or zero, so at each new iteration at least one arc of the residual graph disappears, although other arcs may appear. This happens as a direct consequence of the choice of flow quantity θ to be sent along the path. Since paths going from s to t in the residual graph may have at most (n-1) arcs, the number of iterations is limited. There still remains a critical question to solve: the estimation of how many times an arc (i,j) can disappear from the residual graph; in other words: how many times an arc of the original graph becomes saturated or cancelled out. Let us consider a specific iteration of the algorithm in which arc (i,j) of the residual graph (Fig. 25) is used as being part of the path from s to t and according to which θ is equal to the residual arc capacity. We denote by $d[i]$ and $d[j]$ the number of arcs of the path going from *s* to *i* and from *s* to *j*, respectively. Since (*i*,*j*) is part of the path, it must be:

$$d[j]=d[i]+1.$$

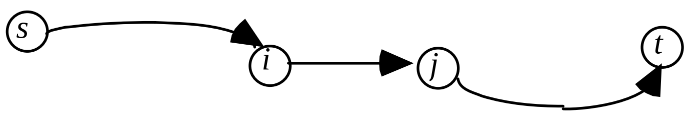

Fig. 25: first augmenting path on the residual graph using (*i*,*j*)

Once we have updated the flow, arc (*i*,*j*) disappears from the residual graph because we have used up its residual capacity. If not already present, arc (*j*,*i*) will appear. In order to bring into play again arc (*i*,*j*), it is necessary that in some iteration of the algorithm an augmenting path is found (with length greater than or equal to the length of the paths previously found) which uses the arc (*j*,*i*) (Fig. 26).

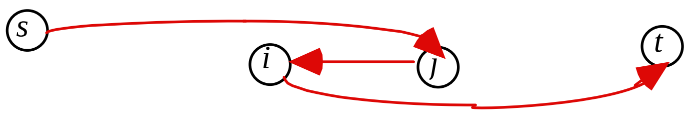

Fig. 26: augmenting path using (*j*,*i*)

Hence, considering the new path and denoting by *d'*[*i*] and *d'*[*j*] the number of arcs of the path going from s to *i* and to *j*, we have:

$$d'[i] = d'[j]+1;$$

with $d'[j]\geq d[j]$. Once we have modified the flow along the detected path, the arc (*i*,*j*) appears again in the new residual graph. This arc can be used (and possibly be saturated) just when we find a new augmenting path between *s* and *t* which uses (*i*,*j*) (Fig. 27), therefore when:

$$d''[j] = d''[i]+1;$$

with $d''[i]\geq d'[i]$.

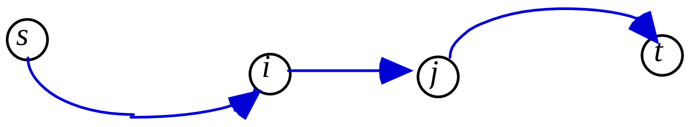

Fig. 27: second augmenting path on the residual graph using (*i*,*j*)

We estimate how much the path going from *s* to *j* has lengthened if compared with the first time the arc (*i*,*j*) was used. We know that

$$d''[j] = d''[i]+1 \geq d'[i]+1 = d'[j]+1 + 1 \geq d[j] + 2.$$

This means that the length of the path using the arc (*i*,*j*) has lengthened of at least 2, therefore, since paths can be long at most (*n*-1) arcs, an arc can disappear from the residual graph at most (*n*-1)/2 times. By repeating this reasoning for all *m* arcs, the maximum number of iterations of Edmonds-Karp algorithm becomes *O*(*nm*). Since the operation of each iteration being computationally heavier is the search of the residual graph with complexity $O(m)$, the resulting algorithm's complexity is $O(m^2n)$.

<u>* Capacity Scaling algorithm</u>

The Capacity Scaling algorithm as well is a peculiar type of augmenting path algorithm. The Capacity Scaling algorithm for the maximum flow problem is based on the idea of augmenting the flow along paths having a high residual capacity. The detection of the path from s to t having maximum capacity, although relatively simple, would bring few advantages in terms of complexity, if compared with other algorithms. In fact, a greater efficiency of the detected path would cost a longer computing time for finding such path. Yet, the idea is interesting, so we make use of an approximation maintaining the simplicity of a search algorithm while looking for a high capacity path going from s to t. A parameter Δ is fixed for selecting arcs having sufficient capacity. Given a feasible flow $\bar{x}$, we call $G_R(\bar{x},\Delta)$ the subgraph of $G_R(\bar{x})$ containing only arcs of residual capacity greater than or equal to Δ. By assuming integer capacities, we have $G_R(\bar{x}) = G_R(\bar{x},1)$. Denoting by U the maximum capacity of the arcs of G, the algorithm can be summarized as follows:

```
Procedure Capacity Scaling (G,x);
   begin
      x:=0;
      $\Delta:=2^{\lfloor \log U \rfloor}$;
      while $\Delta \geq 1$ do
      begin
         Augmenting_paths (Gr(x, $\Delta$),x)
         $\Delta:=\Delta/2$;
      end
   end.
```

If we consider integer capacities, we realize that the algorithm ends with the optimal solution. In fact, at each external iteration parameter Δ is halved and at the last iteration we have $G_R(\bar{x}) = G_R(\bar{x},1)$. Hence, if the flow detected by the Capacity Scaling algorithm does not admit any augmenting paths, we can state with reason that such flow is optimal. Concerning the analysis of the algorithm's complexity, the question is a little bit more difficult. We can prove that at each iteration of the cycle "while" the procedure "Augmenting_paths" performs at most $2m$ iterations (each costing $O(m)$, since it is a search of the residual graph). The iterations of the cycle are $O(\log U)$, since at each iteration the value of parameter Δ is halved. It follows that the algorithm's complexity is $O(m^2 \log U)$.

Exercise

Describe an algorithm for computing the path of maximum residual capacity from s to t in the residual graph. Analyze its complexity.

6.4 *A particular case: maximum cardinality matching*

Consider a bipartite graph $G=(S,T,A)$ with the set of arcs $A \subseteq S \times T$. The problem of the maximum cardinality matching consists in determining a subset M of maximum cardinality arcs such that M does not contain more than one arc incident on the same node. The problem can be mathematically formulated in the following way:

$$\max \sum_{(i,j)\in A} x_{ij}$$
$$\sum_{(i,j)\in FS(i)} x_{ij} \leq 1 \qquad \forall i\in S$$
$$\sum_{(i,j)\in BS(j)} x_{ij} \leq 1 \qquad \forall j\in T$$
$$x_{ij} \in \{0,1\} \qquad \forall (i,j)\in A,$$

where the logical variable x_{ij} has value 1 if and only if the arc $(i,j)\in M$. The same problem can be formulated in terms of maximum flow on an appropriate graph, an example of which is shown in Figure 28.

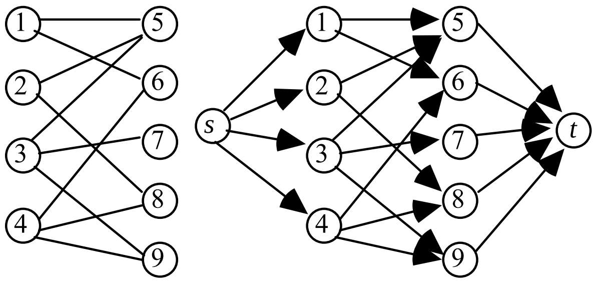

Fig. 28: bipartite graph and formulation in terms of maximum flow

To the bipartite graph – in which the nodes of sets S and T are arranged on the two central columns – we added the source node s and the sink node t. To the arcs oriented from S to T we added the arcs connecting s to all nodes of S as well as the arcs connecting all nodes of T to t. It is evident that, if arc capacities are set at 1 and we search for the maximum flow from s to t, then at most one unit of flow can transit across each node of S and each node of T. This, actually, makes the maximum flow problem equivalent to the problem of the maximum cardinality assignment on bipartite graph, where there is at most one arc going out of each node of S and at most one arc coming into each node of T. The only difference existing between the two formulations is that the maximum flow problem admits, among the different solutions, also some having fractional flow values, whereas the maximum cardinality matching problem, as it is a problem with variables of a logical type, only admits solutions in which variables are integer. However, in a maximum flow problem in which arc capacities are integer, given an optimal solution with fractional flow values, it is always possible to find an equivalent optimal solution in which all flows are integer. Therefore, in the specific case of the maximum cardinality matching problem we can find a solution in which flows take value 0 or 1.

The augmenting path algorithm takes special advantage of the fact that in the graph there are unit capacities. In fact, at each iteration the flow augments of exactly one unit. Since the maximum cardinality matching has value at most equal to the minimum between the cardinality of S and the cardinality of T, the problem can be solved in time $O(m \min\{|S|,|T|\})$.

Exercise

Apply the algorithm described in the instance of Figure 7. At each iteration, augmenting paths alternate arcs of $A^+(\bar{x})$ and arcs of $A^-(\bar{x})$, always beginning and ending with an arc of $A^+(\bar{x})$. Demonstrate that this occurs at all times for any matching problem.

7 Minimum cost flow

The minimum cost flow problem is a problem of optimization on networks of a more general kind: both the maximum flow problem and the shortest-path tree problem may be viewed as particular cases of the minimum cost flow problem. A network flow is specified by a digraph $G=(N,A)$; with each node $i \in N$ we associate a value b_i (called *balance* of the node); with each arc (i,j) we associate the unit *cost* c_{ij} and a *capacity* u_{ij} limiting the maximum quantity of flow that can transit. Balances at nodes regulate the flow on the network. A node with balance $b_i = 0$ is called a *transshipment* node because it neither requires nor provides any flow; a node with $b_i < 0$ is called an *origin* node (or *source*) because it provides some flow; a node with $b_i > 0$ is called a *destination* node (or *sink*) because it requires some flow. Later we will assume that the global network balance is null, i.e., that the sum of the b_i is equal to zero. By appropriate tricks we can always refer to a network case with null global balance.

The *minimum cost flow problem* consists in determining the flow on the arcs of the network so that all available flow leaves from sources, all required flow arrives at origins, arc capacities are not exceeded and the global cost of the flow on arcs is minimized. By employing the usual flow variables x_{ij} on arcs, we obtain the following problem's formulation:

$$\begin{aligned} \min \sum_{(i,j)\in A} c_{ij}x_{ij} & \\ -\sum_{(i,j)\in FS(i)} x_{ij} + \sum_{(j,i)\in BS(i)} x_{ji} = b_i & \quad \forall i \in N \\ 0 \le x_{ij} \le u_{ij} & \quad \forall (i,j) \in A \end{aligned}$$

If in the formulation of a minimum cost flow problem we obtain that $b_i = 0$ for each node i, then we have a *circulation* problem. In this class of problems the quantity of flow circulating in the network is not imposed by balances but is induced by costs on arcs. Note that in order to induce a flow other than zero, the cost of some arc will have to be negative.

It must be observed that the shortest-path tree problem can be seen as a minimum cost flow problem in which arc costs are given by lengths, arc capacities are unbounded and balances at nodes are: for the root of tree r we have a supply of n-1 units of flow, hence $b_r = -(n-1)$, whereas for all other nodes i we have a demand for one unit of flow, hence $b_i = 1$.

In the second and more general formulation which was previously proposed, the maximum flow problem involves a circulation problem, in fact all balances at nodes are zero. What makes profitable the fact of circulating flow, in comparison with a null solution, is the objective function. In the case of the maximum flow we can fix the cost of all arcs at 0 except for the dummy arc (t,s), for which we fix the cost at -1.

Exercise
In a minimum cost flow problem in which the sum of balances is other than 0 the formulation must be changed. Supposing that we have an excess of supply (negative sum of balances), then the flow conservation constraints of the source nodes must be changed into constraints of greater than or equal to; in fact, not all provided flow will be able to leave from sources. Wishing to refer to a problem in which all constraints are equality constraints, we should introduce appropriate slack variables. Give an interpretation of slack variables in terms of flow, taking into account that in flow problems variables are related to arcs.

7.1 Applications: road construction and communications network design

Example: road construction
Let us consider the problem of optimizing the excavation operations for the construction of a road on an undulating ground. Once the course of the road has been fixed, it is necessary to excavate the ground so as to eliminate humps and level depressions. The amount of material to be moved is quantified in number of truckloads.
After having partitioned the road into appropriate portions, we can represent the problem by using a network flow. With each road section we associate a node i, whose balance is given by the number of truckloads of material to be moved. Obviously, if in a section there is a hump we have a source (negative b_i), and if there is a soil depression we have a sink (positive b_i). Each node is connected to the nodes of adjacent sections. Since we may assume that there are no constraints on the number of truckloads moved from one node to the other, arc capacities are all equal to infinity. The price of transport between two adjacent nodes is specified by the cost c_{ij}. In this case the flow circulating on each arc represents the number of truckloads moving from one place to the other.

Example: communications network design
A set of n terminals must be connected to the central computer. For each terminal there are two possibilities: either to build a line directed to the central computer or to connect the terminal through a concentrator. There are p concentrators available; at most k terminals can be connected to each concentrator. The line connecting each concentrator to the central computer is already present. The cost of building the lines for each terminal i is given by c_{i0} for the line directed from the terminal to the central computer and by c_{ih} for the line connecting the terminal to the concentrator h. The problem of planning the minimum cost communications network can be formulated as a problem of flow on an appropriate network. In the network we have a node 0 representing the central computer, nodes 1,…, p representing concentrators and nodes p+1,…, p+n representing terminals. Nodes p+1,…, p+n have balance –1 (each of them provides one unit of flow), nodes 1,…, p are transshipment nodes, node 0 is a sink node and demands exactly n units of flow. For each terminal-node i there are p+1 outgoing arcs: one directed to node 0, with cost c_{i0}, and the others directed to concentrators, with cost c_{ih}, h=1,…, p. Further, we have an arc connecting each concentrator h to node 0, whose capacity u_{h0} is equal to k. The capacities of the other arcs are set at infinity. An "integer" optimal solution (i.e., one in which all flows take value 0 or 1) corresponds to the optimal communications network design. In this case the flow on every single arc represents a decision variable: if it is equal to 1 the corresponding line is built, if it is equal to 0 it is not built.

Note that if there exists a non-integer optimal solution, it is always possible to find an equivalent integer one.

7.2 Negative cycle elimination algorithm

As confirmed by the two preceding problems, to characterize optimal solutions is quite helpful for providing an algorithm. Let us consider a flow $\bar{x}$ being feasible for the problem. As we did for the maximum flow problem, we can define the residual graph $G_R(\bar{x})$ associated with it. The residual graph, just like in the preceding case, provides the information needed to evaluate the possible flow variations. In the minimum cost flow case the "cost" component is essential to decide whether a flow variation is advantageous, therefore with each arc of the residual graph we can associate a *residual cost* g_{ij} quantifying the effect on the objective function of a unitary variation of the flow corresponding to the residual graph's arc:

$$g_{ij} = \begin{cases} c_{ij} & \text{if } (i,j) \in A^+(\bar{x}) \\ -c_{ji} & \text{if } (i,j) \in A^-(\bar{x}) \end{cases}$$

A route on the residual graph corresponds to a possible variation of flow $\bar{x}$, according to the same rules we saw for the maximum flow, i.e., a positive variation for the flow variables corresponding to arcs of $A^+(\bar{x})$ and a negative variation for the flow variables corresponding to arcs of $A^-(\bar{x})$.

Example: residual graph

Consider the minimum cost flow problem represented in Figure 29, where a feasible solution is shown as well.

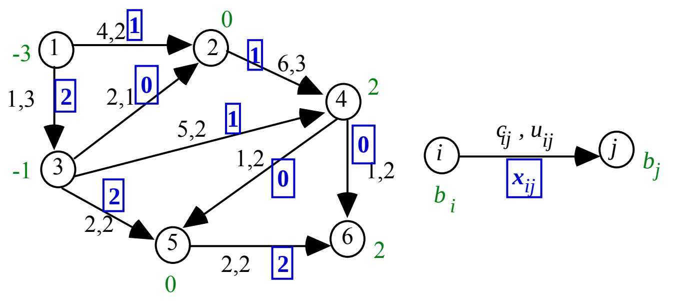

Fig. 29: minimum cost flow problem with feasible flow

The residual graph relative to the given flow is represented in Figure 30.

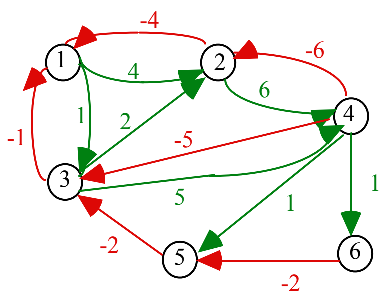

Fig. 30: residual graph

The only way of modifying the solution without violating the flow conservation constraints is to vary the flow on a cycle (or on a set of cycles) by an identical quantity θ. In fact, a flow variation on a route which does not close into itself would immediately bring to a violation of the flow conservation constraints which were satisfied by the original flow $\bar{x}$.
Considered a cycle C, the maximum practicable flow variation is given by:

$$\theta = \min\{\{u_{ij}-\bar{x}_{ij}: (i,j)\in A^{+}(\bar{x})\cap C\}, \{\bar{x}_{ij}: (i,j)\in A^{-}(\bar{x})\cap C\}\}.$$

Note that, due to how the residual graph has been built, θ>0. The flow is updated according to the following rules:

$$x'_{ij} = \begin{cases} \bar{x}_{ij} + \theta & \text{if } (i,j) \in A^{+}(\bar{x}) \cap C\} \\ \bar{x}_{ij} - \theta & \text{if } (j,i) \in A^{-}(\bar{x}) \cap C\} \\ \bar{x}_{ij} & \text{else} \end{cases}$$

The variation of the objective function is the following:

$$\sum_{(i,j)\in A} c_{ij}x'_{ij} - \sum_{(i,j)\in A} c_{ij}\bar{x}_{ij} = \sum_{(i,j)\in C} g_{ij}\,\theta.$$

This means that there is an improvement in the solution value only if cycle C has sum of negative residual costs. We can try the reverse as well, i.e., if there exist no negative cycles in the residual graph, then the flow is not improvable, as suggested by the following flow decomposition theorem. Before discussing the theorem, let us introduce a definition. As previously seen, a flow x is called a circulation if the quantity of flow coming into each node is equal to the quantity of outgoing flow, that is if:

$$-\sum_{(i,j)\in FS(i)} x_{ij} + \sum_{(j,i)\in BS(i)} x_{ji} = 0 \qquad \forall i\in N$$

A circulation x is said to be *simple* if arcs i(i,j) in which x_{ij}>0 detect a cycle. Note that, because of the flow conservation, the quantity of flow on the arcs of a simple circulation is constant.

Flow decomposition theorem

Consider a minimum cost flow problem on graph $G=(N,A)$.

$$\begin{aligned} \min \sum_{(i,j)\in A} c_{ij}x_{ij} & \\ -\sum_{(i,j)\in FS(i)} x_{ij} + \sum_{(j,i)\in BS(i)} x_{ji} = b_i & \quad . \ \forall i\in N \\ 0 \le x_{ij} \le u_{ij} & \quad . \ \forall (i,j)\in A \end{aligned}$$

Let there be a feasible flow $\bar{x}$ and an optimal flow x^*. There exist k simple circulations $x^1, \ldots, x^k$ allowing to transform $\bar{x}$ into x^*, i.e., such that $\bar{x} + x^1 + \ldots + x^k = x^*$.

<u>Proof</u>

Consider $\hat{x} = x^* - \bar{x}$: since $\bar{x}$ and x^* are both feasible, we have:

$$-\sum_{(i,j)\in FS(i)} \hat{x}_{ij} + \sum_{(j,i)\in BS(i)} \hat{x}_{ji} = 0 \qquad \forall i \in N.$$

We construct a graph $\hat{G}$ starting from G and introducing the arcs (i,j) and (j,i) for each arc (i,j) of A. When we obtain a $\hat{x}_{ij} < 0$ we transform $\hat{x}$ by removing the flow from arc (i,j) and bringing it back, with changed sign, on arc (j,i), which operation does not affect the flow conservation. We can easily verify that $\hat{x}$, so transformed, is a circulation on $\hat{G}$. Now we have to demonstrate that the circulation $\hat{x}$is *decomposable* in k simple circulations.
If $x^* = \bar{x}$, $\hat{x}= 0$, hence $k = 0$, and the theorem is banally valid.
Otherwise, there exists an arc (i,j) for which $\hat{x}_{ij}>0$. Due to flow conservation, there will exist at least one arc (j,h) for which $\hat{x}_{jh}>0$. By successively repeating the same reasoning, after at most n arcs we reach again a node which had already been visited, thus closing a cycle C_1. Let θ_1 be the minimum quantity $\hat{x}_{hl}$ on arcs $(h,l)\in C_1$, so we define x^1 as follows: $x^1_{hl} = \theta_1$ if $(h,l)\in C_1$ and 0 else. Now we update $\hat{x}$ by subtracting θ_1 from each arc of C_1. Note that, according to how we determined θ_1, $\hat{x}_{ij}$ has set itself at zero on at least one arc (i,j). We can repeat this procedure until – in a finite number k of steps – $\hat{x}$ is wholly null.
So, we may conclude that $x^*-\bar{x}= x^1 + \ldots + x^k$.

♦

The preceding theorem has two important and simple consequences.

Corollary
The number k of simple circulations is limited by the number m of arcs of G.

Corollary
The residual cost of cycles $C_1, \ldots, C_k$ is smaller than or equal to zero.

Hence, it clearly appears that an optimality condition is:

The feasible flow $\bar{x}$ is optimal if and only if $G_R(\bar{x})$ does not contain cycles whose sum of residual costs is negative.

Starting from the optimality condition stated above and from the discussion previous to it, we can provide a simple solution algorithm for the minimum cost flow problem which is based on the next flow update until the residual graph contains no negative cycles. The algorithm receives as input the network G and a feasible flow. The algorithm internally uses a procedure detecting negative cycles on the residual graph. This procedure can be efficiently carried out on the basis of shortest-path computation, as explained in paragraph 5.10. The flow update is performed according to the rules discussed above.

```
Procedure Negative_cycle_elimination (G,x);
  begin
    Gr:=Residual_graph(x);
    while ∃ negative cycle C in Gr do
    begin
      update_flow(x,C); update residual_graph(x)
    end
  end.
```

Example: an iteration of the algorithm

If we observe the residual graph of Figure 30, we may detect various negative cycles: 4-2-1-3-4 with total cost -4, 1-3-2-1 with cost -1. The algorithmic scheme we presented does not suggest any criterion by which cycles can be chosen. In the literature there are various algorithms that differ according to the criterion by which cycles are chosen. In the considered case we choose the first cycle. θ=min{1, 1, 3-2, 2-1}=1. The flow update involves an augmentation of one unit of flow along the arcs of $A^+(\bar{x})$ (1,3) and (3,4), and a diminution of one unit for the flow on the arcs of $A^-(\bar{x})$ (2,4) and (1,2).

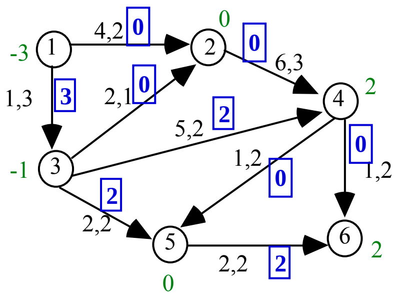

Fig. 31: updated flow

Exercise

Verify that the flow of Figure 31 is optimal. Considering the cycle 1-3-2-1, try to execute the negative cycle elimination algorithm.

7.3 Feasible starting flow

In order to apply the algorithm we need a method enabling to generate a feasible starting flow, provided that one exists. The computation of a feasible flow can be performed in polynomial time by means of a maximum flow algorithm on an appropriate graph. Let us consider an *auxiliary* graph $G'=(N\cup\{s,t\},A'\}$ in which A, in addition to all arcs of A, also includes arcs connecting source s to nodes i having negative balance (flow sources in G) and arcs connecting nodes having positive balance (flow destinations) to sink t. Arcs of type (s,i) have capacity $u_{si}=-b_i$, whereas arcs (i,t) have capacity $u_{it}=b_i$. All other arcs have unchanged capacity with respect to the original graph G. At this point node balances are set at zero. A feasible flow (if one exists) can be found by solving a problem of maximum flow from s to t. If the value of the maximum flow is equal to $-\sum_{i:b_i<0} b_i$, then there exists a feasible flow, and it is precisely given by the maximum flow which is found. Note that in this case the cut found by Edmonds-Karp algorithm is $N_s=\{s\}$ $N_t=N\cup\{t\}$; otherwise, the cut includes in N_s nodes of N as well.

Example: auxiliary graph

Consider the minimum cost flow problem of Figure 29. We must add nodes *s* and *t*, arcs from *s* to supply nodes 1 and 3 (having capacity 3 and 1, respectively) and arcs from demand nodes 4 and 6 (both having capacity 2). If, when solving the maximum flow problem, we obtain a flow 4, then the flow on inner arcs is feasible.

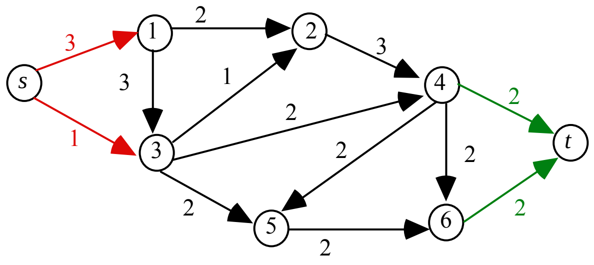

Fig. 32: auxiliary graph for the problem of Figure 29

7.4 Complexity analysis

Now let us analyze the complexity of the negative cycle elimination algorithm which was performed without adopting any specific rule for the choice of the negative cycle to be removed, in case there are more than one alternative. We suppose that all costs and capacities of network *G* are non-negative integers and we call C_{max} and U_{max} the maximum cost and the maximum capacity of the graph arcs, respectively. Since the choice of the initial feasible solution is not in the least dictated by cost criteria, at the beginning in the worst case the cost of the provided solution is given at most by $mC_{max}U_{max}$. On the basis of a similar reasoning, we may suppose as well that the optimal solution in the extreme case has cost 0. Observing that at each iteration of the algorithm the flow varies by at least one unit and that this induces a decrease in the value of the objective function by at least one unit, we will have to perform, in the worst case, $O(mC_{max}U_{max})$ iterations. Knowing that negative cycles can be recognized in $O(nm)$, the algorithm's complexity is $O(nm^2C_{max}U_{max})$.
It can be demonstrated that, in case of a particular choice of the negative cycle, the algorithm is polynomial.

Exercise
The augmenting path algorithm for the maximum flow problem can be interpreted as a negative cycle elimination algorithm. Discuss the details.

**7.5 Shortest-path sequence algorithm*

Now we consider another algorithm for the minimum cost flow problem which works in an incremental way, i.e., instead of starting from a feasible solution, it directly tries to build the optimal solution by sending the flow from source nodes to destination nodes. The algorithm is based on an optimality condition which is different and yet equivalent to the one previously seen. Let us recall the concept of reduced cost already mentioned in the case of the shortest-path tree. Given a potential π associated with graph nodes, the reduced cost of an arc $(i,j)\in A$ is defined as follows:

$$\bar{c}_{ij} = c_{ij} + \pi_i - \pi_j.$$

Given a path from node x to node y P_{xy}, the sum of reduced costs is given by:

$$\sum_{(i,j)\in P_{xy}} \bar{c}_{ij} = \sum_{(i,j)\in P_{xy}} c_{ij} + \pi_x - \pi_y.$$

Hence, the sum of a path's reduced costs is equal to the sum of costs except for a constant merely depending on the path's endpoints and not on the arcs composing it.
Because in a cycle C endpoints coincide, the sum of a cycle's reduced costs is equal to the sum of costs:

$$\sum_{(i,j)\in C} \bar{c}_{ij} = \sum_{(i,j)\in C} c_{ij}.$$

This implies that if a cycle is negative with respect to reduced costs, it is also negative with respect to original costs.
An optimality condition equivalent to the one concerning the non-existence of negative cycles on the residual graph is the following:

The feasible flow $\bar{x}$ is optimal if and only if there exists a potential π such that $\bar{c}_{ij} \geq 0$ for each arc $(i,j)\in A(\bar{x})$.

We prove that the two optimality conditions are equivalent. Let us suppose that the condition on reduced costs is valid:

$$\bar{c}_{ij} \geq 0 \qquad \forall (i,j)\in A(\bar{x}),$$

then for each cycle C in $G_R(\bar{x})$

$$\sum_{(i,j)\in C} \bar{c}_{ij} \geq 0$$

but, given that the sum of reduced costs is equal to the sum of original costs, we have

$$\sum_{(i,j)\in C} c_{ij} \geq 0.$$

Otherwise, let us suppose that $G_R(\bar{x})$ does not contain negative cycles: in this case we can compute the shortest-path tree taking as root any node. The shortest-path algorithm comes to an end and provides optimal labels $d[i]$. If we set $\pi_i = d[i]$, then for each arc of $A(\bar{x})$ Bellman's conditions are valid:

$$c_{ij} + \pi_i - \pi_j = \bar{c}_{ij} \geq 0 \qquad \forall (i,j)\in A(\bar{x}).$$

Now we can provide an algorithm on the basis of the new optimality conditions we have found. In the proposed algorithm we start from an unfeasible solution x for which, nevertheless, the optimality conditions hold; at each iteration we keep the optimality conditions but we try to reduce the unfeasibility of solution x.

A solution x is called a *pseudoflow* if it respects capacity constraints but not necessarily constraints concerning flow conservation at nodes:

$$-\sum_{(i,j)\in FS(i)} x_{ij} + \sum_{(j,i)\in BS(i)} x_{ji} - b_i = \overline{g}_I \qquad \forall i\in N.$$

The quantity $\overline{g}_i$ is called *excess* of node i. If $\overline{g}_i = 0$ the node is *balanced*, if $\overline{g}_i<0$ the node has a flow *defect* and if $\overline{g}_i > 0$ the node has a flow excess.

The main purpose of the algorithm we are going to consider is to send flow from excess nodes to defect nodes. Obviously, this must be done while keeping the optimality conditions on reduced costs of the residual graph. The critical point reveals to be the following. If we augment the flow on an arc (i,j) by a quantity greater than zero, at the next iteration the residual graph will contain arc (j,i), whose reduced cost is:

$$\overline{c}_{ji} = c_{ji} + \pi_j - \pi_i = -\overline{c}_{ij}.$$

A similar reasoning can be made for negative variations. Consequently, it is evident that operations of flow variation are particularly delicate as they threaten to invalidate the optimality conditions the algorithm has to keep invariant all along its execution. So, before performing flow variations on arcs we have to ensure conditions such that no arcs of negative reduced cost are created; this means that before performing a flow variation we have to take care to update potentials so that the reduced costs of arcs on which we augment or diminish the flow are null. Of the utmost importance for the algorithm's functioning is the following observation.

Given a pseudoflow x and a potential π respecting optimality conditions on reduced costs

$$\overline{c}_{ij} \geq 0 \qquad \forall (i,j)\in A(\overline{x}),$$

if we consider optimal labels $d[i]$ of a shortest-path tree with root s on residual graph $G_R(\overline{x})$, where s is any excess node, and we modify the potential as follows:

$$\pi'_i = \pi_i + d[i] \qquad \forall i\in N,$$

then optimality conditions on reduced costs continue to be valid.

In fact, the new reduced costs are given by:

$$c'_{ij} = c_{ij} + \pi'_i - \pi'_j,$$

and for label optimality what follows is valid

$$d[j] \leq d[i] + \overline{c}_{ij} = d[i] + c_{ij} + \pi_i - \pi_j,$$

hence

$$c_{ij} + \pi_i + d[i] - \pi_j - d[j] \geq 0$$

that is $c'_{ij} \geq 0$.

Note that, due to the proposed potential update, all arcs of the shortest-path tree have a reduced cost which is exactly equal to zero, so on these arcs flow can be varied. On the basis of the fundamental (though not restrictive) hypothesis that all costs are greater than or equal to zero, the algorithm becomes:

```
Procedure Shortest_path_sequence (G,x,$\pi$);
   begin
      D=ø; O=ø;
      for each (i,j)$\in$A do $x_{ij}$:=0;
      for each i$\in$N do
      begin $\pi_i$:=0; $g_i$:=-$b_i$; if $g_i$<0 then D:=D$\cup${i} else if $g_i$>0 O:=O$\cup${i} end;
      while O$\neq$ø do
      begin
         select i from O;
         SPT(i,P,d,G(x));
         for each j$\in$N do $\pi_j$:=$\pi_j$+d[j];
         let $P_{ik}$ be the shortest path from i to k with k$\in$D;
         $\theta$:=min{$g_i$, -$g_k$, min{$r_{jh}$, (i,h)$\in P_{ik}$};
         augment_flow($\theta$,$P_{ik}$); update (O,D);
      end
   end.
```

The algorithm, after the initialization phase in which it sets the initial pseudoflow and the potential at zero, seeks a shortest path from an excess node to a defect node. Before sending the maximum possible quantity of flow, potentials are updated so that all arcs of the detected path have a null reduced cost. The maximum quantity of flow that can be sent is given by the minimum between the excess, the defect and the path's residual capacity computed like in the case of the negative cycle elimination algorithm or of the augmenting path algorithm for the maximum flow. Note that after the update of potentials it is possible to send flow on more than one path. The algorithms of this class differ from others precisely because of these aspects. The algorithm ends when there are no more excess nodes (or defect nodes), or else when defect nodes are no longer reachable from excess nodes.

Let us analyze the algorithm's complexity. We call

$$B = \sum_{i \in O} \bar{g}_i$$

the quantity of excess to be sent from origin nodes. Supposing we have integer capacities, at each iteration at least one unit of flow is sent, so the initial quantity of excess is diminished by at least one unit. Hence, we will have no more than B iterations. At each iteration the most expensive operation is given by the computation of the shortest-path tree in a graph with costs greater than or equal to zero. Therefore, the complexity is given by $O(B\ S(n,m))$, where $S(n,m)$ is the complexity of the shortest-path algorithm which was used.

Exercise
In case the network contains arcs with negative costs, the algorithm of successive shortest paths cannot be directly applied because the null pseudoflow and the null potential do not respect the optimality conditions. How can the starting solution be modified in order to ensure that the optimality conditions hold?

7.6 Minimum cost assignment

Given a bipartite, complete (i.e., $A=S\times T$) graph $G=(S,T,A)$, with $|S|=|T|=n$ and c_{ij} weights on arcs $(i,j)\in A$, the minimum cost assignment problem consists in finding a subset of arcs M such that exactly one arc of M is incident on each node and that the sum of weights of arcs of M is minimized. Using x_{ij} logical variables associated with arcs, as in the case of the maximum cardinality matching, the problem can be formulated as follows:

$$\begin{aligned}
\min \ & \sum_{(i,j)\in A} c_{ij}\, x_{ij} \\
& \sum_{j\in T} x_{ij} = 1 && \forall i\in S \\
& \sum_{i\in S} x_{ij} = 1 && \forall j\in T \\
& x_{ij} \in \{0,1\} && \forall (i,j)\in A.
\end{aligned}$$

The problem can be interpreted in terms of minimum cost flow. We associate with each node of S a supply of one unit of flow (b_i=-1, $\forall i\in S$) and with each node of T a demand for one unit of flow (b_j=1, $\forall j\in S$). The cost of each arc (i,j) is exactly the weight c_{ij}, whereas the capacity is unbounded.

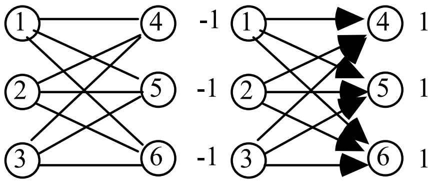

Fig. 33: transformation of the bipartite graph into the network flow

It is evident that, for the same reasons discussed in the case of the maximum cardinality matching problem, the two problems are equivalent. Therefore, by optimally solving the minimum cost flow problem we obtain the optimal solution of the minimum cost assignment problem. The algorithm of successive shortest paths can derive special benefit from the problem's characteristics. In fact, at each iteration of the algorithm it is detected a path between a node of S being not assigned yet (called *exposed origin*) and a node of T being not assigned yet (called *exposed destination*). Along the detected path, which is not necessarily composed of only one arc, we can send only one unit of flow. This means that at each iteration the number of exposed origins (or exposed destinations) will decrease by one unit. Therefore, the number of iterations is n and the algorithm's complexity is $O(n^3)$, if we compute the shortest-path tree by using Dijkstra's algorithm implemented with an unordered list.

Exercise
Solve the problem of Figure 33 with the weights given in Table 4.

c_{ij}	**4**	**5**	**6**
1	15	2	5
2	8	3	8
3	5	4	2

Table 4

8 Multi-commodity flow problems

The flow problems we have seen so far are characterized by the fact that the flow circulating in the network is homogeneous. However, several practical applications cannot be based on this restrictive hypothesis and have to consider distinct flows contributing to the utilization of common resources. This typically happens in the case of distribution of goods. Consider the example illustrated in Figure 34.

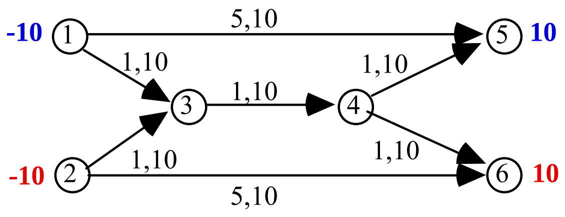

Fig. 34: example of multi-commodity flow; costs and capacities are indicated on arcs

Node 1 supplies 10 units of commodity A, node 2 supplies 10 units of commodity B, nodes 5 and 6 demand 10 units of commodity A and 10 units of commodity B, respectively. Therefore, the flows of the two commodities must be distinct, as we cannot send commodity B to node 5 nor commodity A to node 6. This, however, does not imply solving the flow problems for commodity A and for commodity B separately and then recompose the solution by superimposing the two flows, since they contribute to the utilization of network resources such as arc capacities. Denoting by *K* the set of commodities, the problem can be formulated as follows:

$$\min \sum_{(i,j)\in A} \sum_{k\in K} c_{ij} x_{ij}^k$$

$$-\sum_{(i,j)\in FS(i)} x_{ij}^k + \sum_{(j,i)\in BS(i)} x_{ij}^k = b_i^k \qquad \forall i\in N,\ \forall k\in K$$

$$0 \le \sum_{k\in K} x_{ij}^k \le u_{ij} \qquad \forall (i,j)\in A$$

where variables x_{ij}^k indicate the flow of commodity *k* on arc (*i*,*j*). Note that, in comparison with a minimum cost flow problem, in the multi-commodity flow problem we repeat the flow conservation constraints for each commodity *k*, while the role of flow variables in the objective function and in the capacity constraints is now played by the sum, applied to all commodities, of flows on the arc. In fact, the problem's structure makes it necessary to find flows for all commodities simultaneously. The solution of this problem, although based on flow-problem features, makes use of Linear Programming methods and, more generally, of mathematical programming methods. This subject will be more deeply examined in the next chapters.

Now let us analyze a particular class of multi-commodity flow models, which is used in solving telecommunications network routing or planning problems. Consider a short example concerning a small intranet network with 5 nodes and 16 unidirectional links, as sketched in Figure 35; for more simplicity, the figure shows arcs with double orientation, which in the model correspond to two arcs having opposite directions.

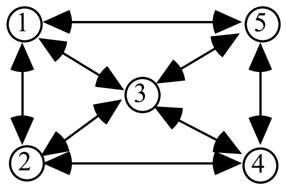

Fig. 35: the scheme of a small intranet network

In addition to the forwarding costs c_{ij} for each information unit and to the maximum capacities u_{ij} on each link (i,j), the problem's data include the information quantities d_{st} to be sent between each pair of nodes (s,t). The latter information is summed up in the so-called *origin/destination matrix*. Clearly, we are in front of a flow problem, but, equally clearly, the flow circulating in the network is not homogeneous; in fact, the flow of each origin/destination pair (s,t) must be separated from all others. So, we have to make use of a multi-commodity flow formulation in which each commodity k corresponds to a pair (s_k,t_k) such that $d_{s_kt_k}$ is other than 0. The model provided above, which introduces a variable for each arc and origin/destination pair, correctly formulates the minimum cost information routing problem. Yet, this formulation reveals a small defect: the number of introduced variables, especially if the origin/destination matrix is very dense, is very high. In Figure 35, for example, if the origin/destination matrix contains elements other than zero for all pairs of nodes, we have 16×20=320 variables. Therefore, wishing to use the version of Xpress limiting the number of variables to 300, it would be impossible to solve the model provided above.

We can try to formulate the problem differently. The central idea of the model we are proposing is to specify the flow no longer on single arcs, but on entire paths from each possible origin to each possible destination. We denote by x_p the flow variable along path p. We denote by P the set of all paths useful for solving the problem; for each origin/destination pair (s_k,t_k) we define a subset $P_k \subset P$ containing all paths going from s_k to t_k. Obviously, for each k in K, subsets P_k constitute a partition of the set of all paths P. The cost of a path p is given by the sum of costs of the arcs composing it:

$$c_p = \sum_{(i,j)\in p} c_{ij}.$$

The problem's formulation, which is called *path formulation*, becomes:

$$\min \sum_{k\in K} \sum_{p\in P_k} c_p x_p$$

$$\sum_{p\in P_k} x_p = d_{s_k t_k} \qquad \forall k\in K$$

$$\sum_{p\in P:(i,j)\in p} x_p \leq u_{ij} \qquad \forall (i,j)\in A$$

$$x_p \geq 0 \qquad \forall p\in P$$

The first set of constraints corresponds to the flow conservation constraints of the first formulation. The second set of constraints concerns arc capacities: the flow on an arc (i,j) is given by the sum of flows on all paths containing the arc (i,j).

A possible objection to the path formulation is that the defect concerning the number of variables was not solved, but rather, the fact of introducing a variable for each path might have made it worse because the number of paths connecting two nodes is exponential in the graph's dimension. Actually, as we shall see, it is not necessary to consider all possible paths between each pair of nodes but only a subset, and this subset can be dynamically generated during the problem's solution every time it is detected some path being particularly profitable from the viewpoint of the objective function. In order to adjust such a technique we need the tools provided by Linear Programming.