OPERATIONAL RESEARCH: Combinatorial Optimization

Keywords:
optimization, combinatorial, algorithms, graphs, computer science, data structures, complexity.

Jacques Carlier

Dritan Nace

April 2023

0. INTRODUCTION : What is Operational Research ?

The mathematics components of operational research are frequently minimized. But from the beginning, its main focus has been on finding solutions to real-world issues that pose difficulties for common sense. That is why it seemed necessary to us, in this introduction, to define Operational Research. For this purpose, we rely on the writings of Robert FAURE.

Combinatorial problems are challenges in the common sense. Therefore, they are studied, but they remain unknown even to many computer scientists. Indeed, some believe in the absolute power of the computer and do not worry about issues of data size or algorithmic complexity. Others make a mountain out of so-called NP-hard problems. However, my already long struggle against combinatorial problems has taught me that the real situation is much more complex, especially from a practical point of view. Some of them are indeed simple and rely on a heuristic, such as simulated annealing or genetic method. Others, on the contrary, require detailed studies and specific programs. But, in any case, human intervention is crucial. It is necessary to know how to exploit the specificities of a problem and it is a mistake to believe in the automaticity of its resolution.

0.1. Operational research :

Let's start by quoting Robert FAURE who was one of the main initiators of Operational Research in France...

0.1.1. The practical nature of Opeartional Research :

"Operational Research has been, remains, and will continue to be the art of quickly intervening on behalf of a specific economic entity (individual or community) in a difficult situation in order to try to improve its outcome".

0.1.2. Heuristics and Interactive Processing:

" Since its inception, operational research has developed heuristic techniques that, while not able to produce the formal optimum, can nonetheless produce useful results.".

0.1.3. Principles of Operational Research:

"The Operational Research team, especially the coordinator, must absolutely avoid considering that it is up to them to conclude their study with a decision"... to arrive at Bernard Roy, one of its current most prominent personalities, who proposes the term "Decision Analysis".

0.2. Some Operational Research Problems:

0.2.1. Discrete Combinatorial Problems:

We illustrate this with two examples: the traveling salesman problem and the minimum spanning tree problem. In the traveling salesman problem, a VRP must visit a certain number of cities while minimizing the distance traveled. In the figure below, the traveler must visit 18 cities starting from Paris. This problem is modeled by a valued graph whose vertices are the cities and edges are the connections between these cities. These edges are valued by the kilometric distances.

We must search for a Hamiltonian cycle of minimum value in this graph. There are 17! possible circuits. More generally, if there are N cities, there are N-1! circuits. It is generally impossible to

enumerate them. That is why this problem is a challenge to common sense. We will see during this course that the traveling salesman problem is probably of exponential complexity (it is NP-hard).

Figure 0.1 Map of cities in the TSP

In the minimal spanning tree problem, N sites must be connected at minimum cost so that each site can communicate with all the others. We will see that we need to search for a minimal spanning tree, and that this problem is an easy one because it can be solved by a polynomial algorithm. An example of a graph is shown in Figure 0.2, along with its minimal spanning tree in Figure 0.3. It was obtained by successively retaining the edges with the smallest costs, provided that they are "useful".

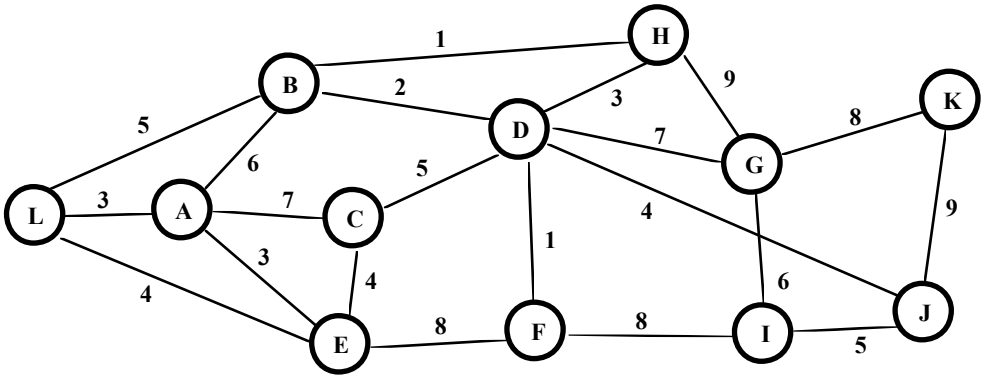


Figure 0.2 : Graph

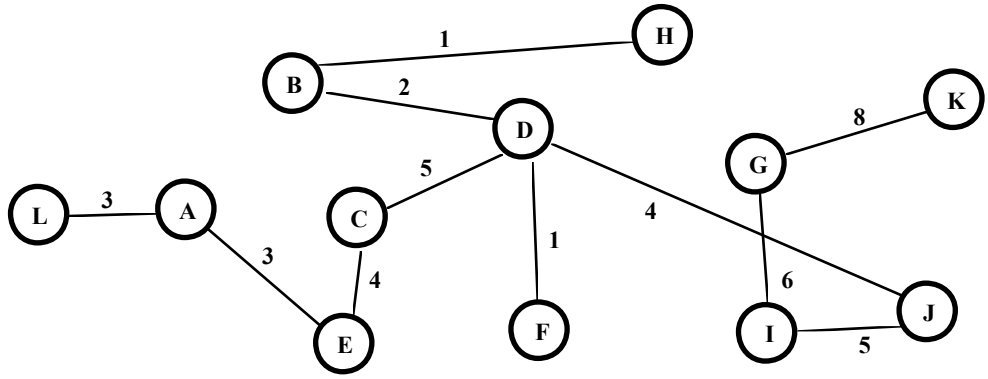


Figure 0.3 : Minimal tree

0.2.2. Continuous combinatorial problems:

A country wants to buy weapons and approaches an international arms dealer who has stolen or purchased stocks. The dealer offers 2 types of lots. The first type of lot contains 100 machine guns, 200 bulletproof vests, 5 light armored vehicles, and 50 bazookas. The second type of lot contains 50 machine guns, 100 bulletproof vests, 10 light armored vehicles, and 100 bazookas. A type 1 lot costs p1 francs and a type 2 lot costs p2 francs. The country wants to buy at least 1000 machine guns, 2500 bulletproof vests, 30 light armored vehicles, and 250 bazookas. If x1 is the number of type 1 lots and x2 is the number of type 2 lots that it buys, then it must be solve by the following linear program:

Minimise $p_1 x_1 + p_2 x_2$

Under the constraints :

$$100x_1 + 50x_2 \geq 1000$$
$$200x_1 + 100 x_2 \geq 2500$$
$$5x_1 + 10x_2 \geq 30$$
$$50x_1 + 100 x_2 \geq 250$$

$x_1 \geq 0$ and $x_2 \geq 0$ where x_1 and x_2 integers

We talk about a linear program because the constraints and the objective function are linear. This program is discrete because the variables are assumed to be integers. This integrality of variables makes the resolution difficult. That is why we relax the integrality constraint and assume the variables

to be real, which makes the problem easy. It is therefore easy (polynomial) in the continuous case and difficult (NP-hard) in the discrete case. To obtain an integer solution, we can, for example, retain the integer parts of the continuous solution. But we will not have the optimal solution.

In general, for functional optimization under constraints, we talk about mathematical programming. An example is provided by the problem that Queen Dido had to solve during the foundation of Carthage, which is: what is the geometric shape with a given perimeter that has the largest surface area? The answer is a circle. However, it should be noted that most mathematical programming problems are difficult. These problems are studied in the course RO04, where the simplex method is presented, which allows for the treatment of large continuous linear programs.

0.2.3. Random problems:

A random phenomenon occurs at the checkouts of a supermarket. An observer was able to measure the frequency of the number of customers who arrive per minute for 30 minutes.

Number of customers per minute	0	1	2	3	4	5	6
Observation frequency	4	8	8	6	3	1	0

He also observed the distribution of service times:

Service times	Number of customers
Less than 1 mn	24
1 to 2 mn	14
2 to 3 mn	8
3 to 4 mn	5
4 to 5 mn	3
5 to 6 mn	2
6 to 7 mn	1
7 to 8 mn	1
8 to 9 mn	1

These statistics allow for estimating arrival and service laws. Here, for example, we have Poisson arrivals and exponential services. It is necessary to find a compromise between customer waiting time and the number of open checkouts. Other random problems concern inventory, equipment renewal, and reliability. These problems are studied in UV RO05, where Markov chains and queuing theory are presented in particular.

0.2.4. Work scheme of an operational research team:

We have three steps. The first step aims to model the problem and determine the criteria. The second step is the solution, that is, the determination of a solution that appears good (we will say abusively optimal). The third step is a discussion with the decision maker and, if necessary, a return to the first step.

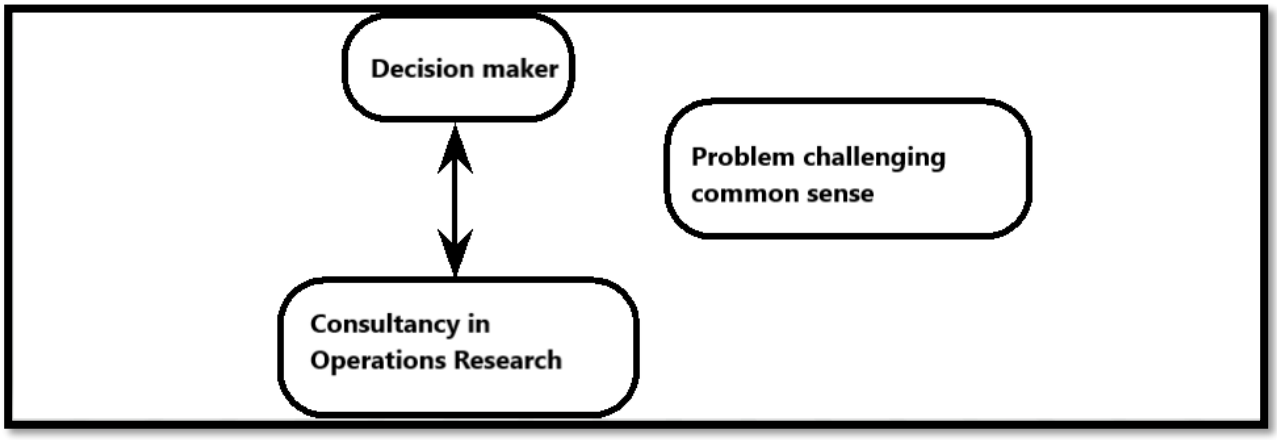


Figure 0.4 : Operating scheme of a team.

0.3. Summary:

The Operations Research:

- deals with a practical problem;
- has a limited objective (this application);
- requires a toolbox (algorithms and data structures, combinatorial optimization, graphs, complexity, linear and mathematical programming, stochastic processes, probability and statistics, multicriteria methods...);
- is multidisciplinary (Mathematics, Computer Science, Economics);
- is common (Linear Programming, PERT, ...);
- helps in decision making.

0.4. Combinatorial problems :

Among the challenges to common sense, there are therefore combinatorial problems for which it is, a priori, impossible to enumerate. However, easy problems are distinguished from difficult problems.

0.4.1. Easy problems :

It is the case for problems of very small size (you need to enumerate!), for strongly constrained problems (enumeration is sufficient!), for weakly constrained problems (a heuristic), and finally for problems solved polynomially!

0.4.2. Difficult problems :

My practical experience allows me to assert that difficult problems are frequent and that to solve them, multiple tools must be used (serial methods, simulated annealing, tree methods, dynamic programming, A* algorithm...). It should be added that for some of them, the most effective current method is to replicate human expertise.

0.4.3. Automatic methods :

It doesn't exist yet!) 40 years ago, GOMORY discovered his famous cuts for Integer Linear Programming (ILP). However, many combinatorial problems can be modeled as an ILP. At the time, it was believed that combinatorial problems could be solved. Hope was disappointed! 40 years later, we still don't know how to automatically solve combinatorial problems. But theoretically, it is not excluded that this could be possible in the future. This means that NP-hard problems could be solved "practically" in polynomial time. A great algorithmic breakthrough would be needed.

0.4.4. Semi-Automatic methods:

It is meaningless to model a combinatorial problem without additionally describing its resolution procedure, more precisely excellent evaluations or reasonable constraints. Hence the idea of creating programming languages adapted to combinatorics and including tree traversal with automatic choices.

The advantage is being able to program very quickly, at least for the specialist of such a language. In my opinion, such a tool remains ambitious because of the additional costs related to the rigidity of such a system and the weakness of their data structures.

0.4.5. Specific programs :

I will mention tree-based methods, polyhedral methods, dynamic programming, but also heuristics. The experience of solving combinatorial problems shows the crucial importance of the following points to build an effective method:

- an appropriate modeling;
- the complexity of algorithms and data structures;
- proximity to the machine (procedural language, for example);
- proximity to the problem (if we do not have excellent evaluations, they are useless).

(1) In fact, every model is a simplification of reality, and when modeling, one should seek to obtain the most representative model that can be solved satisfactorily!

0.5. Conclusions :

It would be a joke to claim to teach all of combinatorial optimization in just sixty hours (including lectures and exercises). We will not make such a claim. We will focus on the main ideas and presentation of the most fundamental algorithms. Indeed, these algorithms are the basic tools for more advanced methods. Our objectives are to raise awareness of the complexity of problems, the danger of combinatorics, and the usefulness of graphs for modeling. Hopefully, this will prevent you from designing beautiful models that are perfect for small school examples but unusable for real problems in the field.

BIBLIOGRAPHY

[1] A. ALJ, R. FAURE, Guide de la Recherche Opérationnelle, 2 tomes, MASSON, 1990.

[2] R. PENROSE, L'esprit, l'ordinateur et les lois de la physique, INTEREDITIONS, 1992.

[3] B. ROY, "Recherche Opérationnelle et Aide à la Décision", Discours de remerciements à l'occasion de la remise du diplôme de Docteur Honoris Causa de l'Université de POZNAN, 1992.

[4] M. GONDRAN; M. MINOUX “Graphes et Algorithmes” Eyrolles 1985.

[5] ROSEAUX. “Exercices corrigés de Recherche Opérationnelle” 3 Tomes. MASSON, 1982.

[6] M.R. GAREY, D.S JOHNSON, “Computers and Intractability” W. H. FREEMAN, San Francisco, 1979.

1. GRAPHS:

1.1. Why graphs? :

Graphs are invaluable tools for modeling and solving numerous real-world problems. Indeed, they allow, on the one hand, to guide intuition during reasoning, and on the other hand, to connect with known results from graph theory. We define them briefly below and illustrate their usefulness with some examples.

A **directed graph** is a pair **G = (X, U),** where X is a set whose elements are called vertices, and U is a subset of X x X whose elements are called arcs.

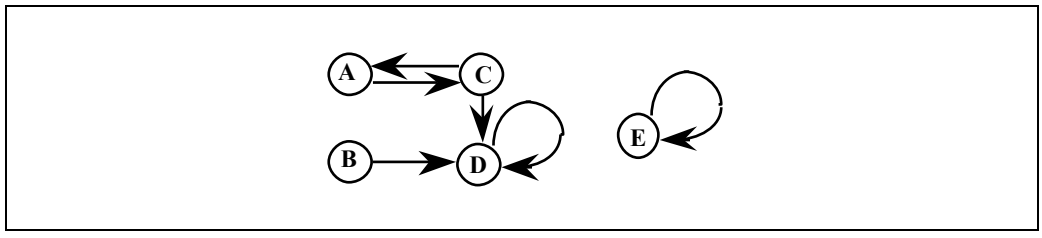


Figure 1-1 Directed graph

Example 1 :

A first example of a graph is illustrated in Figure 1.1. It is given by: X = {A, B, C, D , E} and U = {(A , C) (C ,A) (C ,D) (B ,D) (D ,D) (E , E)}.

Example 2 :

We have 3 jars with respective capacities of 8, 5, and 3 liters. Initially, the largest jar is full, and the other two are empty. We want to reach the situation where the two largest jars contain 4 liters each, under the constraint that when we pour from one jar to another, we either fill the receiving jar completely or empty the pouring jar completely.

To this problem, we associate a graph whose vertices represent the states of the system. A vertex is a triplet (i, j, k) where i, j, and k are the contents of the 3 jars. The edges correspond to the possibilities of transition between states (see Figure 1.2).

An undirected graph is a pair G = (X, E), where X is a set of vertices and E is a set of pairs of vertices called edges. We have a multigraph when we allow the simultaneous existence of multiple identical pairs.

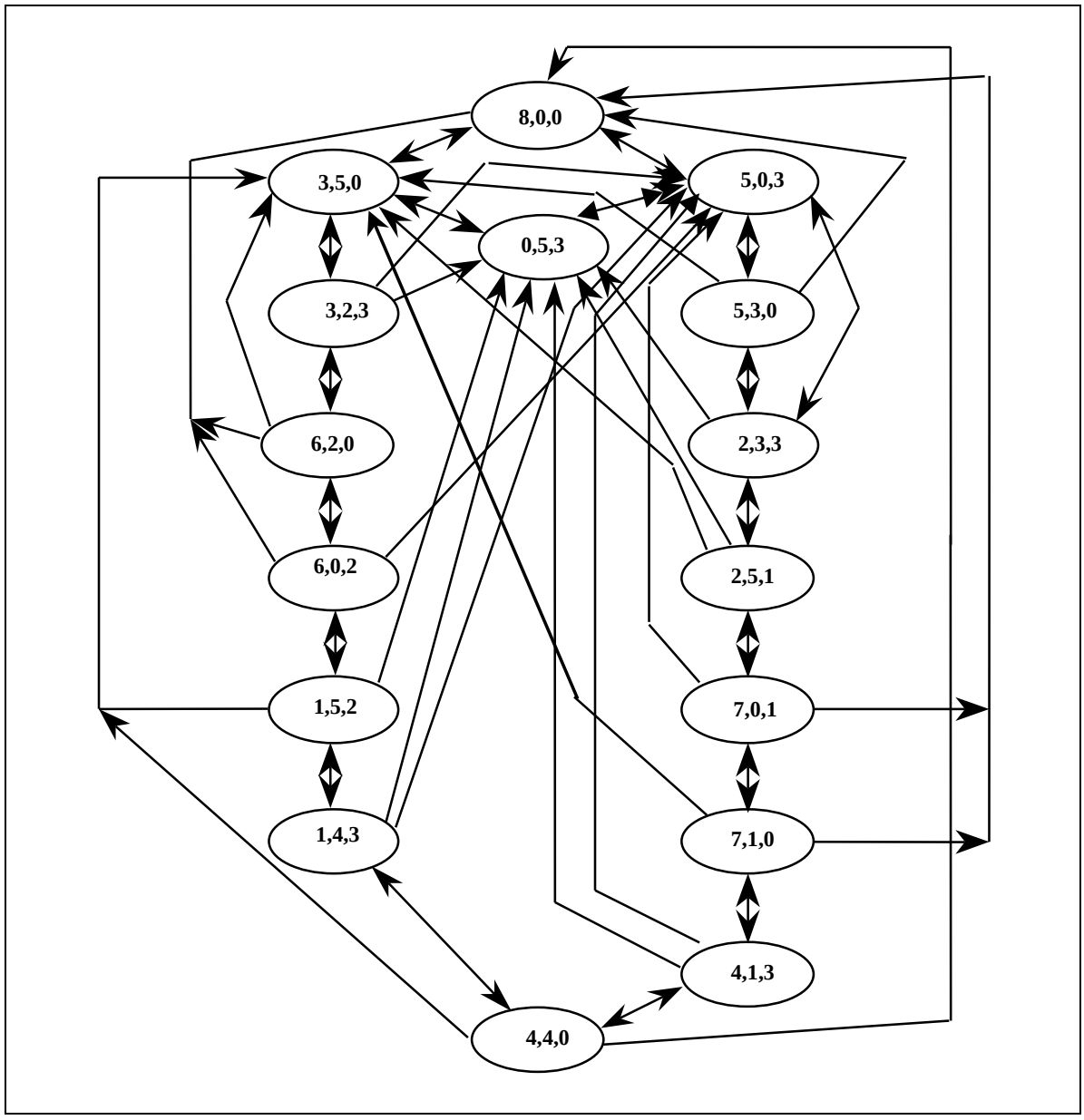


Figure 1-2 Graph of the Three Jars

Example 3 :

The Königsberg Bridge (Kaliningrad) spans the Pregel River. In the middle of this river, there is the island of Kneiphoff (see Figure 1.3).

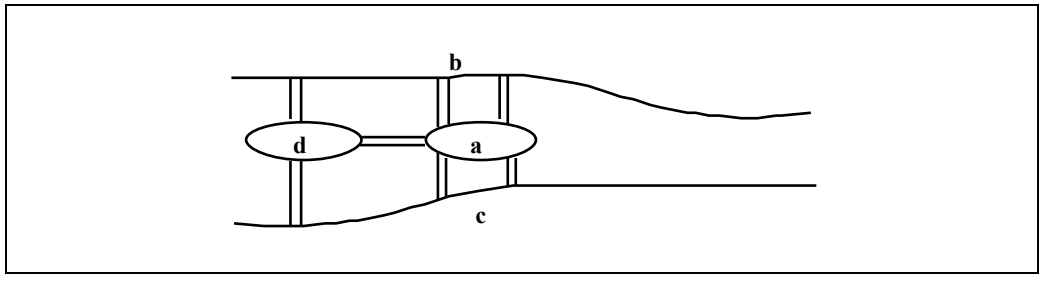


Figure 1-3 The Königsberg Bridge

Can a pedestrian cross each bridge exactly once? In 1736, Euler proved that this is impossible. To demonstrate this, he associated the problem with the multigraph shown in Figure 1.4.

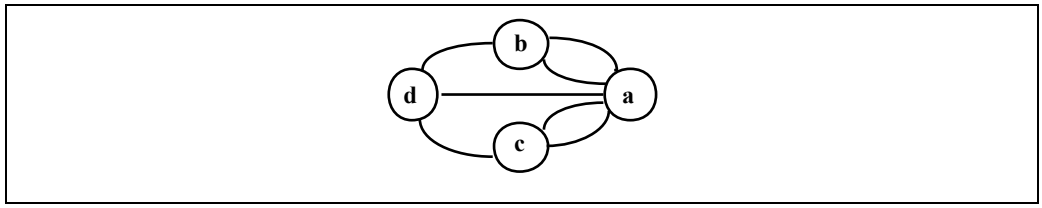


Figure 1-4 Euler's Graph

To each landmass corresponds a vertex, and to each bridge corresponds an edge. The problem can then be formulated as follows: is there a path that passes exactly once through each edge of the multigraph (referred to as an Eulerian path)? The answer is given by the following theorem, which we will not prove:

Euler's Theorem: A simple multigraph (without loops) G has an Eulerian path if and only if it is connected (i.e., "in one piece") and the number of vertices with odd degree is either 0 or 2.

In the case of the Königsberg bridges problem, the 4 vertices in the graph of Figure 1.4 have odd degrees, so there is no solution.

1.2. Vocabulary of Graph Theory:

The purpose of this paragraph is to provide the basic definitions of graph theory. These definitions are numerous but very intuitive, which makes them easy to learn!

1.2.1. Directed and undirected graph, weighted graph:

directed graph: A **directed graph** is a pair G = (X, U), where X is a set whose elements are called **vertices**, and U is a subset of X x X whose elements are called arcs.

undirected graph: An **undirected graph** is a pair G=(X, E), where X is a set whose elements are called **vertices**, and E is a subset of X choose 2 (i.e., all possible pairs of vertices) whose elements are called **edges**.

Note: We will denote n as the number of vertices in a graph and m as the number of arcs (or edges).

An undirected graph can be associated with a directed graph by "dropping the orientation." Similarly, an undirected graph can be associated with a directed graph by either introducing two arcs corresponding to each edge or choosing only one of the two arcs (see Figure 1.5).

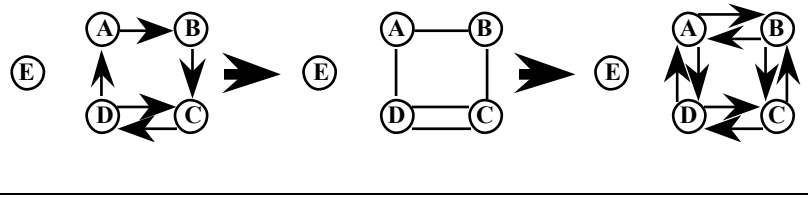


Figure 1-5 Directed graphs and undirected graphs

Weighted graph:

A weighted graph is a triplet G = (X, U, v), where (X, U) is a graph and v is a mapping from U to $\mathbb{R}$ (set of real numbers).

1.2.1 Initial endpoint, terminal endpoint, successor, predecessor:

Initial and terminal (successor and predecessor) ends:

Consider an arc (i, j); i is called the **initial end** of the arc (i, j), and j is the **terminal end**. It is also said that j is a successor of i, and i is a **predecessor** of j. The arc (i, j) is said to be **incident** outward at i and inward at j.

We denote U+(i) as the set of successors of i, U-(i) as the set of predecessors of i, d+(i) as the **half out-degree** of i, which is the cardinality of U+(i), and d-(i) as the **half in-degree** of i, which is the cardinality of U-(i) (refer to Figure 1.6).

1.2.2. Degree :

Vertex degree:

The degree of a vertex is the number of arcs incident to that vertex.

When there are no loops on a vertex, meaning there are no arcs where the initial and terminal ends coincide, its degree is the sum of its half in-degree and half out-degree (refer to Figure 1.6).

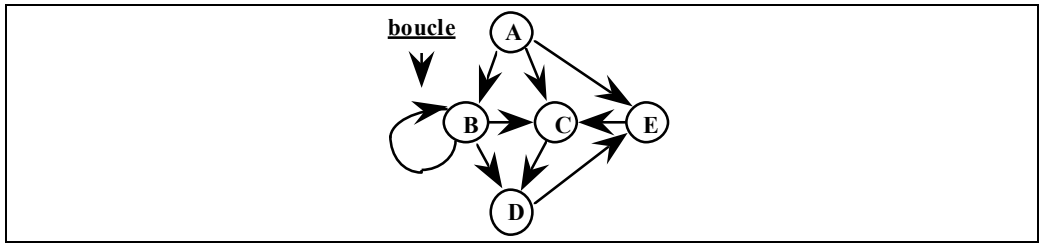


Figure 1-6 Directed graph

In Figure 1.6, we have:

$U^-(E) = \{A, D\}$ $d^-(E) = 2$ $d^-(A) = 0$ $d(C) = 4$

$U^+(E) = \{C\}$ $d^+(E) = 1$ $d^+(A) = 3$ $d(B) = 4$

1.2.3. Paths, circuits:

Path:

A **path** is a sequence of vertices [x0, x1, ..., xp] such that the arcs (x0, x1), (x1, x2), ..., (xp-1, xp) belong to the graph. The value of a path is the sum of the weights/valuations of the arcs along this path. The length of a path is the number of arcs it contains.

The vertex x0 is called the **initial vertex** of the path, and the vertex xp is called the **terminal vertex**.

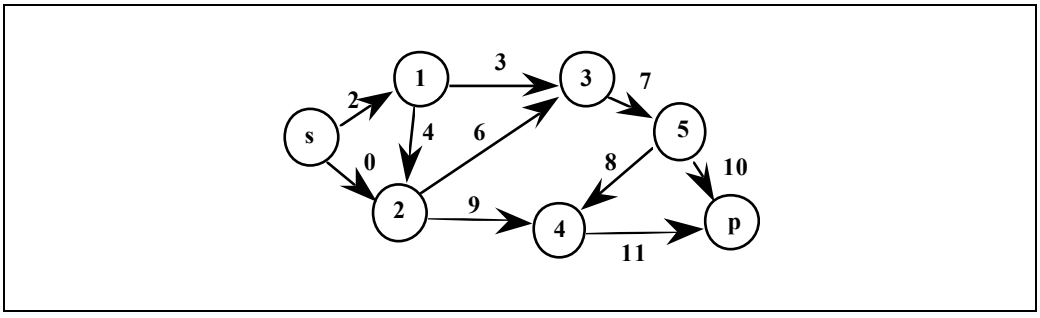


Figure 1-7 Weighted graph

For example, in the weighted graph shown in Figure 1.7, the value of the path {s, 1, 2, 4, p} is 26 and its length is 4.

Circuit :

A circuit is a path in which the initial vertex coincides with the terminal vertex.

Elementary and simple paths:

A path is said to be **elementary** (or **simple**) if it does not pass through the same vertex (or arc) more than once.

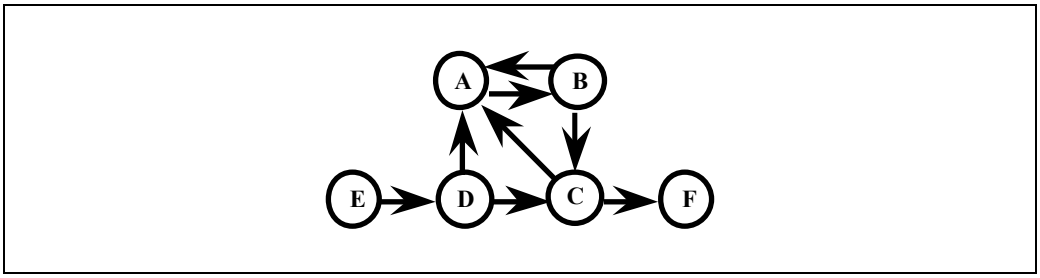


Figure 1-8 Directed graph

On figure 1.8: The paths [E, D, C, F] and [E, D, A, B, C, F] are elementary, whereas the path [E, D, A, B, A, B, C, F] is not elementary.

Eulerian and Hamiltonian paths:

A path is said to be **Eulerian** (resp. **Hamiltonian**) if it traverses each

Descendant, ascendant:

A vertex j is a **descendant** of a vertex i if there exists a path going from i to j, or if i = j; in this case, i is called an **ancestor** of j.

Source, sink :

A **source** is a vertex that is an ascending point to all other vertices, while a **sink** is a vertex that is a descending point from all other vertices.

Note A synonym for source (respectively, sink) is root (respectively, anti-root).

1.2.4. Chains, cycles:

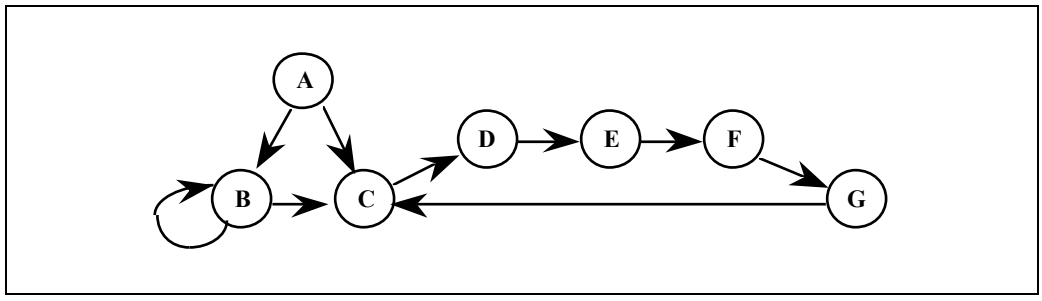


Figure 1-9 Directed graph

Chain :

A **chain** is a sequence of arcs u1, u2, ..., up such that one endpoint of the arc ui ($2 \leq i \leq p-1$) is common with the arc ui+1, while the other endpoint is common with the arc ui-1. The length of a chain is the number of arcs it contains.

In Figure 1.9, (A B) (B C) (G C) (F G) (E F) (D E) is a chain.

A chain can be "seen" as a sequence of vertices such that two consecutive vertices are connected by an arc, with some arcs being traversed in the direction of the chain (positive direction), and others in the opposite direction (negative direction).

Cycle :

A cycle is a chain in which the initial endpoint coincides with the terminal endpoint.

Similarly, we define a simple chain, Hamiltonian chain, elementary chain, Eulerian chain, Hamiltonian cycle, and Eulerian cycle.

1.2.5. Partial graph, subgraph:

Subgraph :

Let G = (X, U) be a graph.

The subgraph associated with the subset A of X is defined by the graph G_A as follows:

$G_A = (A, U \cap A \times A)$.

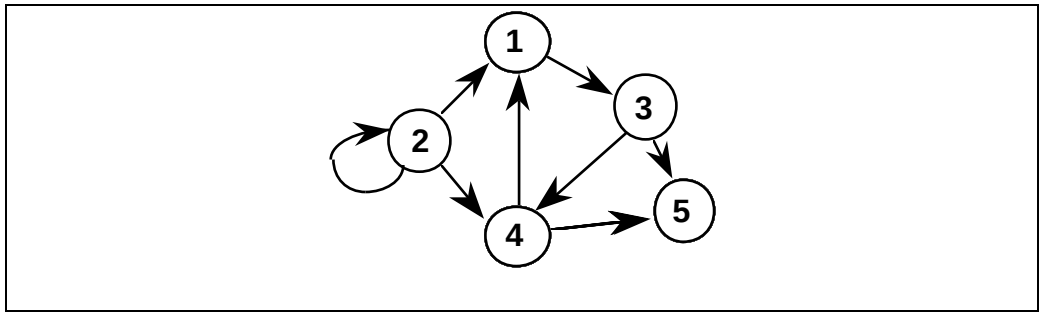


Figure 1-10 Directed Graph

Let's consider the example from Figure 1.10 where n = 5 and m = 8, the subgraph associated with the set A = {1, 4, 5} is shown in Figure 1.11.

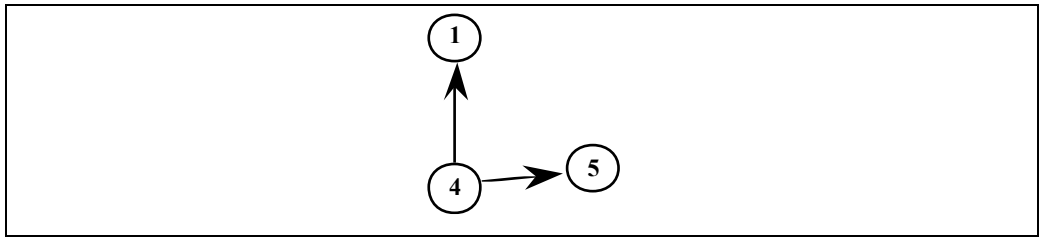


Figure 1-11 Subgraph

Partial graph:

A **partial graph** G' of G is a graph that has the same set of vertices as G, and whose set of edges is a subset of the set of edges of G: G' = (X, U') where U' is a subset of U.

On Figure 1.12, a partial graph of the graph shown in Figure 1.10 is depicted.

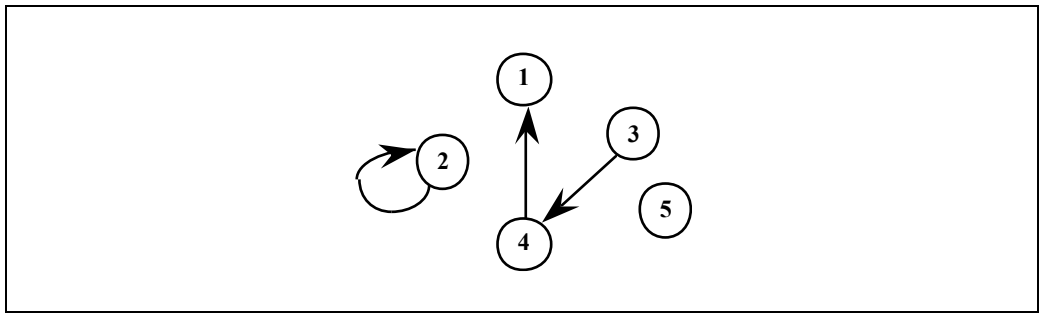


Figure 1-12 Partial graph

A **partial subgraph** is a subgraph of a partial graph.

1.2.6. Connectivity:

Simple Connectivity (Undirected):

The relation i C j (read as i connected to j) holds if i = j or if there exists a chain from i to j. This is an equivalence relation whose classes are called connected components.

Strong Connectivity (Directed):

The relation i FC j holds if i = j or if there exists a circuit passing through i and j. This is an equivalence relation called strong connectivity. The classes are referred to as strongly connected components.

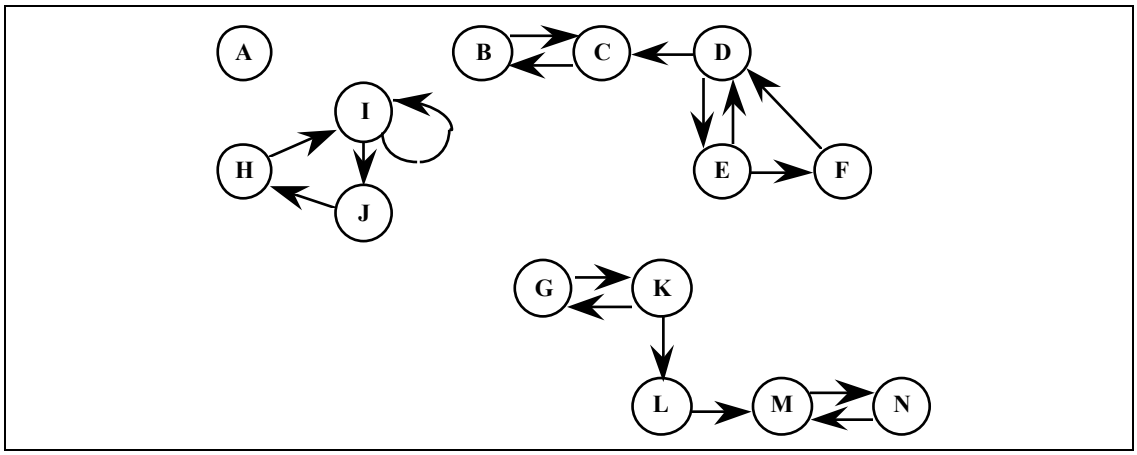


Figure 1-13 Connected components

In the example of Figure 1.13, we have 4 connected components: {A}, {B, C, D, E, F}, {H, I, J}, {G, K, L, M, N}.

In the previous example, we have 7 strongly connected components: {A}, {B, C}, {D, E, F}, {H, I, J}, {G, K}, {L}, {M, N}.

Reduced graph:

The **reduced graph** GR is the graph (X/FC,V) whose set of vertices is the set of strongly connected components, and where every arc connects two distinct components, each containing at least one pair of vertices that are connected in G. The reduced graph is acyclic.

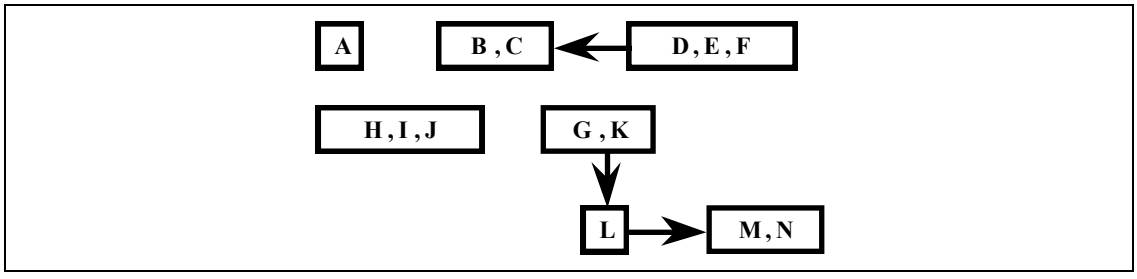


Figure 1-14 Reduced graph

On Figure 1.14, we have drawn the reduced graph of Figure 1.13.

1.2.7. Cocycle, cocircuit :

Cocycle :

A **cocycle** is a set of arcs that connect two complementary parts of a graph. If Y is a subset of X, the associated cocycle is denoted by $\omega(Y)$, and we have $\omega(Y) = \omega(X - Y)$, where X - Y represents the complement of Y in X.

Cocircuit :

A **cocircuit** is a cocycle in which all the arcs have the same direction.

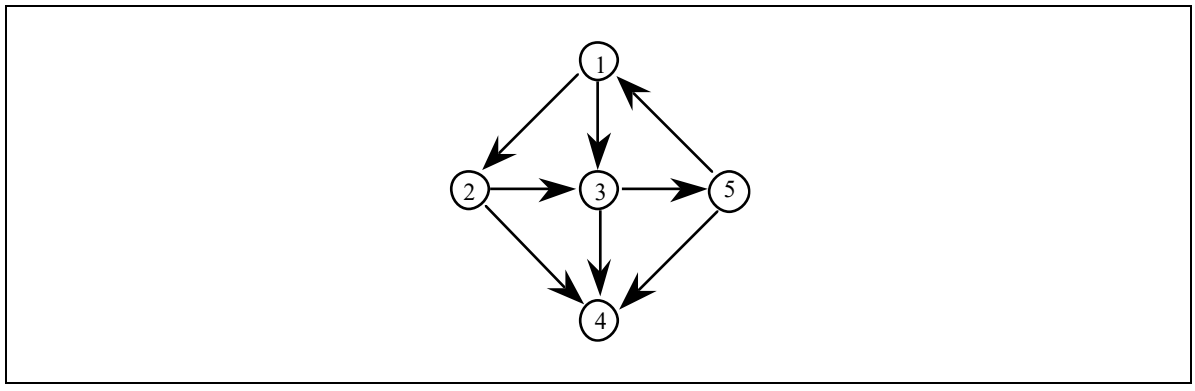


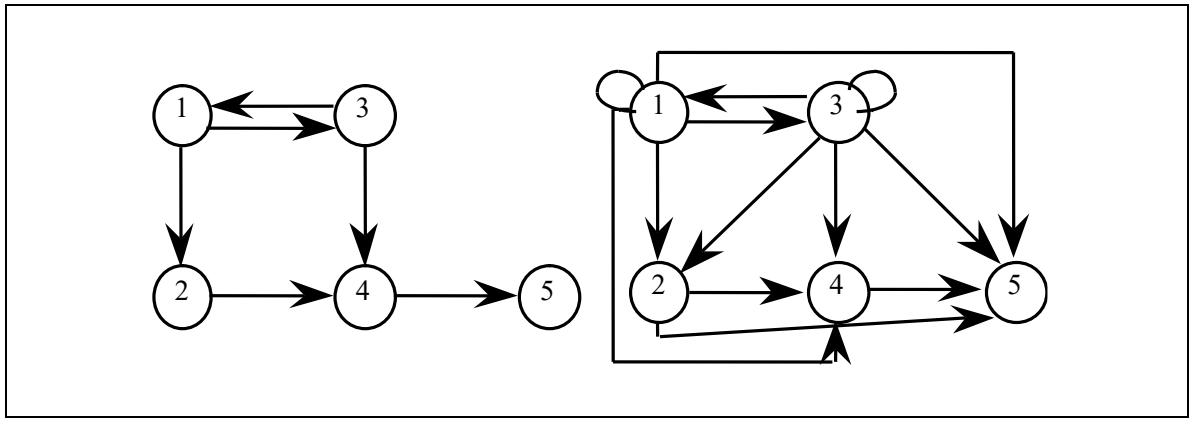
Figure 1-15

In figure 1.15, ω(4) = {(2,4),(3,4),(5,4)} is a cocircuit, whereas ω{(1,2)} = {(2,4), (2,3), (1,3),(5,1)} is not a cocircuit.

1.2.8 Transitive closure:

Transitive closure :

The **transitive closure** of a graph G = (X,U) is the graph G" = (X,U") with the same set of vertices as G, such that (i,j) ∈ U" if there exists a path from i to j in G.



G = (X , U) G" = (X, U")

Figure 1-16 : The transitive closure of a graph

1.2.9. Chromatic number and stability number:

Stability :

A subset S of X is stable if there is no edge connecting two vertices in S. The stability number $\alpha(G)$ is the maximum cardinality of a stable set.

Chromatic number:

A graph G is c-chromatic if it is possible to color its vertices with c colors such that no two adjacent vertices have the same color. The chromatic number $\gamma(G)$ is the smallest c for which G is c-chromatic.

Example:

We consider a graph whose set of vertices is the set of exams for the second semester at U.T.C: two vertices are connected by an arc if the same student must pass the two corresponding exams. We want to determine a schedule in a minimum number of periods (one period being equal to half a day). Each color will correspond to a period of time during which all the exams of this color will take place.

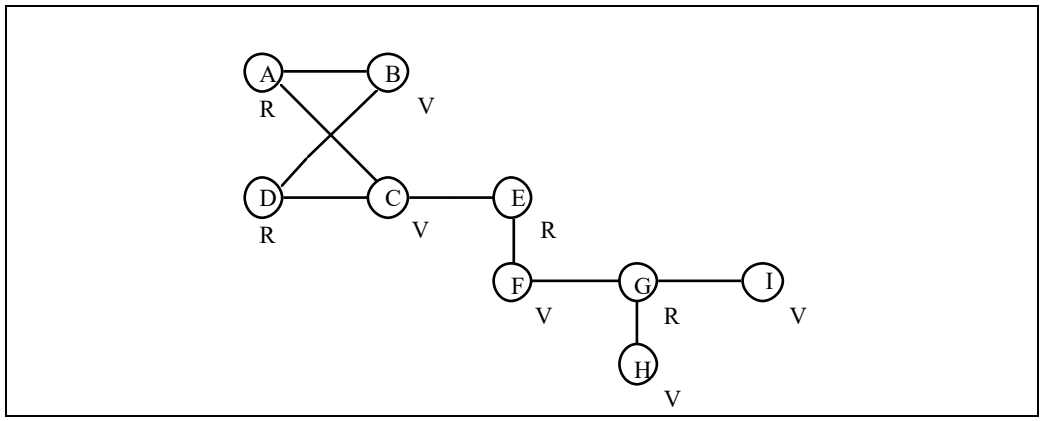


Figure 1-17 Coloring

In Figure 1.17, vertices A, D, E, and G are colored red, while vertices B, C, F, H, and I are colored green. The chromatic number $\gamma(G) = 2.1.3$.

Particular graphs:

1.3.1. Forest :

A forest is a graph without cycles.

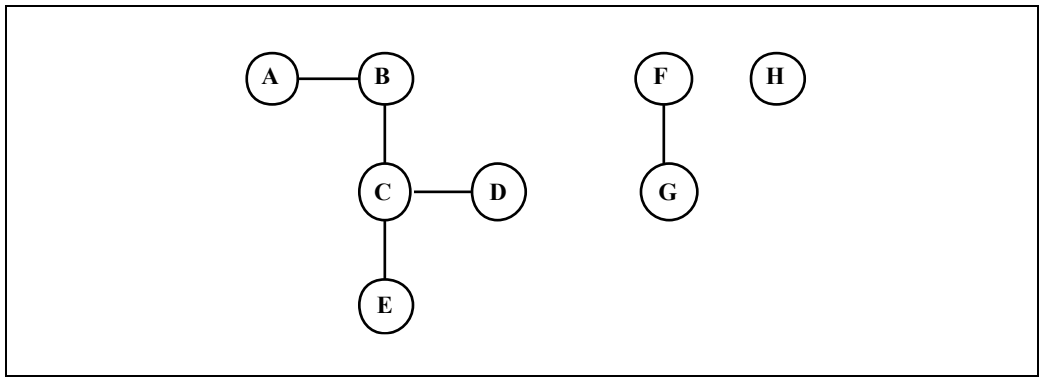


Figure 1-18 Forest

1.3.2. Tree :

A tree is a connected graph without cycles (with $|X| \geq 2$).

On Figure 1.19, a tree with 9 vertices and 8 edges is drawn.

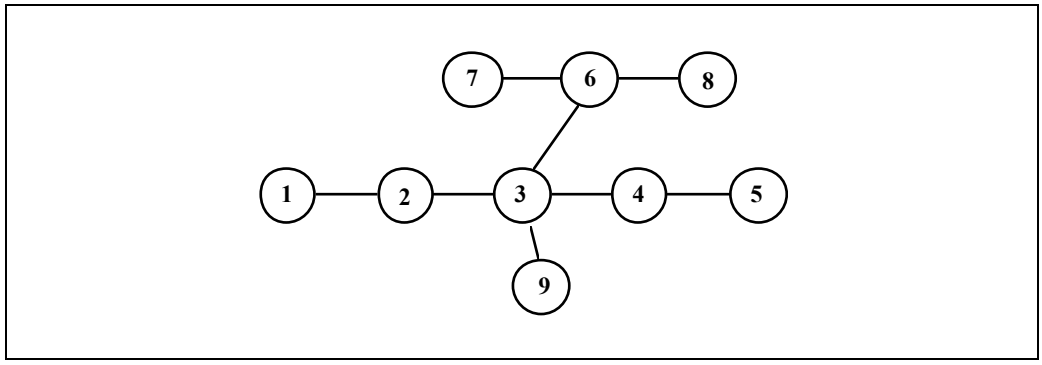


Figure 1-19 Tree

Characteristic properties (see tutorial):

A tree is connected and has n-1 edges.

A tree is acyclic and has n-1 edges.

A tree is connected, and if one edge is removed, it is no longer connected.

A tree is acyclic, and if one edge is added, it creates a cycle.

There exists a unique simple path between any pair of vertices in a tree.

1.3.3. Arborescence :

Arborescence :

An **arborescence** is a directed tree in which every vertex (except one) has exactly one predecessor. The vertex without any predecessor is called the **root** of the arborescence.

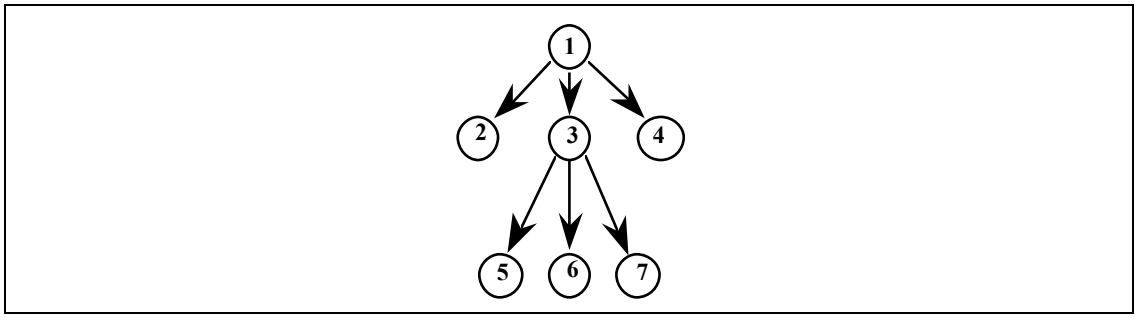


Figure 1-20 Arborescence

For every vertex, there exists a unique path connecting it to the root. Vertices without any successors are called **pendant vertices** or **terminal vertices.**

1.3.4. Biparte graph :

A **bipartite graph** G = (X, U) is a graph in which the set X of vertices can be partitioned into two disjoint sets Y and Y' such that every edge has one endpoint in Y and the other in Y'.

Example: Battle of Britain.

In 1941, the British squadrons were composed of biplane aircraft, but some pilots could not team up with certain mechanics due to language or habit reasons.

To address this issue, a graph can be associated where the set of vertices includes the pilots and the mechanics. A pilot is "connected" to a mechanic if they can both embark on the same aircraft.

In Figure 1.21, the bold lines represent a matching between the set of pilots and the set of mechanics.

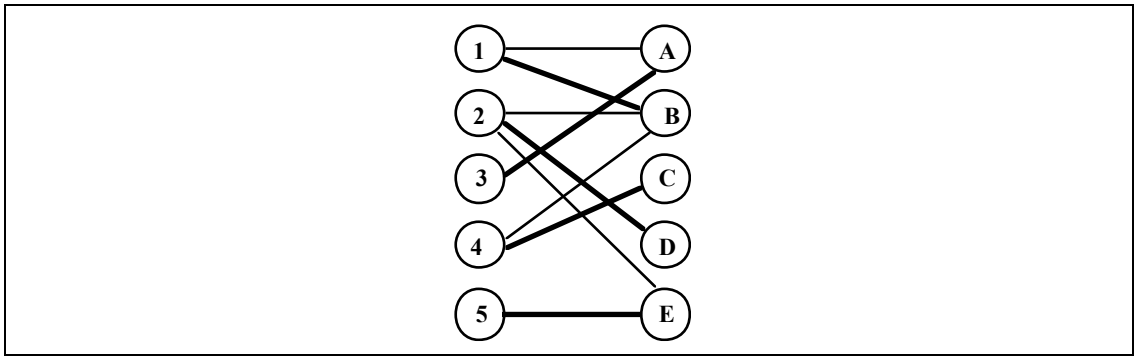


Figure 1-21 Matching

Matching:

A **matching** is a set of edges such that no two edges in the set share a common vertex.

Example: { (1, B), (2, D) } is a matching in the graph shown in figure 1.21, while { (1, B), (2, D), (3, A), (4, C), (5, E) } is a maximal matching.

1.3.5. Planar graphs:

A graph is said to be **planar** if it can be represented in the plane in such a way that the vertices are distinct points and the edges do not intersect.

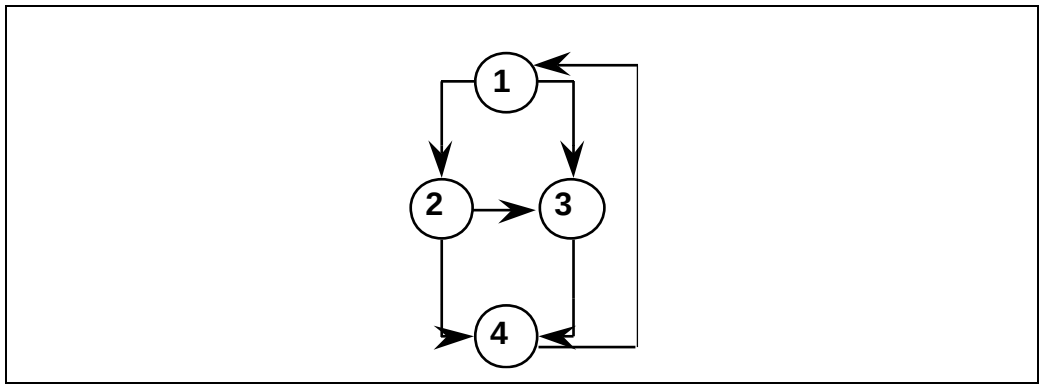


Figure 1-22 : example of planar graph

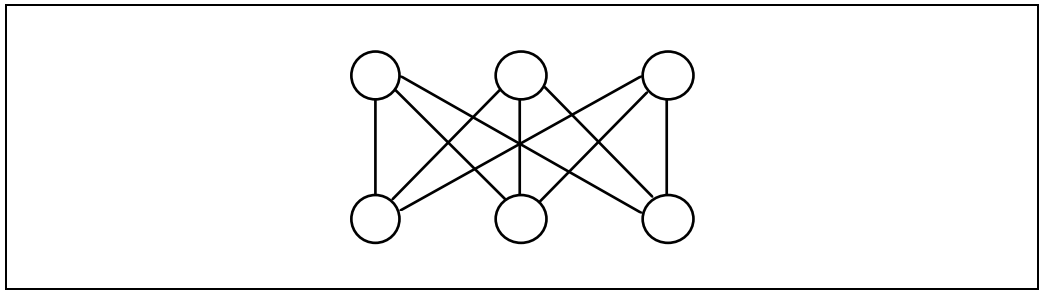


Figure 1-23 :graph of the 3 factories (non-planar)

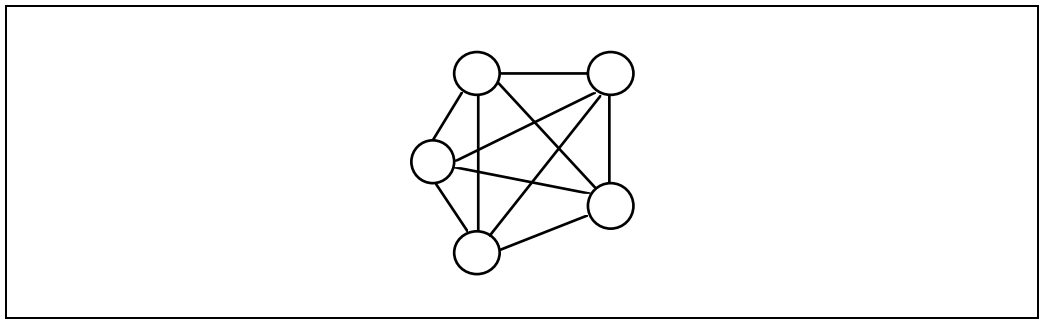


Figure 1-24 : complete graph with 5 vertices(non-planar)

On figures 1.23 and 1.24, the two minimal non-planar graphs are drawn. Indeed, if we remove an edge from either of these two graphs, the resulting graph becomes planar. Furthermore, any non-planar graph contains a partial subgraph that can be matched with one of these two graphs by associating a chain from the partial graph with an edge from the graph (Kuratowski's theorem).

Four-color conjecture (1875 Peterson):

"Every planar graph is 4-colorable" (the vertices can be colored with 4 colors). This conjecture was "proved" with the help of a computer (1982).

Originally, this problem originated from the coloring of a geographical map: Can a geographical map be colored with 4 colors?

1.3.5. Cliques:

A clique is a complete graph without loops (in directed and undirected cases). In the directed case, a clique has n(n-1) arcs, and in the undirected case, it has n(n-1)/2 edges. Only cliques of cardinality 1, 2, 3, and 4 are planar.

2. BASIC POLYNOMIAL ALGORITHMS FOR GRAPHS:

2.1. Introduction :

We will present in this chapter two polynomial graph algorithms: one algorithm for properly numbering the vertices of a graph, and one algorithm for finding a maximum cardinality matching in a bipartite graph. Before that, we will describe the different data structures for representing a graph in a computer and define the notion of a polynomial algorithm. This notion is important because it is basic for the efficiency of programs. We will see that the complexity of an algorithm strongly depends on the data structure used and that some problems are inherently exponential. Other graph algorithms will be studied in the rest of the course and in directed exercises.

2.2. Definition of polynomiality:

> We say that a function f(x) is $\mathbf{O(x^p)}$, if there is a polynom P(x) of graph p such as $\forall x \in \mathbf{N}$, $f(x) \leq P(x)$.

We distinguish between memory and computational time complexities for algorithms. The former gives an upper bound on the number of memory words required to represent the parameters and variables in a data structure. The latter gives an upper bound on the number of elementary operations required to execute an algorithm. In both cases, x represents the size of the problem data to be processed, or more precisely, the number of digits in the data.

Note: For graphs, the complexity is expressed in terms of the number n of vertices and the number m of edges.

2.3. Coding a graph :

There are 4 main data structures for coding a graph: the adjacency matrix, the incidence matrix, the adjacency queue, and the inverse adjacency queue.

2.3.1. Adjacency Matrix :

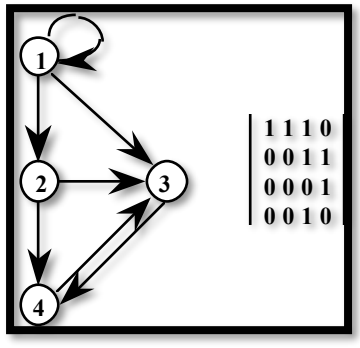


Figure 2.1 : Example of a graph with its adjacency matrix

The adjacency matrix of vertices-vertices is the square matrix of order n denoted by $A = (a_{ij})$ such as : $a_{ij} = 1$ si $(i, j) \in U$,

$a_{ij} = 0$ sinon.

Note that the encoding using the associated matrix is in $O(n^2)$ (complexity of encoding) because this encoding uses n^2 memory words. This is therefore costly, but it is the simplest way to encode a graph.

2.3.2. Vertex-Edge Incidence Matrix:

The vertex-edge incidence matrix N is defined by N_{ij}:

N_{ij} = 1 If vertex i is the initial endpoint of u_j.

N_{ij} = -1 if i is the terminal endpoint of uj.

N_{ij} = 0 other.

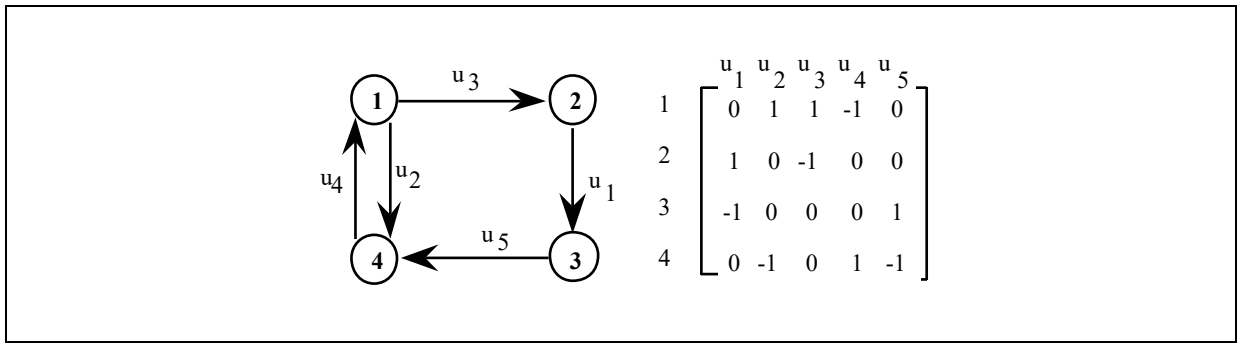


Figure 2.2 : Graph and its incidence matrix

Note: The coding complexity in terms of n, the number of vertices, and m, the number of arcs is O(n x m). This matrix is useful when we need to model graph problems using linear programming (such as flow).

2.3.3. Successor queue :

The successor queue is composed of two arrays, ALPHA and BETA. BETA is the queue of successors of the vertices arranged in the order of their numbering, and ALPHA is the array that gives to a given node number the location in the BETA array of its first successor. This assumes that the vertices are numbered in the interval [1..n] (where n is the number of nodes in the graph).

Remarks: Information about the successors of vertex I can be found in the BETA array between the addresses ALPHA(I) and ALPHA(I+1)-1. The complexity of the coding is in O(n + m).

This structure is widely used because it is very memory-efficient for sparse graphs and efficient in computation time when direct access to the successors of vertices is needed.

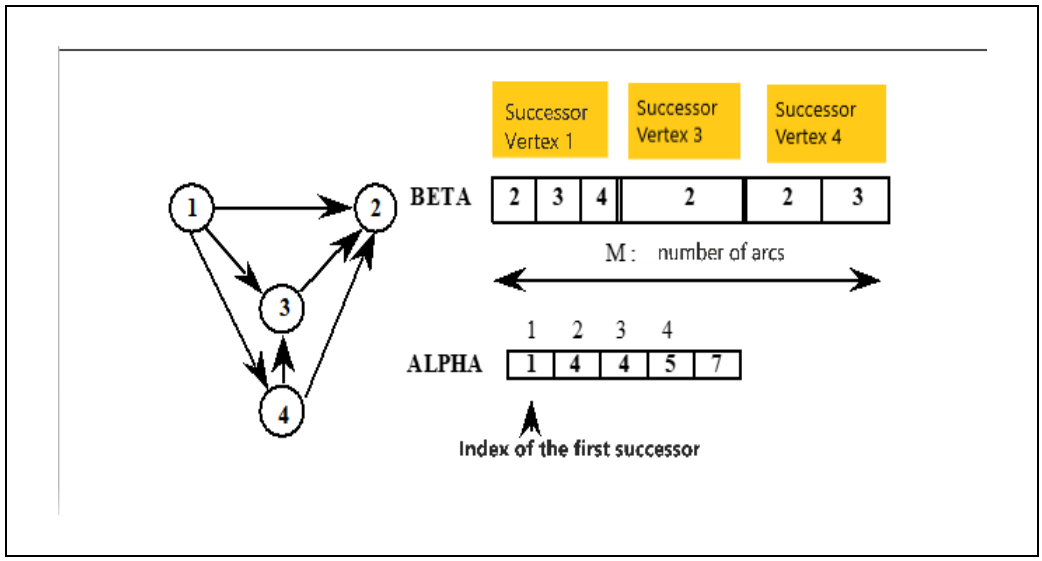


Figure 2.3 : Example of a graph and its successor queue.

2.3.4. Predecessor queue:

This queue is defined in a similar way. It allows direct access to predecessors.

2.4. Algorithm for a good numbering of a graph:

Good numbering :

> We say that numbering (x) (mapping from X -> [1..n]) is a good numbering if, for all arcs (x,y) belonging to U, numbering (x) is strictly smaller than numero(y).

We will show below that for there to exist a good numbering, it is necessary and sufficient that the graph is acyclic. We also state a result that allows us to construct a linear complexity algorithm to determine a good numbering of an acyclic graph.

Proposition 1

A necessary and sufficient condition for a graph to be acyclic is that any non-empty subset of vertices A has at least one element whose every predecessor is in the complement of A. That is, the subgraph G_A has at least one vertex without a predecessor.

Proof :

Suppose G has a circuit $[x_0, x_1, ..., x_r]$. Let $A = \{x_0, x_1, ..., x_r\}$ and consider the subgraph G_A: every vertex of G_A belonging to the circuit has at least one predecessor in A. Therefore, there exists a set A for which the property is false. Conversely, suppose G is without a circuit and the property is false. There is thus a subgraph G_A in which all vertices have at least one predecessor in A. Starting from xi0 in G_A, x_{i0} has a predecessor x_{i1}... x_{ih-1} has a predecessor x_{ih}. We can thus construct

a path [x_{in}, x_{in-1}, ...xi0]. Since the graph has n distinct vertices, two vertices in this path are identical. This graph thus has a circuit, which is absurd. (Q.E.D.).

Proposition 2 :

A necessary and sufficient condition for a graph to be without circuit is that there exists a good numbering.

Proof :

If G has a circuit [x_{i0}, x_{i1},...., x_{ip}, x_{i0}], there cannot exist a good numbering because otherwise we would have: number(x_{i0}) < number(x_{i1}) < ... < number(x_{ip}) < number(x_{i0}). We prove the converse by induction on the number of vertices in the graph. It is clear that if G has only one vertex and no circuit (thus no loop), there exists a good numbering. Now suppose that the property, i.e. every graph without a circuit with at most K vertices has a good numbering, is true for K = n-1, and prove it for K = n. If G has n vertices and has no circuit, then there exists a vertex without a predecessor, according to Proposition 1 by taking A = X. We number this vertex as 1 and reason on the subgraph obtained by removing it. This subgraph has n-1 vertices. Therefore, according to the induction hypothesis, it has a good numbering. We use this good numbering to number 2, ..., n for these vertices, which results in a good overall numbering. (Q.E.D.).

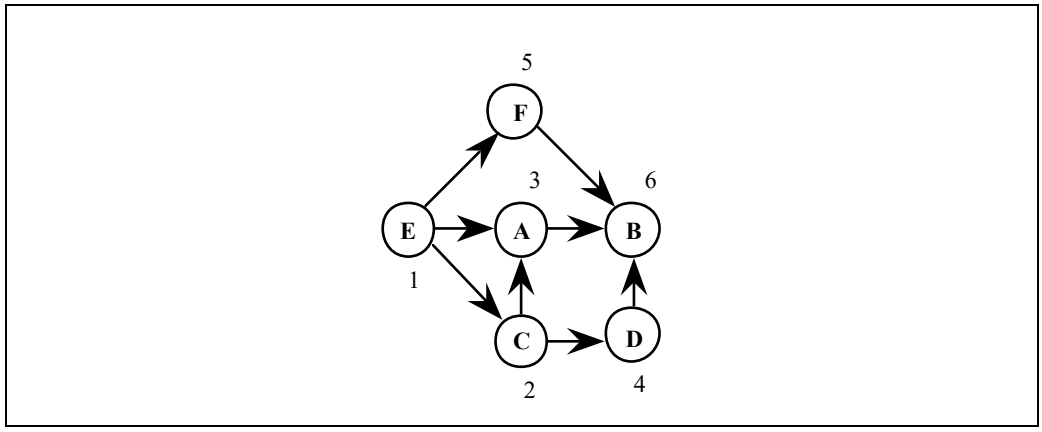


Figure 2.4 : Example of good numeration.

The NUMEROTER algorithm below allows for properly numbering the vertices of a graph without circuits. The graph initially has an arbitrary numbering and is encoded using the successor queue.

NUMBERING Algorithm :

(i) **Initialization**

Read N, M, ALPHA, BETA. {ALPHA is an array of size N+1 and BETA, of size M}

(ii) **Calculation of the number of predecessors for each vertex**

```
For I = 1 to N do NOMBRE(I) = 0
For K = 1 to M do
Begin
I = BETA(K)
NOMBRE(I) = NOMBRE(I) + 1
End
```

(iii) **Initialization of the stack of vertices without predecessors**

```
vertex = 0
For I = 1 to N do
Begin
If NOMBRE(I) = 0 then
Begin
vertex = vertex + 1
stack(vertex) = I
End
End
```

(iv) **Numbering of the vertices**

```
For J = 1 to N do
Begin
I = stack(vertex)
vertex = vertex - 1
NUMERO(I) = J
For H = ALPHA(I) to ALPHA(I+1) - 1 do
Begin
L = BETA(H)
NOMBRE(L) = NOMBRE(L) - 1
If NOMBRE(L) = 0 then
Begin
vertex = vertex + 1
stack(vertex) = L
End
End
```

End

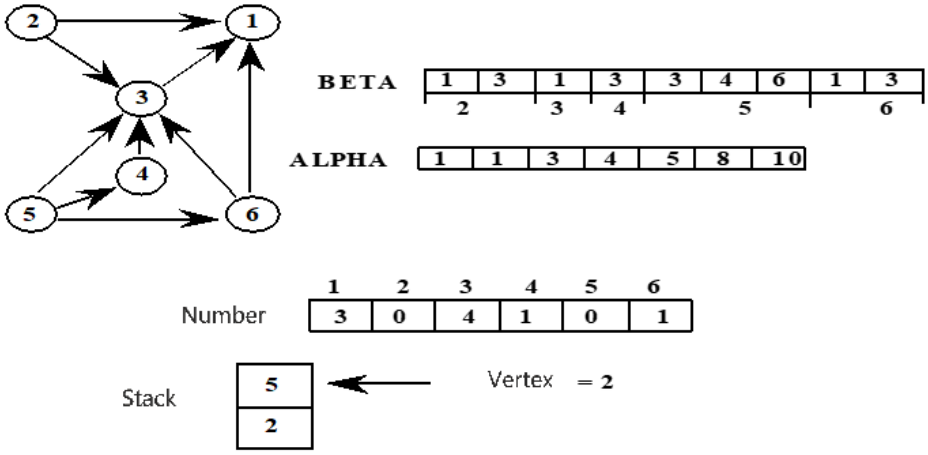


Figure 2.5 : Starting graph with initialization of number and stack.

Numeric Application

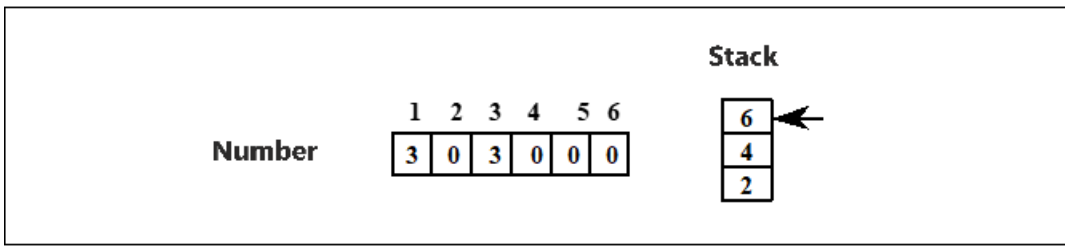


Figure 2.6 Evolution of the stack and array after the numbering of 5.

During the numbering of 5, we get:

I = 5

NUMERO(5) = 1

For H = 5 to 7

L = 3

Number(3) = Number(3) - 1

L = 4

Number (4) = 0 (Vertex 4 --> stack)

L = 6

Number (6) = 0 (Vertex 6 --> stack)

etc.

Complexity of the Algorithm:

(i) Initialization:

Reading N and M is in O(1). Reading BETA and ALPHA is in O(N + M).

(ii) Calculation of Number:

The first loop is in O(N) while the second loop is in O(M).

(iii) Stack:

The first instruction is in O(1) while the loop is in O(N).

(iv) Numbering:

The loop is in O(N) for the first three instructions. The other instructions are in O(M) because the loop corresponding to H amounts to traversing the BETA array once.

Total Complexity:

The total complexity is O(M) because:

O(N) + O(M) = O(N+ M) and O(N) + O(1) = O(N).

In general, it is assumed that N-1 ≤ M, hence O(N) + O(M) = O(Max(N, M)) = O(M).

2.5. Finding a maximal matching:

In this paragraph, we will study the definitions and properties of matchings, in particular the notion of augmenting path. We will see that a matching is maximal if and only if the graph does not contain an augmenting path. Algorithms for finding a maximum matching are based on this result. We present below such an algorithm for the particular case of bipartite graphs with a complexity of $O(n^3)$. Note that there exist polynomial algorithms in the general case, but these algorithms are too technical to be presented in this course.

2.5.1. Matching :

Definition :

> A matching C is a set of edges (or arcs) that does not contain two edges (or arcs) adjacent to the same vertex. The edges (or arcs) of the matching are called thick, while the others are called thin.

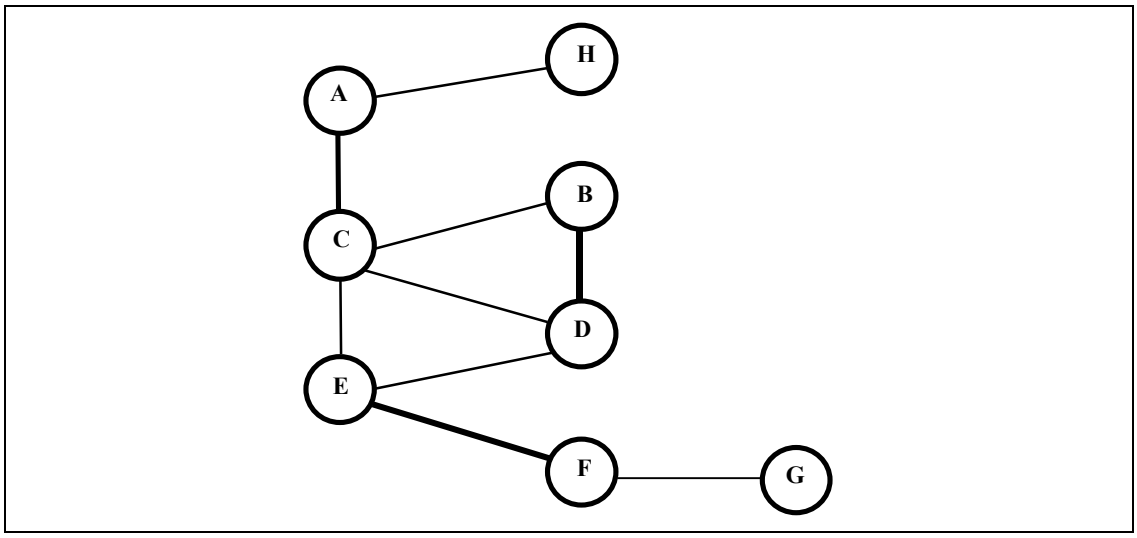


Figure 2.7 : A matching of size 3.

Transversal :

A transversal T of a graph is a subset of vertices that intersects every edge of the graph at least once.

Saturated and unsaturated vertex:

For a matching C, a vertex is said to be saturated if an edge of the matching is adjacent to this vertex, otherwise it is said to be unsaturated.

Chain alternated, improving:

An alternated chain of a matching C is a chain in the graph whose edges are alternately in the matching and not in the matching. An improving chain is an alternated chain connecting two unsaturated vertices.

Note: An improving chain has an odd length and contains h thick edges and h+1 thin edges. To improve the matching, we remove the thick edges of this chain from C and replace them with its thin edges.

In Figure 2.7, C = {AC, BD, EF} is a matching of cardinality 3. The vertices A, B, C, D, E, F are saturated and the vertices H and G are unsaturated.

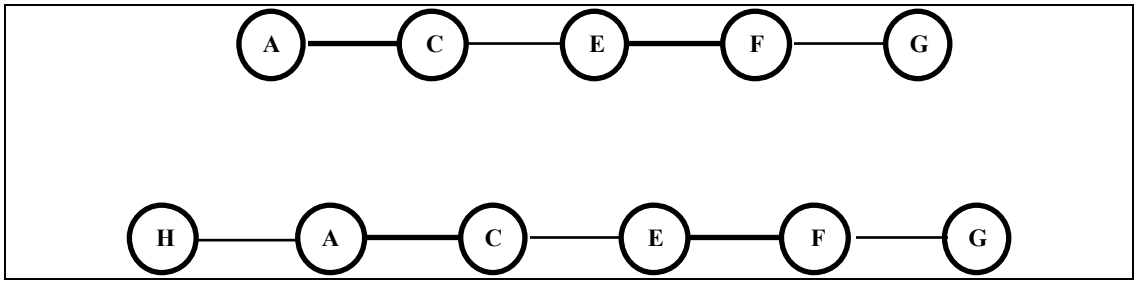


Figure 2.8 : Alternating path and augmenting path.

On figure 2.8, [A, C, E, F, G] and [H, A, C, E, F, G] are alternating paths. [H, A, C, E, F, G] is an augmenting path because it connects 2 unsaturated vertices. This path allows to move from the cardinality-3 matching C = {AC, BD, EF} to the cardinality-4 matching C' = {HA, CE, FG, BD}. C' is a maximum cardinality matching because all vertices in the graph are saturated (see figure 2.9).

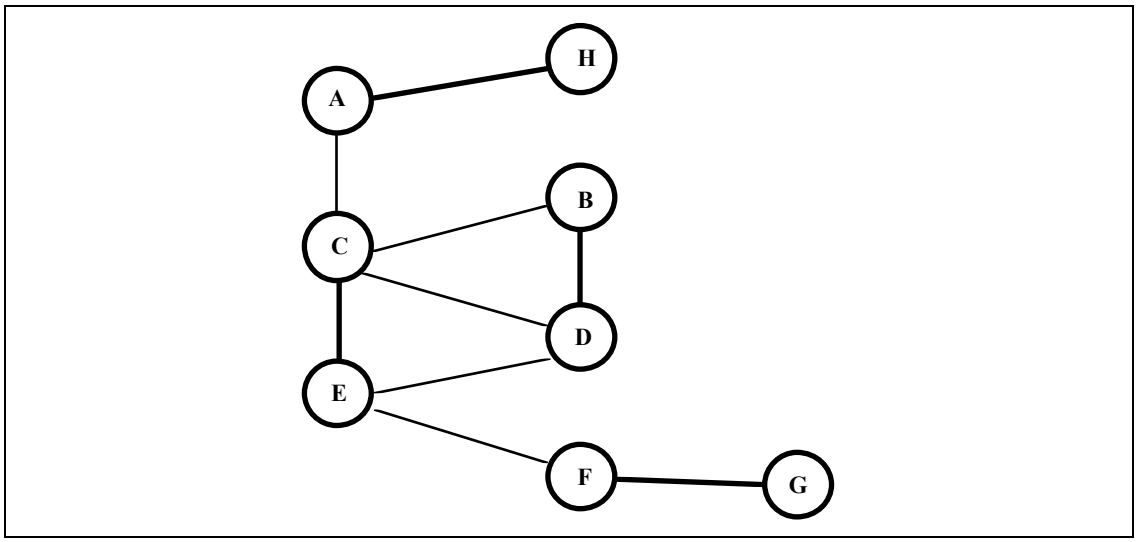


Figure 2.9 : Maximal matching.

2.5.2. BERGE's theorem:

Definition :

> A matching is of maximal cardinality if and only if the graph does not contain any augmenting path for that matching.

Proof :

Necessary condition: If there exists an augmenting path, it contains p thick arcs and p+1 thin arcs, so we can improve the matching by replacing the thick arcs in this path with the thin arcs.

Sufficient condition: Let G=(X,U) be a graph, C_0 be a maximum cardinality matching of G, and C_1 be a matching of G with no augmenting path. Let $D_0 = C_0 - C_0 \cap C_1$ and $D1 = C_1 - C_0 \cap C_1$. We reason about the graph $H=(X, D_0 \cup D_1)$.

The vertices of the graph H have degree 0, 1 or 2, since at most two incident arcs belong to it: one arc from D_0 and one arc from D_1. Consequently, the connected components of H are either isolated vertices or chains alternating between arcs from D0 (thus from C_0) and arcs from D_1 (thus from C1), that is, alternating paths for C0 and C1.

If one of these paths were of odd length, it would be an augmenting path for C_0 or C_1, which is impossible by hypothesis. These paths are therefore of even lengths and contain the same number of elements from D_0 as from D_1. Hence, the sets D_0 and D_1 have the same cardinality. Consequently, the matchings C_0 and C_1 have the same cardinality. It follows that C_1 is a maximum cardinality matching.

2.5.3. Algorithm in the case of a bipartite graph:

The BERGÉ theorem provides a method for finding a maximum matching. It suffices to search for augmenting paths successively and to use them to improve the matching. The matching will be maximum when there are no more augmenting paths. We present here an algorithm for bipartite graphs.

2.5.3.1. MAXIMUM MATCHING algorithm:

Begin.

Step 0 : Initialization

Read the data - a bipartite graph G=(X,Y,U) and an initial matching C (possibly C is empty);

Step 1 :Search for an augmenting path and improvment of the matching

Mark * every unsaturated vertex x in X;

While there exists an unexamined marked vertex i, do:

Begin

(a) If i = x ∈ X then

For each edge [x, y] □□C incident to vertex x do:

If y is not marked, then mark y with x;

(b) Otherwise, (i = y ∈ Y) do:

If y is unsaturated (it is an endpoint of an augmenting path):

Begin

Improve the matching C by reversing the augmenting path obtained;

Erase the marks;

Mark * every unsaturated vertex x in X;

End

Otherwise,

Begin

Determine the unique edge [x, y] ∈ C ;

Mark x with y;

End;

End (While.)

Step 2 : End.

Write the maximum matching C.

End

2.5.3.2. Application of the algorithm:

Consider the bipartite graph G = (X, Y, U) with X = {a, b, c, d, e} and Y = {1, 2, 3, 4, 5) shown in figure 2.10.

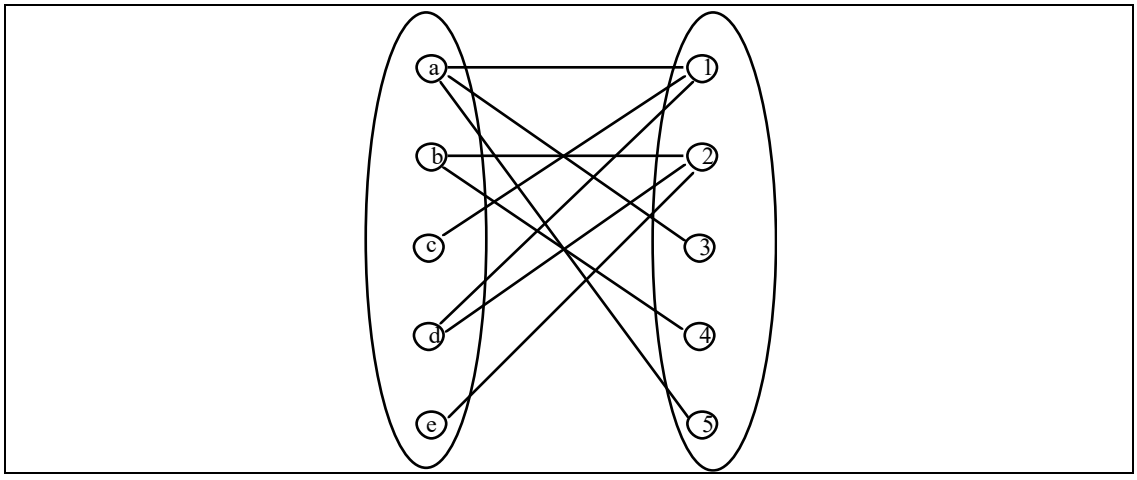


Figure 2.10 : Example.

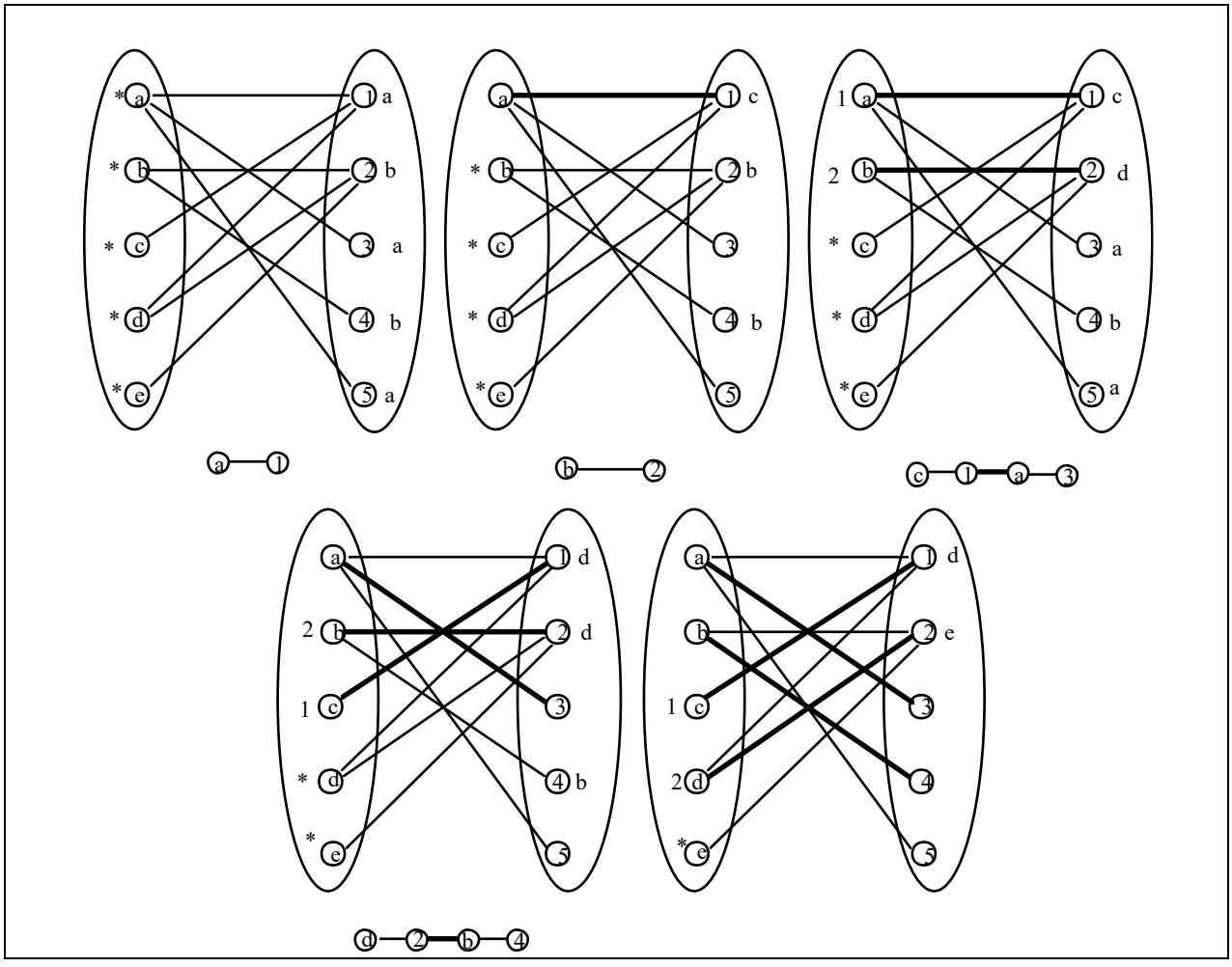


Figure 2.11 : Application of the algorithm.

We start with an empty matching and modify it using successive augmenting paths until we obtain a matching without augmenting paths, which is maximal according to the Berge's theorem.

The successive steps of the algorithm are shown in Figure 2.11, using the following rule: first marked vertex, first examined.

2.5.3.3. Data structures:

We assume that the bipartite graph and the matching are coded by an array TAB such that, for x ∈ X and y ∈ Y, TAB(x,y) = (0 if [x,y] ∉ U) or (1 if [x,y] ∉ C ∩ U) or (2 if [x,y] ∈ C ∩ U). If we assume without loss of generality that Cardinal(X) = Cardinal(Y) = N/2, then the TAB array is of dimension $N^2/4$. The TAB array allows reading data and will serve as a working table for the matching.

Let us list the other arrays that will be useful for this algorithm. A SATURE array (of size N) will be used to note the saturated vertices. A MARQUE array of size N will be used to store the marks of the vertices. It will be initialized to -1 (no mark), 0 will be put when a vertex is marked * by the algorithm, otherwise it will contain the mark of the vertex. A FILE array will be used to store the marked vertices not yet examined (rule: first marked, first examined). FILE allows direct access. A CHAINE array will be used to reconstruct the augmenting path.

2.5.3.4. Complexity of the algorithm:

The complexity of the initialization step is $O(N^2)$ (reading a matrix).

Step 1. We first consider the iterations that do not include any augmenting path. The initial marking costs O(N). One iteration of (a) in the while loop costs O(N) since we need to go through a row of the TAB matrix. One iteration of (b) when y is saturated costs O(N) since we need to read the x row of the TAB matrix. A step (b) when y is unsaturated costs O(N). Thus, obtaining an augmenting path would cost $O(N^2)$, and there will be at most N/2 augmenting paths, giving a complexity of $O(N^3)$ for step 1. The complexity of step 2 is $O(N^2)$ since we need to go through the TAB structure to retrieve all the edges of the matching. Therefore, the complexity of the algorithm is $O(N^3)$.

2.5.3.5. Algorithm Proof:

We will prove that this algorithm determines in step 1 an augmenting path if and only if one exists. Then we improve the matching. Thus, starting from the empty matching and after at most N/2 operations, we will obtain a matching without augmenting path which will be optimal according to Berge's theorem.

First, note that an augmenting path of odd length connects a vertex of X to a vertex of Y (bipartite graph). Therefore, we can start from X and search for alternating paths starting from an unsaturated vertex of X. An augmenting path will be such a path whose last vertex in Y is an unsaturated vertex.

Let's make the following induction hypothesis: if y ∈Y (resp. x ∈X) is one of the first p marked vertices, it is an endpoint of an alternating path starting from an unsaturated vertex $x_1 \in X$ of odd (resp. even) length whose last edge is thin (resp. thick).

The property is initially true: only unsaturated vertices of X are marked and the corresponding paths are of zero length. Let us show that if the property is true up to order p, it remains true at order p+1. Let j be the p+1th marked vertex and i its mark. There are two cases to consider:

• i = x $\in$X (j = y $\in$Y): then by the induction hypothesis, x is an endpoint of an alternating path starting from an unsaturated vertex x1 in X of even length whose last edge is thick. By concatenating this path with the edge (x,y), we obtain an alternating path starting from an unsaturated vertex x1 in X of odd length whose last edge is thin and going to j = y (the edge (x,y) is thin because y is marked in 1(a) of the algorithm).

• i $\in$y in Y (j = x $\in$X): then by the induction hypothesis, y is an endpoint of an alternating path starting from an unsaturated vertex x1 in X of odd length whose last edge is thin. By concatenating this path with the edge (x,y), we obtain an alternating path starting from an unsaturated vertex x_1 $\in$X of even length whose last edge is thick and going to j = x (the edge (x,y) is thick because x is marked in 1(b) of the algorithm).

The property is therefore true in both cases, which is proven by induction. Let M be the set of marked vertices at the end of step 1, and let us show that M is exactly the set of endpoints of an alternating path starting from an unsaturated vertex x1 $\in$X. According to the property proven by induction, all vertices in M have this property. We must show that conversely, a vertex having this property is in M.

Before proving this, note that when we reach step 2, the vertices of M belonging to Y are saturated. This reciprocal will show that there is no alternating path connecting an unsaturated vertex in X to an unsaturated vertex in Y, which will prove the result, namely that there is no augmenting path and that the matching is maximal.

Let k be a vertex connected by an alternating path to an unsaturated vertex x_1 $\in$X. We define j as the first vertex of this path not belonging to M (such a vertex exists because by hypothesis, k $\notin$ M). There are two possibilities: either j = x $\in$X or j = y $\in$Y. Let us show that in both cases, we arrive at a contradiction. We denote i the predecessor of j on this path.

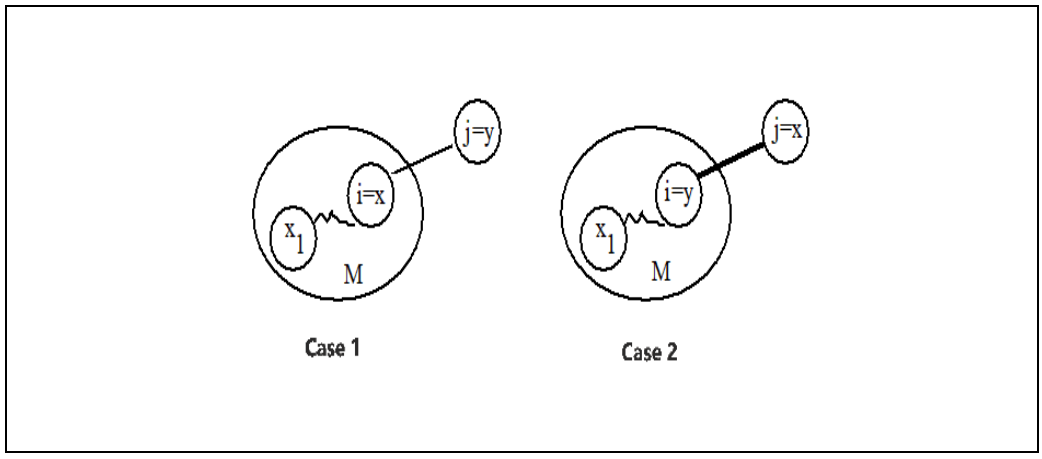


Figure 2.12 : The two possible cases.

In the first case (see Figure 2.12), the chain up to y has an odd length. So when we examine x, we mark y (the edge (x, y) is thin because the chain is alternating). This contradicts the fact that j = y is not marked. In the second case (see Figure 2.12), the chain up to x has an even length.

So when we examine y, we mark x (the edge (x, y) is thick because the chain is alternating). This contradicts the fact that j = x is not marked. Therefore, when we reach step 2, we have marked all vertices connected to an unsaturated vertex $x1 \in X$ by an alternating chain.

We have also seen that the vertices in $M \cap Y$ with this property are saturated. It follows that there is no augmenting path, that is, no alternating chain connecting an unsaturated vertex $x1 \in X$ to an unsaturated vertex $y \in Y$. The matching is therefore maximal. Q.E.D.

2.6. Other graph algorithms:

Tarjan proposed linear-time (O(M)) algorithms for finding connected components (see Section D) and strongly connected components. He also developed a linear-time algorithm for recognizing a planar graph.

In the next chapter, we will discuss the complexity of finding a Hamiltonian circuit, coloring a graph, and the stability number.

3. COMPLEXITY OF COMBINATORIAL PROBLEMS:

3.1. What is combinatorial optimization?:

Until the 19th century, the focus was on **problems of existence.** The question being addressed was: Does an object with certain properties exist?

Then, combinatorial **enumeration problems** (combinatorial analysis) were solved. The question being addressed was: How many objects have certain properties?

With the advent of computers, **combinatorial optimization problems** are now being tackled. The goal is to determine an object with certain properties, and algorithms are developed to compute such objects.

3.2. Different types of problems:

We present below a list of combinatorial problems:

Shortest path problem $O(n^3)$

Given a weighted graph G and a root node x0, determine the minimum values of paths from x0 to all vertices in the graph.

Smallest number problem O(n):

Determine the smallest number among n numbers.

Sorting problem O(nlogn):

Arrange n numbers $a_1, a_2, ..., a_n$ in ascending order.

Minimum spanning tree problem $O(n^3)$

Given a weighted, undirected graph G = (X, U), determine a connected graph with minimal total cost.

Assignment problem $O(n^3)$ (Hungarian algorithm):

Given a matrix W(n, n) representing assignment costs for n individuals to n jobs, assign each individual to a different job with minimal total cost.

	T1	T2	T3
P1	6	5	8
P2	15	20	14
P3	8	5	3

Figure 3.1 : Cost matrix.

For the cost matrix shown in Figure 3.1, the assignment (P1, T1) (P2, T2) (P3, T3) has a cost of 29. The assignment (P1, T2) (P2, T1) (P3, T3) is optimal and has a cost of 23.

Partition problem:

Given n numbers a1, a2, ..., an, is it possible to partition these n numbers into two subsets of equal sum?

Example: For the set {8, 6, 15, 37, 42, 4}, the answer is yes, because 8 + 6 + 42 = 15 + 37 + 4 = 56.

Maximum stable set problem:

Given a graph G = (X, U), determine a stable set of G with the maximum cardinality.

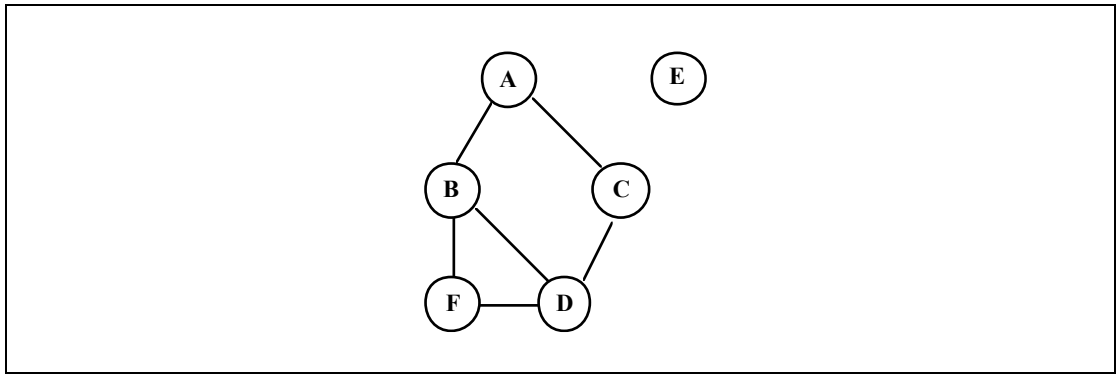


Figure 3.2 : graph.

On the graph shown in Figure 3.2, {E, A, F} is a maximum cardinality stable set.

Traveling salesman problem:

Given a weighted graph G = (X, U), determine a Hamiltonian circuit with the minimum total weight.

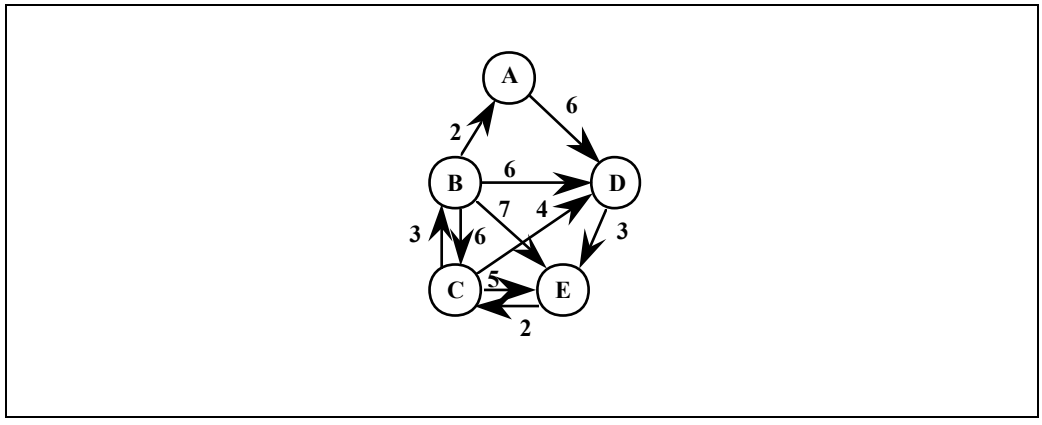


Figure 3.3 : a weighted graph.

On Figure 3.3, [A, D, E, C, B, A] is a Hamiltonian circuit with the minimum total weight.

Satisfiability problem:

Given a boolean function in conjunctive normal form (CNF), is there an assignment of variables that makes this function true?

Example:

For the given function: (x1 + x2 + ¬x3) (¬x2) (x3 + x1) (x3 + x2) = 1, the answer is yes, because x2 = 0, x3 = 1, x1 = 1 is a solution. Moreover, this solution is unique.

3.3. When is a problem considered solved?

For each problem, there is a finite number of solutions (1). It seems possible to enumerate all these solutions to determine an optimal solution (or a feasible solution). For example, for the traveling salesman problem: there are at most (n-1)! Hamiltonian circuits, for the satisfiability problem: there are at most 2^n assignments, for the partition problem: there are 2^{n-1} solutions.

It is unrealistic to rely on an enumeration method in the combinatorial field when the resulting complexity is exponential.

The following table reports the durations of enumeration, assuming a computer can calculate each solution in one microsecond (10^{-6} s).

	10	20	40	60
n	10^{-5}s	2.10^{-5}s	4.10^{-5}s	6.10^{-5}s
n^2	10^{-4}s	4.10^{-4}s	16.10^{-4}s	36.10^{-4}s
n^3	10^{-3}s	8.10^{-3}s	64.10^{-3}s	216.10^{-3}s
n^5	0,1s	3,2s	1,7 mn	13 mn
2^n	10^{-3}s	1s	12,7 days	366 centuries
3^n	0,059s	58 mn	3855 centuries	$1,3.10^{13}$ centuries
n!	3,63s	100 centuries	...	...

Efficient algorithms are mainly those with polynomial complexity (2). We can say that a problem is well solved if a polynomial algorithm (with low degree) has been found to solve it (3). We can also study the variation of the maximum size that can be processed in 1000s when we multiply the power of the machine by 10:

T(n)	Maximum size for 1000s with the first machine	Maximum size for 1000s with the second machine	Increase
100n	10	100	x 10
$10n^2$	10	32	x 3,2
n^3	10	22	x 2,2
2^n	10	13	x 1,3

(1) The finiteness of the number of solutions guarantees the existence of resolution algorithms, i.e., programs that terminate. However, this is not always the case in computer science where some problems are undecidable, meaning they cannot be solved by a program in finite time.

(2) The complexity of algorithms will be evaluated in this course by considering the worst case scenario. However, when an algorithm is frequently used, it may be preferable to analyze its average case. Unfortunately, such an analysis can be very difficult and even infeasible for moderately complex programs.

(3) In a first approximation, the coefficients of the polynomial can be neglected, as experience shows that most polynomial-time algorithms have small coefficients. However, there are exceptions where even linear-time algorithms are impractical (e.g., Robertson and Seymour's Theory).

3.4. NP-hard problems:

Polynomial-time algorithms have not been found for problems such as partition, maximum stable set, traveling salesman problem, and satisfiability, among many others. It has not been proven that there are no polynomial-time algorithms for these problems, but a result known as "If one of these problems is polynomial, then all 'reasonable' problems considered difficult are polynomial" holds. These problems are called NP-hard.

3.5. Branch and bound methods:

Branch and bound methods are used to solve NP-hard problems. However, it may not be possible to evaluate the practical complexity of these methods, as it strongly depends on the problem's input data. In fact, the complexity can be exponential for certain data, but manageable for other data sets.

3.5.1. The Queens of Gauss :

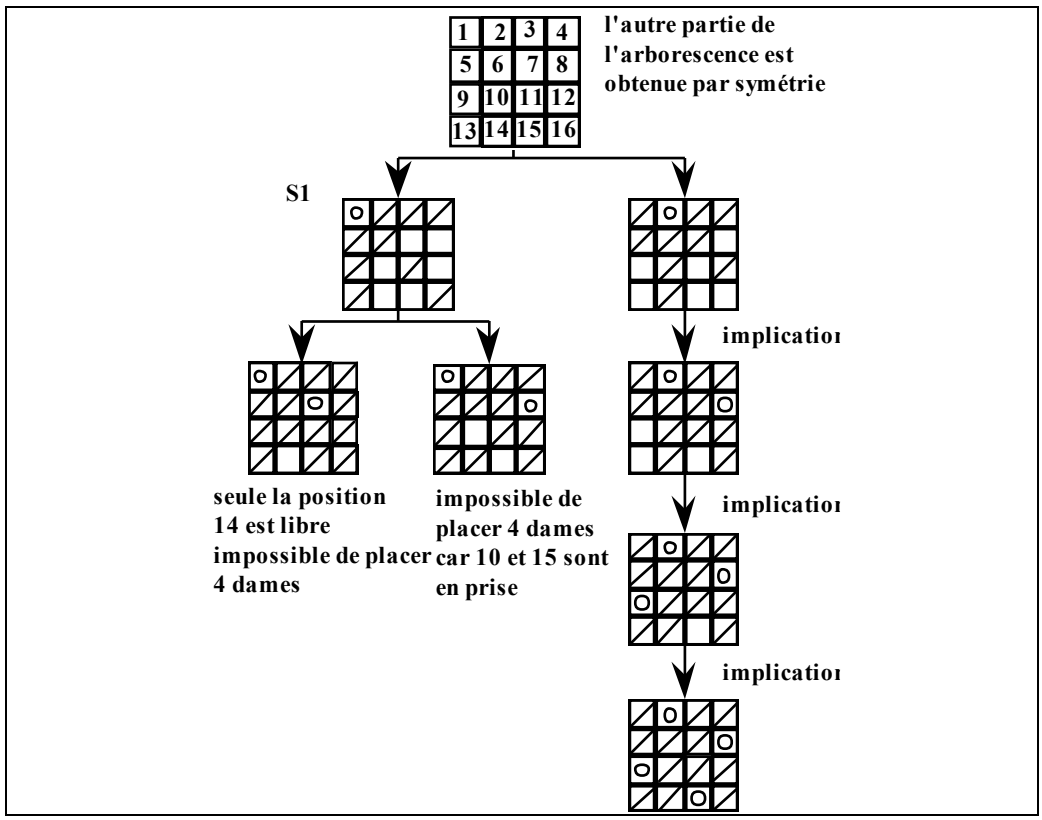


Figure 3.4 : Gauss's tree of Queens.

At the **root** corresponds the set of all solutions. At **node S1** corresponds the set of all solutions where the queen on the first row is placed on square number 1. Each node corresponds to a set of solutions.

The problem is to place **n queens** (here n = 4) on an **n×n** chessboard such that no two queens are attacking each other. To solve this problem, an **arborescence** (tree) is constructed with the root corresponding to the set of all solutions of the problem. The different possible cases are examined by constructing an arborescence (see Figure 3.4), where one solution is shown and the other can be obtained by symmetry.

3.5.2. The LITTLE method for the Traveling Salesman Problem (TSP):

A traveling salesman must visit n cities. The goal is to choose an order of traversal that minimizes the total distance traveled, i.e., finding a Hamiltonian circuit of minimal value. To solve this problem, we construct a dichotomic tree, as shown in Figure 3.5.

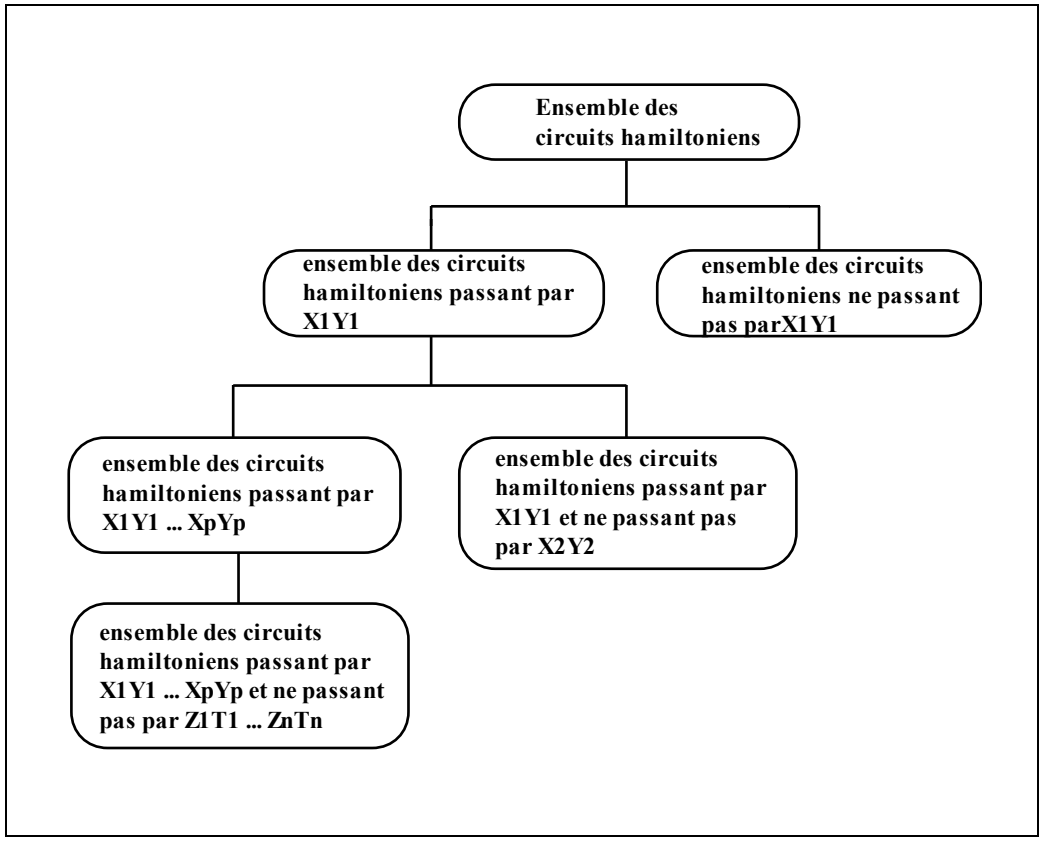


Figure 3.5 : dichotomous tree

We will use an example whose data are shown in Figure 3.6 to explain Little's method. Note that to each Hamiltonian circuit corresponds exactly one element per row and one element per column of the adjacency matrix. Therefore, the problem is not modified if we subtract the minimum of each row from each row, and then subtract the minimum of each column from each column.

	A	B	C	D	E	F
A	∞	6	7	3	1	3
B	7	∞	8	2	9	7
C	5	10	∞	10	1	7
D	8	6	5	∞	5	1
E	7	7	6	7	∞	4
F	9	8	8	5	3	∞

Figure 3.6 : Valued adjacency matrix

By subtracting row-wise, we obtain the middle table in Figure 3.7, and by subtracting column-wise, we obtain the table on the right.

	A	B	C	D	E	F	
A	∞	6	7	3	1	3	1
B	7	∞	8	2	9	7	2
C	5	10	∞	10	1	7	1
D	8	6	5	∞	5	1	1
E	7	7	6	7	∞	4	4
F	9	8	8	5	3	∞	3
							12

	A	B	C	D	E	F	
A	∞	5	6	2	0	2	
B	5	∞	6	0	7	5	
C	4	9	∞	9	0	6	
D	7	5	4	∞	4	0	
E	3	3	2	3	∞	0	
F	6	5	5	2	0	∞	
	3	3	2	0	0	0	8

	A	B	C	D	E	F
A	∞	2	4	2	0	2
B	2	∞	4	0	7	5
C	1	6	∞	9	0	6
D	4	2	2	∞	4	0
E	0	0	0	3	∞	0
F	3	2	3	2	0	∞

Figure 3.7 : Appearance of zeros by row and by column.

The **Hamiltonian circuit** will have a value greater than **20** because we subtracted 12 from rows and 8 from columns. 20 is a default evaluation for the optimal value of the initial matrix. We will work with the reduced matrix, for which all Hamiltonian circuits have a translated value of 20. We aim to construct a Hamiltonian circuit with a value of zero (which is necessarily optimal as the values are positive). To build the enumeration tree, we will choose arcs with a valuation of 0.

To select one of these arcs, we introduce the **notion of regret**. We will choose a zero with **maximum regret**. For example, if we do not take arc **AE**, we will necessarily take another element from row A and another element from column E. The additional cost here, which is 2, is called the regret. This regret is calculated by summing the minima from the corresponding row and column (excluding the 0 on which we calculate the regret). Indeed, if we do not pass through this arc with a value of 0, we will at best pass through two arcs with the minimum cost from the corresponding row and column, incurring the regret cost.

	A	B	C	D	E	F
A	∞				0^{2+0}	
B		∞		0^{2+2}		
C			∞		0^{1+0}	
D				∞		0^{2}
E	0^{1}	0^{2}	0^{2}		∞	0^{0}
F					0^{2}	∞

Figure 3.8 : Matrix of regrets

The regrets for the different 0s are shown in Figure 3.8. It is observed that BD has the highest regret. We choose to take the arc BD and introduce two new nodes in the enumeration tree (see Figure 3.9). The node BD corresponds to the set of Hamiltonian circuits that include the arc BD, while the node ¬BD corresponds to the set of Hamiltonian circuits that do not include the arc BD. The default evaluation for node BD is 20, and the default evaluation for node ¬BD is 24, which is the sum of 20 and the regret.

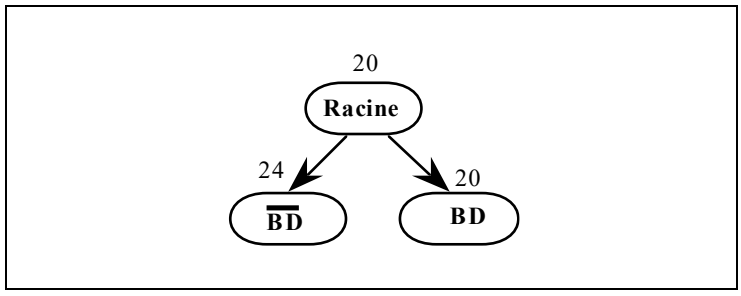


Figure 3.9 : The current tree structure after the first split

In the matrix associated with the node BD, we need to forbid the arc **DB** by setting **M(D, B) =** $+\infty$, in order to avoid the suboptimal circuit [D, B, D], and then repeat the process. In figure 3.10, we have on the left-hand side the matrix associated with the node BD in the tree, and on the right-hand side the regrets obtained by eliminating row D and column B.

	A	B	C	D	E	F
A	∞	2	4	~~2~~	0	2
B	~~2~~	~~∞~~	4	~~0~~	~~7~~	~~5~~
C	1	6	∞	~~9~~	0	6
D	4	∞	2	~~∞~~	4	0
E	0	0	0	~~3~~	∞	0
F	3	2	3	~~2~~	0	∞

	A	B	C	E	F
A				0^2	
C				0^1	
D					0^2
E	0^1	0^2	0^2		0^0
F				0^2	

Figure 3.10 : Matrices associated with the node

We choose a 0 with the highest regret and introduce two new nodes in the current tree (see Figure 3.11), AE and ¬AE.

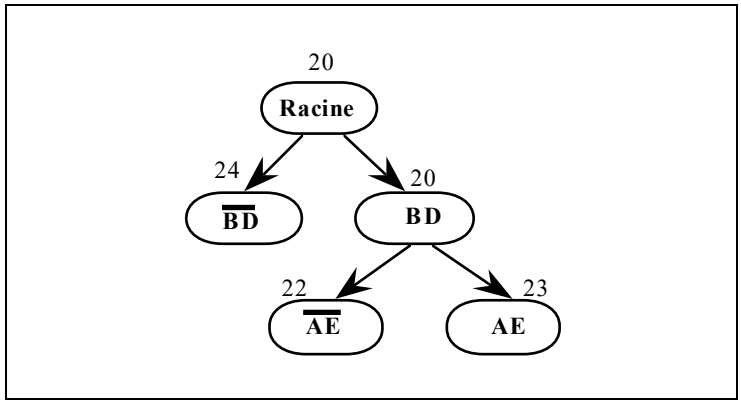


Figure 3.11 : Next state of the current tree

The matrix associated with the node AE after prohibiting EA is shown in figure 3.12. There is no zero per row, so we will subtract the minimum valuation from each row and column. Therefore, a total of 3 is subtracted. This explains why the valuation of node AE is 23, while the valuation of node ¬AE is 22.

	A	B	C	F	
C	1	6	∞	6	1
D	4	∞	2	0	0
E	∞	0	0	0	0
F	3	2	3	∞	2

Figure 3.12 : new matrix with **EA** = ∞

Let's present the LITTLE algorithm.

LITTLE Algorithm:

(1) Initialization:
Generate **zeros** (at least one per row and one per column)
{The default evaluation of the root of the tree is the sum of the subtracted numbers}.
Set **f0 =** ∞ {f0 will be the value of the best known solution}.

(2) Main loop:
While the node S with the smallest default evaluation f(S) is such that f(S) < f0 do {breadth-first}

- Evaluate the regret of each zero in the matrix associated with **S** and keep track of **(i, j)** corresponding to a zero with maximum regret.
- Introduce the nodes **(S + (i, j))** and **(S +** ¬**(i, j))** while forbidding the creation of a sub-tour in (S + (i, j)), generating zeros in both associated matrices, and setting their default evaluations.
- If the node (**S + (i, j))** corresponds to a Hamiltonian circuit, **record** the circuit and calculate its value f1.
- Set **f0 = min(f0, f1)**

End while.

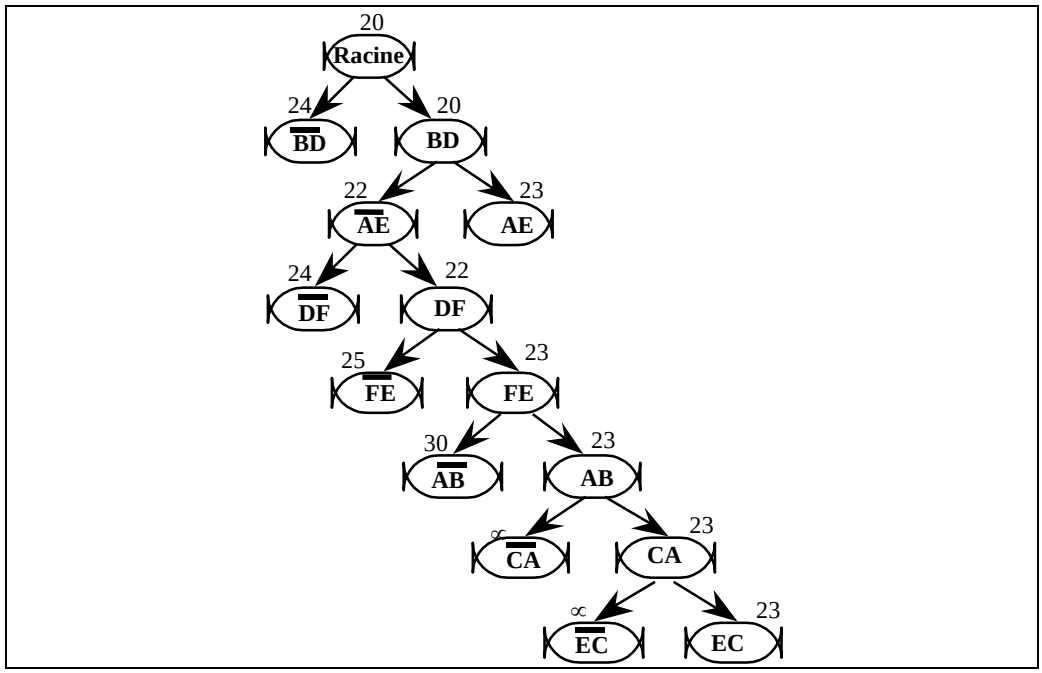


Figure 3.13 : Global tree

Continuation of the example : On the figure 3.13, the tree after applying the algorithm is shown. We provide comments below on its construction.

On the figure 3.14, we have the matrix associated with the ¬AE vertex, which is the pendant vertex of the current tree with the smallest default evaluation, along with its regrets.

The highest regret is obtained for the DF arc. We prohibit the FB arc to avoid creating a parasitic circuit.

	A	B	C	E	F
A	∞	0	2	∞	0
C	1	6	∞	0	6
D	4	∞	2	4	0
E	0	0	0	∞	0
F	3	2	3	0	∞

	A	B	C	E	F
A		0^0			0^0
C				0^1	
D					0^2
E	0^1	0^0	0^2		0^0
F				0^2	

Figure 3.14 :matrices associated with the node ¬AE.

So, there are 2 new nodes in the current tree, which are DF and ¬DF. DF is the pendant node with the smallest default evaluation. Figure 3.15 shows the matrix associated with this node and its regrets. The largest regret is for the arc FE.

	A	B	C	E
A	∞	0	2	∞
C	1	6	∞	0
E	0	0	0	∞
F	3	∞	3	0

	A	B	C	E
A		0^2		
C				0^1
E	0^1	0^0	0^2	
F				0^3

Figure 3.15:matrices associated with the vertex DF.

We need to prohibit the parasitic circuit described in Figure 3.16 by placing ∞ in the cell corresponding to EB.

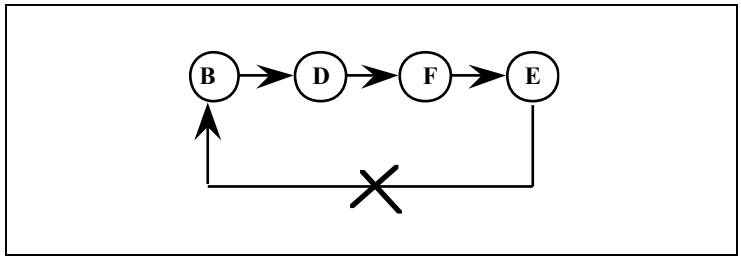


Figure 3.16 : parasite circuit

Having prohibited the arc (EB), new zeros need to be introduced in the matrix associated with the node FE (See figure 3.17). To obtain a zero, 1 is subtracted and thus 1 is added to the evaluation. AB is then the zero with the highest regret. For the matrix associated with the node AB, the parasitic circuit from figure 3.18 must be prohibited.

	A	B	C
A	∞	0	2
C	1	6	∞
E	0	∞	0

(-1)

	A	B	C
A	∞	0^7	2
C	0^5	5	∞
E	0^0	∞	0^2

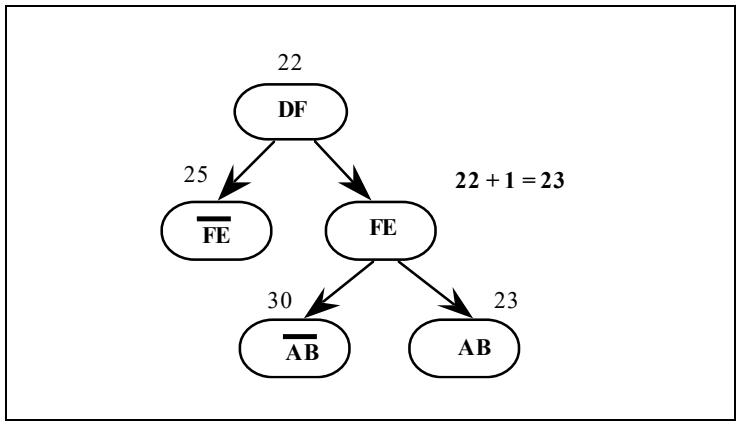


Figure 3.17 : separation of FE.

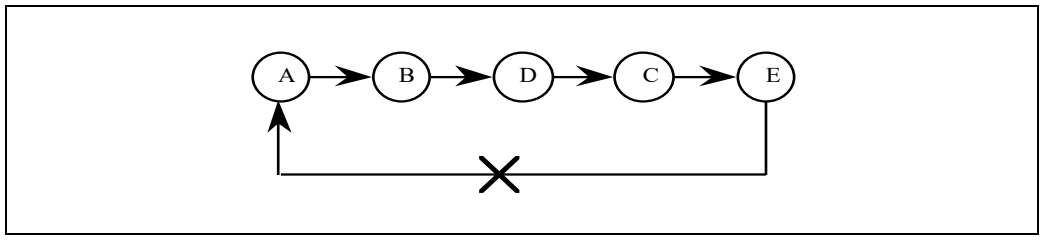


Figure 3.18 : To eliminate the parasite circuit from the node AB.

On figure 3.19, the choice of arcs CA and EC is mandatory. Thus, the Hamiltonian circuit of value 23 is obtained as shown in figure 3.20. This Hamiltonian circuit is optimal as all the dangling nodes of the global tree have an evaluation greater than or equal to 23.

Discussion:

Currently, the most effective method for solving the Traveling Salesman Problem is not the LITTLE algorithm, which can optimally handle examples with around 40 cities. Symmetric and asymmetric cases are distinguished. In the asymmetric case, methods that use linear assignment coupled with Lagrangian relaxation (see RO04) for default evaluation can handle examples with around 300 cities. In the symmetric case, generalized linear programming with the introduction of integrity cuts (polyhedral methods) can handle examples with around 400 cities.

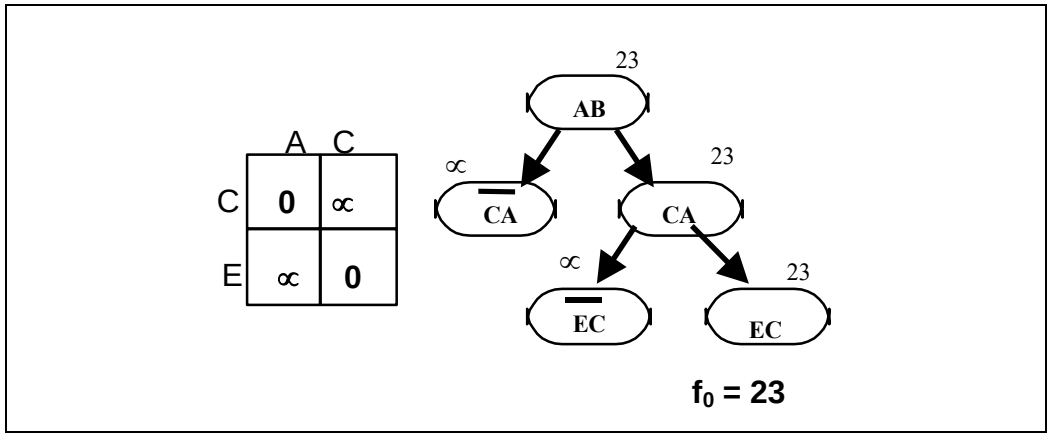


Figure 3.19 : separation of AB.

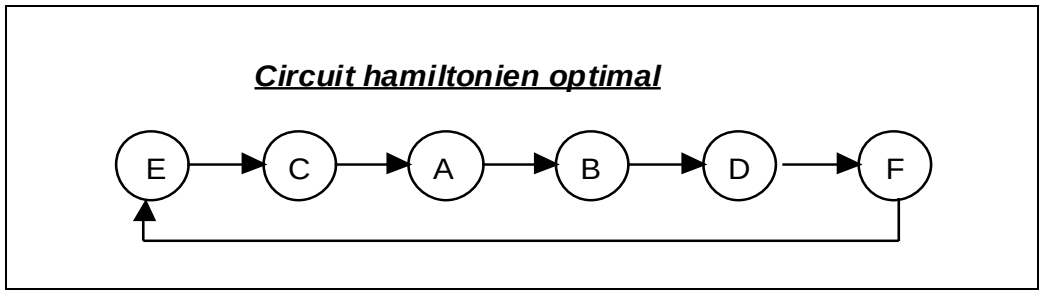


Figure 3.20 : circuit hamiltonien optimal.

3.5.3. Boolean Linear Programming: FAURE-MALGRANGE method.

We will present the FAURE-MALGRANGE method using an example to solve a linear programming problem with boolean variables. We want to solve the following linear program:

Max $F = 3x_1 + 5x_2 - x_3 + 2x_4$

sous les contraintes :

$x_1, x_2, x_3, x_4 \in \{0, 1\}$

$3x_1 + 4x_2 - 2x_3 + x_4 \leq 5$

$2x_1 - x_2 + 3x_3 - x_4 \leq 4$

$4x_1 - x_2 - x_3 + 2x_4 \leq 5$

We introduce $\neg x1, \neg x2, \neg x3, \neg x4$ defined by $xi + \neg xi = 1$. This allows us to replace the minus signs with plus signs in the inequality constraints, which become:

$3x_1 + 4x_2 + 2\neg x_3 + x_4 \leq 7$

$2x_1 + \neg x_2 + 3x_3 + \neg x_4 \leq 6$

$4x_1 + \neg x_2 + \neg x_3 + 2x_4 \leq 7$

By replacing the + signs with - signs in the objective function, we have:

$F = 10 - (3\neg x_1 + 5\neg x_2 + x_3 + 2\neg x_4)$

Note that F cannot exceed 10.

The problem is put in a form facilitating an implicit enumeration. But this time it is a maximization problem. We will therefore build a tree whose vertices will be valued by an evaluation by excess. The separation will consist of choosing a Boolean variable and fixing the truth value of this variable.

The best constructed solution will provide a default evaluation of the optimal solution. Figure 3.21 shows the global tree whose construction we comment on below.

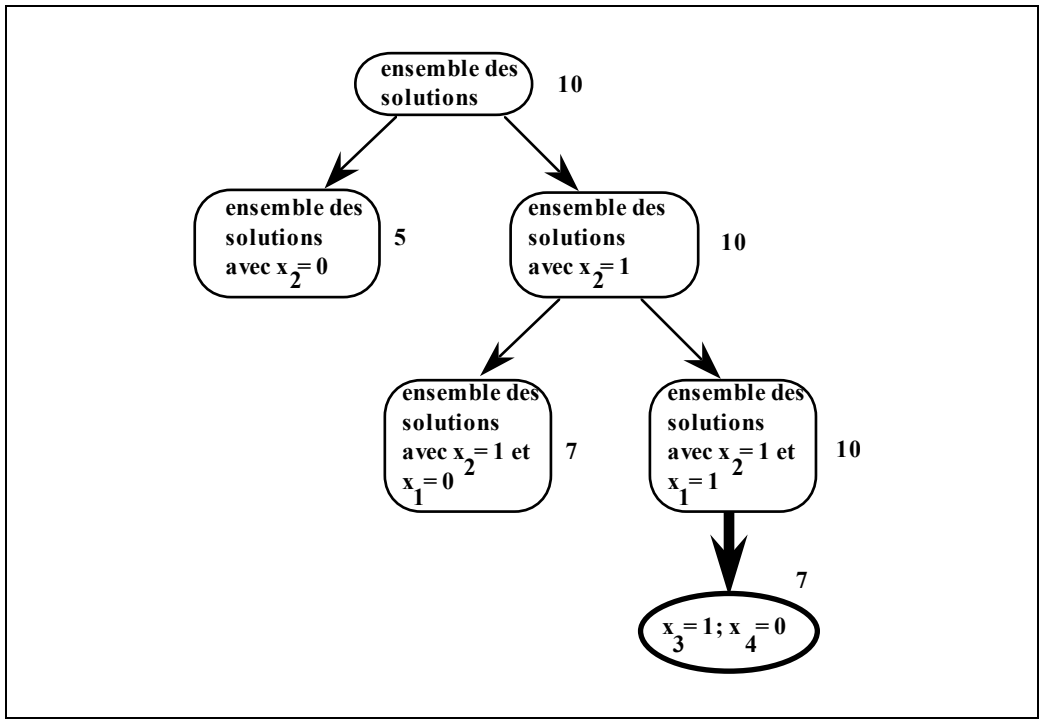


Figure 3.21 : Tree structure for searching an optimal solution.

First choice:

We choose to branch on x2 because its coefficient in the objective function is the largest (x2 has the maximum regret). At the node x2 = 1 in the tree, new constraints are added:

$3x_1 + 2\neg x_3 + x_4 \leq 3$

$2x_1 + 3x_3 + \neg x_4 \leq 6$

$4x_1 + \neg x_3 + 2x_4 \leq 7$

$F = 10 - (3\neg x_1 + x_3 + 2\neg x_4)$

We branch on this node:

Second choice:

We choose to branch on x1 because its coefficient in the objective function is the largest (x1 has the maximum regret). At the node x1 = 1 in the tree, new constraints are added:

$2\neg x_3 + x_4 \leq 0$

$3x_3 + \neg x_4 \leq 4$

$\neg x_3 + 2x_4 \leq 3$

$F = 10 - (x_3 + 2\neg x_4)$

We then have the implications x3 = 1 and x4 = 0. Thus, we have found a solution with a value of 7. This solution is optimal as all the overestimations (evaluations par excès) of the leaf nodes in the tree are lower than 7.

Here, the advantage of having inequalities with positive terms becomes apparent. In fact, if $\Sigma a_i y_i \leq b$ with yi as Boolean variables, ai as positive coefficients, and for some i, ai > b, then necessarily yi = 0. In this case, we have traversed the tree using a depth-first search approach, choosing to branch on the last introduced node. Such a policy can be managed using a stack, which further reinforces its usefulness.

3.5.4Linear programming with integer variables :

In the course RO04, the simplex method is presented as an efficient approach for solving linear programming problems with rational variables. However, when the variables are required to be integers, the problem becomes NP-hard. It can be solved either by cutting plane methods (see RO04), or by tree-based methods.

Consider the following linear programming problem with integer variables:

Max F = 3x1 + 8x2

x1 + 4x2 ≤ 20

x1 + 2x2 ≤ 11

3x1 + 2x2 ≤ 19

4x1 - x2 ≤ 22 where x1, x2 ∈ ℕ**.**

A first method to solve this type of problem is to bound the variables in order to convert them into Boolean variables. The first constraint implies x2 ≤ 5. A linear combination of the third and fourth constraints leads to x1 ≤ 5. We then introduce the following substitutions: x1 = 4X11 + 2X12 + X13 and x2 = 4X21 + 2X22 + X23, which allows us to convert the variables into Boolean variables and use the FAURE-MALGRANGE method.

A second method is to directly utilize the fact that the variables are bounded and sequentially branch on x1 and x2. The resulting tree, as shown in Figure 3.22, has evaluations as upper bounds, and a depth-first traversal yields two solutions: x2 = 5, x1 = 0 and x2 = 4, x1 = 3. The second solution with a value of 41 is optimal as the child nodes of the tree have lower evaluations.

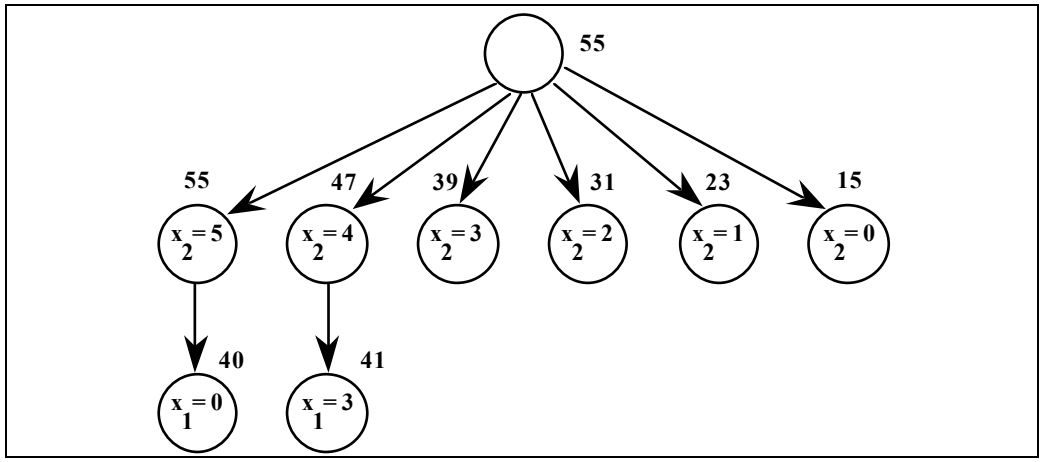


Figure 3.22 : Global tree

3.6. Summary :

a) In a tree-based method, we construct a tree whose vertices correspond to subsets of solutions. The union of the successors of a vertex is equal to that vertex. The terminal vertices of this tree are the solutions of the problem. This is called implicit enumeration because evaluations and implications allow us to enumerate only a limited number of solutions.

b) Each vertex S of the tree is associated with an evaluation f(S), which is by default for a minimization problem, and by excess for a maximization problem.

c) We also have a global evaluation f0, which is by excess for a minimization problem and by default for a maximization problem. f0 is often the value of the best solution constructed.

d) The choice of the vertex to be branched is essential. The main strategies are depth-first and breadth-first.

e) In depth-first, we will branch the deepest vertex of the tree. Consequently, we will traverse the tree branch by branch. The current tree will be a path, and therefore manageable with a stack. This strategy is the most commonly used for this reason.

f) In breadth-first, we will branch the vertex with the smallest evaluation in the tree. Consequently, we will traverse the tree in an anarchic manner. The idea is to branch a promising vertex.

g) Branching allows us to introduce successor vertices for the chosen vertex. To be effective, the default evaluation for a minimization problem (respectively, the excess for a maximization problem) must increase (respectively, decrease) for these introduced vertices.

h) Implications, also known as dominance rules, make the method more efficient.

3.7. Heuristic Methods:

Tree-based methods are efficient methods for solving many problems. However, unlike polynomial algorithms, their efficiency strongly depends on the problem data. Moreover, they are cumbersome to implement and often only allow for optimal solutions for small problem sizes. For these reasons, it is often necessary to use heuristic methods that allow for the construction of solutions to a problem but no longer guarantee optimality.

Let's explain a simple heuristic for the traveling salesman problem. We start by choosing a starting city. The second city chosen is then the closest neighboring city. Once a certain number of cities are chosen, we choose the nearest unselected city to the last introduced city and iterate until we return to the starting point. This naive heuristic obviously gives very poor results in practice. That's why neighborhood methods are introduced.

Let's first introduce descent methods. The idea is to define a neighborhood relation on the set of Hamiltonian circuits in such a way that it is easy to calculate the best neighbor of a vertex. We then start from the Hamiltonian circuit constructed using the naive heuristic. Then we calculate its best neighbor, and if it has a lower cost than the initial tour, we keep it. We iterate, descending, until we obtain a tour whose all neighbors are worse than itself. We have then found a local optimum with respect to the neighborhood relation.

Note that there is no reason for this local optimum to be a global optimum. However, these descent methods are extremely effective when the neighborhood relation is well chosen. For example, for a symmetric traveling salesman problem, the K-opt method of LIN and KERNINGHAN allows to quickly find solutions close to 2 or 3% of the optimal solution, even for large problems (e.g., 1000 cities).

However, the results of neighborhood methods are rarely as good. It has been empirically observed that the traveling salesman problem is an NP-hard problem for which it is relatively easy, most of the time, to construct good solutions. This explains why, for practically more difficult problems, we try to escape from local optima. There are three main techniques to achieve this: simulated annealing, tabu search, and genetic algorithms.

In simulated annealing, when the neighbor of a vertex is better, we descend as previously described. On the other hand, if the neighbor of a vertex is worse, we may retain it instead of the current vertex with a certain probability, denoted as p. This probability decreases over time in order to allow for many ascents at the beginning of the algorithm and very few ascents towards the end. Technically, a decreasing parameter T, called temperature, allows for this.

We have: [p = 1 - exp (-T)]. The disadvantages of simulated annealing are as follows: there is no guarantee of optimality, at least within a reasonable amount of time; it is often computationally expensive; and finally, the temperature is difficult to tune.

In tabu search methods, if we can no longer descend, we also allow for ascents to escape from a local minimum. However, there is a risk of cycling. To avoid this, we maintain a dynamic list of limited length containing a list of recently made modifications. This list is managed according to the first-in, first-out rule. Tabu search methods have similar drawbacks to simulated annealing.

Lastly, let's mention genetic methods. In this case, at each iteration, we keep a population of more than 1 solution. We then choose a subset of the population, often of size 2, and apply a genetic rule to construct a new solution from this subset. We iterate, putting this new solution (birth) into the population and removing another solution (death). These methods also do not guarantee optimality.

The performance of these three methods depends on the problems and the skill of their designer. Therefore, it cannot be said that one is better than the others. They are effective if a good and computationally efficient neighborhood relation can be defined.

4. ROUTING PROBLEMS:

4.1. Introduction :

In this chapter, we will study polynomial algorithms that solve fundamental routing problems. G = (X, U, v) (v a mapping from U to R) will be a valued graph, with $X = \{x_0, x_1, ..., x_{n-1}\}$. Moreover, for convenience, we assume that x0 is a root of G (therefore, there exists a path from x0 to any $x_i \in X$). We mainly consider the following fundamental problem:

Determine the minimal values (if they exist) of the paths originating from x_0, and then calculate a minimal path going from X_0 to X_i (for all i).

We will see that this problem is solved in $O(n^3)$ in the general case (FORD algorithm), in O(m) in the specific case of graphs without cycles (BELLMAN algorithm), and in $O(n^2)$ in the specific case of graphs whose arcs are positively valued (DIJKSTRA algorithm).

4.2. Properties of shortest paths:

Recall that the value of a path is the sum of the valuations of its arcs. A path from x_0 to x_i is of minimal value if its value is smaller than that of any other path from x_0 to x_i.

Lemma 1

Any sub-path of a path of minimal value is also a path of minimal value.

Proof

Let μ be a path from x_i to x_j and let μ' be a sub-path of μ from x_k to x_l. Then, $\mu = \mu_1\mu'\mu_2$ where

$\mu_1 = [x_i, x_k]$ and $\mu_2 = [x_l, x_j]$. Let's suppose that μ has minimal value. If μ' is not of minimal value, there would exist a path μ" of strictly lower value from x_k to x_l, and the path μ1μ"μ2 would have a strictly lower value than μ, which contradicts the fact that μ has minimal value. Q.E.D.

This very simple property is the basis of dynamic programming (see tutorial): for problems involving paths, global optimization is the result of local optimizations.

Lemma 2

A necessary and sufficient condition for there to exist a path of minimal value from x_0 to x_i for all i is that the graph G has no circuit of strictly negative value (such circuits are called absorbing).

Proof : Suppose there exists a circuit $\mu_1 = [x_i, x_i]$ of strictly negative value and $\mu = [x_0, x_i]$ is a path of minimal value from x_0 to x_i. Then, the value of the path $\mu' = \mu\mu 1$ is strictly lower than that of μ, which contradicts its minimality. Conversely, suppose G has no absorbing circuit. From any path from x_0 to x_i, we can extract an elementary path of lower value (if $[x_0, x_i]$ is not elementary, we successively remove the different circuits it uses). Therefore, we can limit the search for minimal paths to that of elementary minimal paths. As the number of elementary paths from x_0 to x_i is finite, there exists an elementary path of minimal value from x_0 to x_i. Q.E.D. The absorbing circuits for minimization are circuits of strictly negative value.

Similarly, if we maximize, the absorbing circuits are those of strictly positive value. We must always be careful of the existence of such circuits when searching for extreme paths.

Theorem 1 :

Let G be a graph without any circuit with strictly negative values, and let λ_i be the values of the paths between x_0 and x_i. A necessary and sufficient condition for these $\{\lambda_i / 0 \leq i \leq n-1\}$ to be the set of values of the minimum paths from x_0 is that:

1°) $\lambda_0 = 0$;

2°) $\lambda_j - \lambda_i \leq v_{ij}$, for all arc $(x_i, x_j) \in U$.

3°) The set of arcs for which the inequality in 2°) is an equality is the set of arcs belonging to minimum paths.

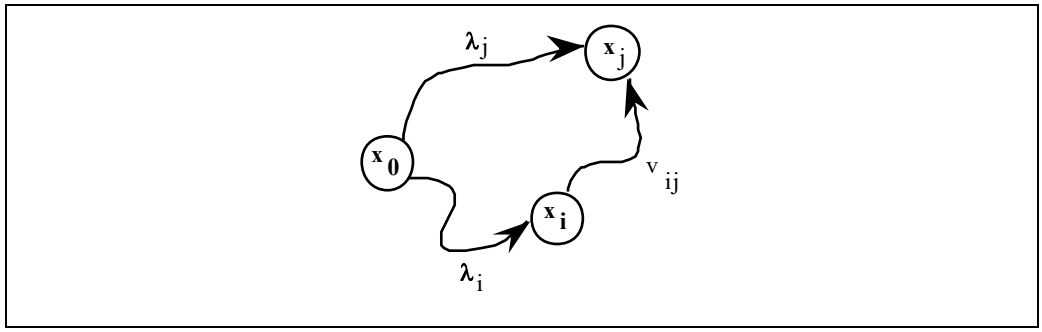


Figure 4.1 : Paths from x_0 to x_j

Proof:

The neccessary conditions:

Let λ_i be the minimum values of the paths from x_0.

1°) $\lambda_0 = 0$ (a vertex is a path of value 0).

2°) λ_j is the minimum value of a path from x_0 to x_j and is therefore less than $\lambda_i + v_{ij}$ which is the value of a path from x_0 to x_j using the arc (xi, xj).

3°) If $\lambda_j - \lambda_i = v_{ij}$, the arc (x_i, x_j) belongs to a minimum value path from x_0 to x_j, obtained by concatenating a minimum value path from x0 to xi with the arc (x_i, x_j). Conversely: If (x_i, x_j) belongs to a minimum value path $[x_0, x_k]$, according to lemma 1, the sub-paths going from x_0 to x_j and from x_0 to x_i are of minimum value; it follows that $\lambda_j = \lambda_i + v_{ij}$, so the inequality is an equality.

The conditions are sufficient:

Assuming that the λ_i satisfy 1°) and 2°) and are values of paths; we will show that they are the minimal values of paths starting from x_0.

Let's consider a path $[x_0, x_i] = \mu$ of minimal value, also noted as: $\mu = [x_{j0} = x_0, \ldots, x_{jp} = x_i]$. We have: $\lambda_{j0} = 0$, $\lambda_{j1} - \lambda_{j0} \leq v_{j0j1}$, $\lambda_{j2} - \lambda_{j1} \leq v_{j1j2}$, ..., $\lambda_{jp} - \lambda_{jp-1} \leq v_{jp-1jp}$. By adding up, we obtain $\lambda_{jp} = \lambda_i \leq v(\mu)$. It follows that λ_i is the minimal value of a path going from x_0 to x_i. This value λ_i is indeed lower and upper bounded by the minimal value of a path between x_0 and x_i. Q.E.D.

The theorem 1 is at the basis of the algorithms presented below. We will start with a set of upper bounds of optimal values and adjust them until condition 2 is satisfied.

4.3. FORD Algorithm:

It can be used for any graph, but in certain cases, it is less efficient than other algorithms.

(i) Initialization
Set $\lambda_0 = 0$ and $\lambda_i = +\infty$ for $i > 0$.

(ii) Examination of arcs
For each vertex x_i examined, traverse the set of arcs (x_i,x_j) originating from x_i and replace λ_j with λ_i+v_{ij} if $\lambda_i+v_{ij} < \lambda_j$.

(iii) Termination test
Iterate (ii) as long as a λ_j has been modified in (ii).

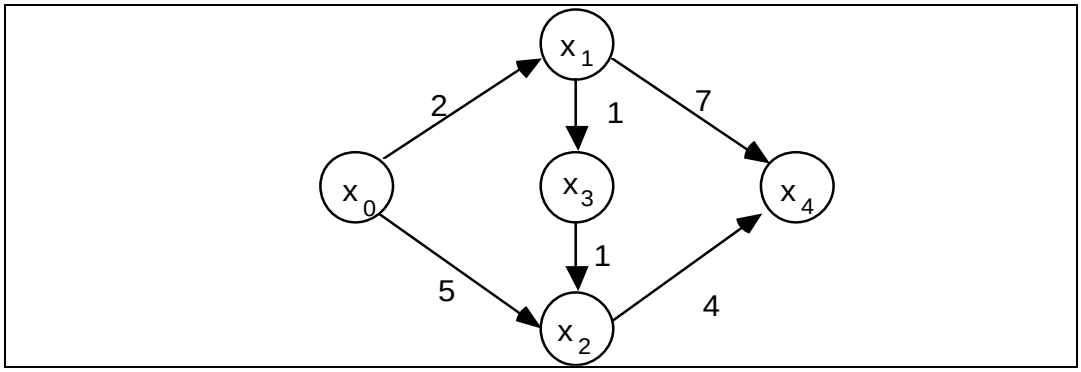


Figure 4.2 : Example

The table below reports the successive values of λ_i during the application of the FORD algorithm to the example in Figure 4.2.

The following questions must be asked:

1- Does this algorithm always give the minimum values?
2- What is the complexity of this algorithm?
3- 3- How to determine the minimum paths?

	λ_0	λ_1	λ_2	λ_3	λ_4	Arcs from
Initialization	0	∞	∞	∞	∞	
First iteration	0	2	5	∞	∞	x_0
	0	2	5	3	9	x_1
	0	2	5	3	9	x_2
	0	2	4	3	9	x_3
Second iteration	0	2	4	3	9	x_0
	0	2	4	3	9	x_1
	0	2	4	3	8	x_2
	0	2	4	3	8	x_3

	0	2	4	3	8	x_0
Third	0	2	4	3	8	x_1
iteration	0	2	4	3	8	x_2
	0	2	4	3	8	x_3
	None	λ_i	Modified	STOP		

Theorem 2 answers question 1.

Theorem 2:

Ford's algorithm computes the values of the minimum paths from x_0 when the graph has no absorbing circuit.

Proof:

Let $\lambda_i(k)$ be the value of λ_i at the end of the k^{th} iteration of the algorithm, and λ_i^k be the value of a minimal path from x_0 to x_i among the paths that take at most k arcs.

The theorem is proved by induction. We assume that a property is true after k iterations and we prove that it remains true after k+1 iterations. At the end of the kth iteration of (ii), $\lambda_i = \lambda_i(k)$ is the value of a path from x_0 to x_i and is less than or equal to the minimal value of a path from x_0 to x_i that takes at most k arcs: $\lambda_i^k \geq \lambda_i(k)$. Let $\mu_{k+1}(x_0, x_j)$ be a minimal path from x_0 to x_j among the paths that take at most k+1 arcs: $\mu_{k+1}(x_0, x_j) = \mu_k(x_0, x_i)$ concatenated with (x_i, x_j), for x_j successor of x_i; λ_j^{k+1} is the value of this path. By the induction hypothesis, $\lambda_i^k \geq \lambda_i(k)$. Using $\lambda_j^{k+1} = \lambda_i^k + v_{ij}$, we have: $\lambda_j^{k+1} \geq \lambda_i(k) + v_{ij}$ and due to the algorithm, $\lambda_i(k) + v_{ij} \geq \lambda_j(k+1)$. Hence the result. The property is therefore true at order k+1. As a result, at the n-1th iteration, the λ_i are the minimal values of the paths from x_0. Q.E.D.

Theorem 3 answers question 2.

Theorem 3:

The complexity of Ford's algorithm is: O(n x m) where n = | X | and m = | U |.

Proof:

We execute (n - 1) (ii), (ii) costs O(m); hence the overall complexity. Q.E.D. Finally, to answer question 3, we use the proposition: the arcs belonging to the minimum paths are the (x_i, x_j) such that $\lambda_j = \lambda_i + v_{ij}$. The paths of the partial graph formed by these arcs are exactly the minimum paths.

4.4. DIJKSTRA Algorithm:

This algorithm will only be applicable if the valuations of the arcs are positive or zero. In this algorithm, we will again adjust the λ_i, but this time the number of adjustments will be minimal, thus equal to the number of arcs of the graph.

Algorithm

(i) Set $S = \{x_0\}$, $\lambda_0 = 0$, $\lambda_i = v_{0i}$, if $(x_0 , x_i) \in U$, $\lambda_i = \infty$,otherwise.

(ii) (ii) While $S \neq X$ do

(a) Choose $x_i \in X - S$ with minimum λ_i.
(b) Set $S = S + \{ x_i \}$.
(c) For every $x_j \in (X - S)$, successor of x_i , set: $\lambda_j = \min \{\lambda_i + v_{ij}, \lambda_j\}$.

Example :

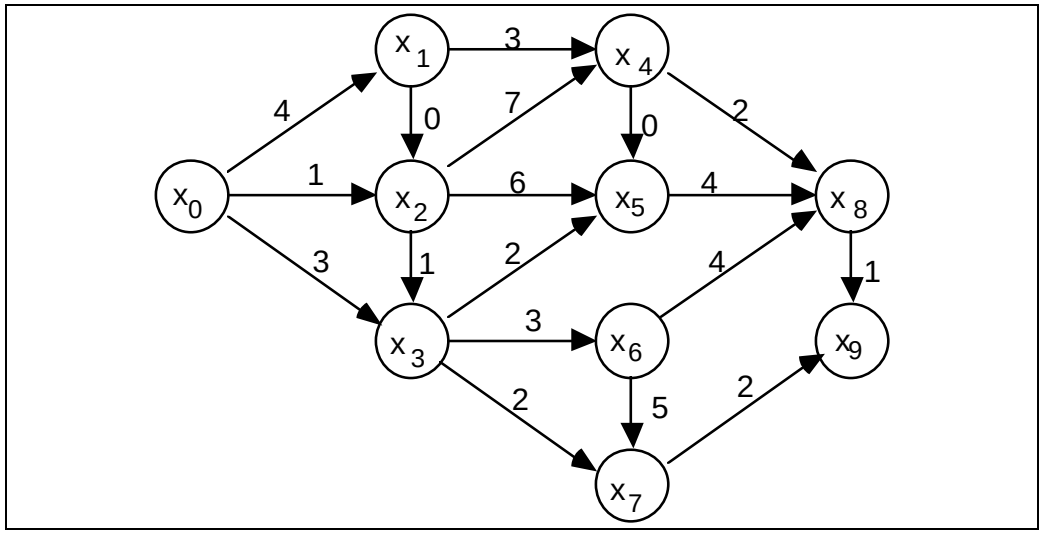


Figure 4.3: Example

	xi	S	λ0	λ1	λ2	λ3	λ4	λ5	λ6	λ7	λ8	λ9
S	{x0}	{x0}	0	**4**	**1**	3	∞	∞	∞	∞	∞	∞
S	{x2}	{x0,x2}	0	4	1	2	8	7	∞	∞	∞	∞
S	{x3}	{x0,x2,x3}	0	4	1	2	8	**4**	**5**	**4**	∞	∞
S	{x1}	{x0,x1,x2,x3}	0	4	1	2	**7**	4	5	4	∞	∞
S	{x5}		0	4	1	2	7	4	5	4	**8**	∞
S	{x7}		0	4	1	2	7	4	5	4	8	**6**
S	{x6}		0	4	1	2	7	4	5	4	8	6
S	{x9}		0	4	1	2	7	4	5	4	8	6
S	{x4}		0	4	1	2	7	4	5	4	8	6
S	{x8}		0	4	1	2	7	4	5	4	8	6

Minimum path tree :

If we define for each vertex x_j its parent as the vertex x_i ,λ_j (thus $\lambda_j - \lambda_i = v_{ij}$), we have a Shortest path tree (see figure 4.4).

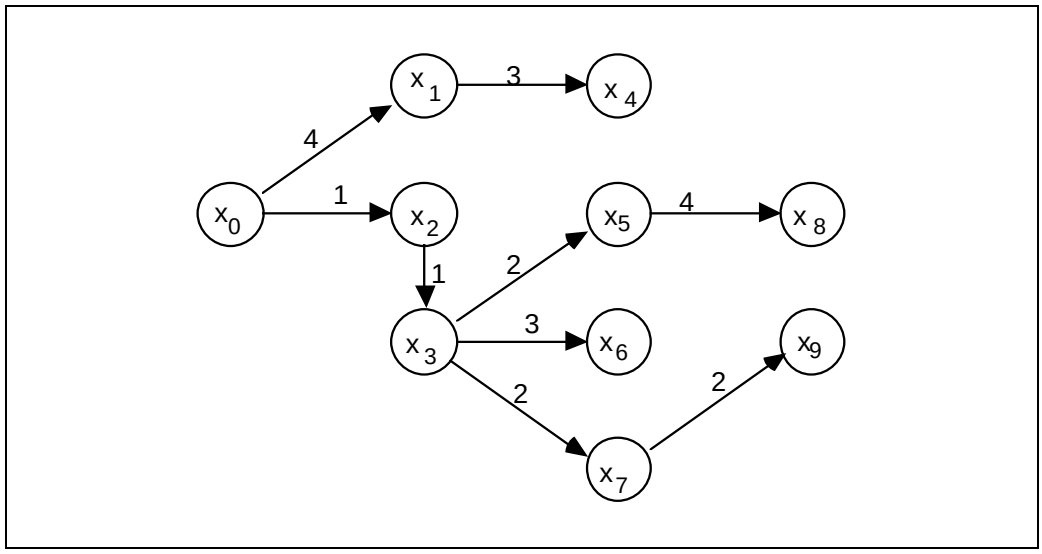


Figure 4.4: Shortest path tree

Complexity of Dijkstra's algorithm:

Lemma 3

Dijkstra's algorithm terminates in a finite number of steps and has a complexity of $O(n^2)$.

Proof:

(i)Initialization costs O(n).(ii) (a) costs O(n) and is done n times. (b) costs O(1) and is done n times. (c) costs overall O(m) because each arc is examined once. The overall complexity is therefore: $O(n^2)$. Q.E.D.

Theorem 4

The λ_i values at the end of the algorithm are the minimal values of the paths starting from x_0.

Proof

Since valuations are positive or zero, there is no absorbing circuit. Therefore, there are minimum paths. Let λ_i * be the minimum value of a path from x_0 to x_i and let us show by induction that at the end of each iteration (ii), we have:

1°) if $x_j \in S$, $\lambda_j = \lambda_j^*$,

2°) if si $x_j \in X - S$, $\lambda_j = \min \{\lambda_k + v_{kj}\}$ over $x_k \in S$, and predecessor of x_j.

The property is true at the end of initialization. Suppose the property is true at the end of the kth iteration and show that it remains true at the end of the $k+1^{st}$ iteration. Let x_i be the vertex of X-S with minimal λ_i at the beginning of the $k+1^{st}$ iteration. According to the induction hypothesis, λ_i is the minimum value of an elementary path from x0 to xi that does not pass through a vertex of X-S. We will show that $\lambda_i = \lambda_i^*$ by comparing the value of an elementary path from x_0 to x_i passing through a vertex of X-S to λ_i and using the fact that λ_i is the minimum value of a path from x_0 to x_i passing only through the vertices of S.

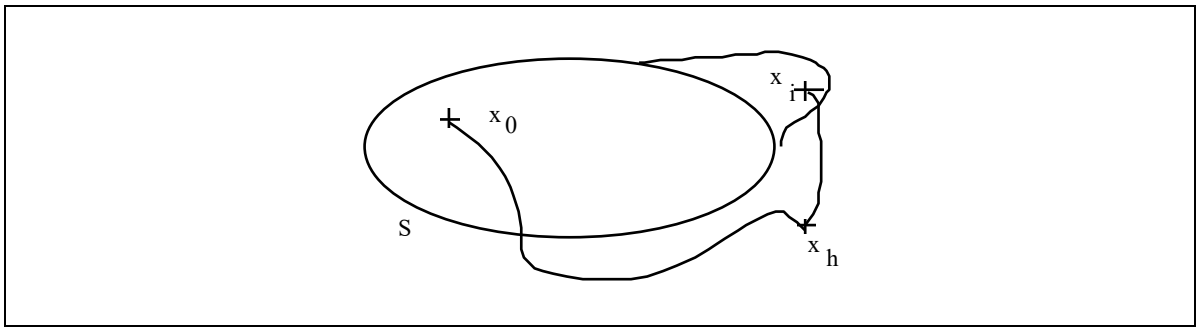


Figure 4-5: A path from x_0 to x_i

Let $\mu(x_0, x_i)$ be such a path. Let x_h be the first vertex of X-S belonging to $\mu(x_0,x_i)$. The sub-path from x_0 to x_h has a value greater than or equal to λ_h. Since the valuations are positive, the value of the path $\mu(x_0, x_i)$ is greater than or equal to λ_h, and therefore greater than or equal to λ_i, according to the algorithm ((ii) a)). It follows that $\lambda_i = \lambda_i^*$ (Figure 4.5).

<u>According to the algorithm ((ii)a))</u>

1°) is verified for the other vertices of S by induction hypothesis;

2°) results from the induction hypothesis and the adjustment in step (c) of the algorithm.

Indeed, let S' = S + $\{x_i\}$ and consider a vertex x_j not belonging to S'. Let us show that after the adjustment $\lambda_j = \min\{\lambda_k + v_{kj}\}$ for $x_k \in$ S' and predecessor of x_j. Before the adjustment, we have by induction hypothesis: $\lambda_j = \min\{\lambda_k + v_{kj}\}$ for $x_k \in$ S. We then set $\lambda_j = \min(\lambda_j , \lambda_i + v_{ij})$. Q.E.D.

4.5. BELLMAN Algorithm:

It is applicable to graphs without circuits. We have seen that we can number the vertices of a graph without circuits in O(m) operations. The number of adjustments is also minimal, equal to the number of arcs in the graph.

<u>Bellman's Algorithm :</u>

(i)Number the vertices of the graph, set $\lambda_0 = 0$.

(ii) For j = 1 to n - 1, set: $\lambda_j = \min \{\lambda_i + v_{ij}\}$ on the set of predecessors x_i of x_j.

<u>Example</u> :

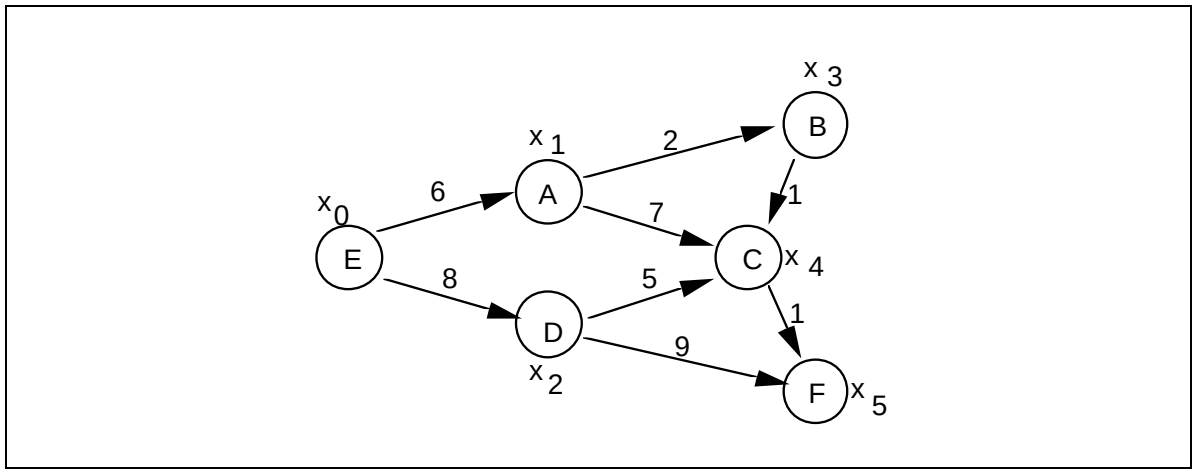


Figure 4.6 : An example of Bellman's Algorithm

$\lambda_0 = 0$. $\lambda_1 = 6$ (0+6). $\lambda_2 = 8$ (0+8). $\lambda_3 = 8$(6+2). $\lambda_4 = 9 = \min \{8+5, 6+7\}$. $\lambda_5 = 10 = \min \{9+1, 8+9\}$.

Theorem 5:

The Bellman algorithm computes the values λ_i of the minimum paths from x_0 in O(m) iterations.

Proof:

costs O(m). (ii) costs O(m) as each arc is examined exactly once. We assume by induction that, at sequence j, we have: $\lambda_1 = \lambda^*_1$, $\lambda_2 = \lambda^*_2$, ..., $\lambda_j = \lambda^*_j$, where λ_i are the minimal values of the paths from x_0 to x_i, which exist because the graph is acyclic, and thus has no absorbing circuit. The property is true at sequence j+1, since a minimal path from x_0 to x_{j+1} is obtained by adding the arc (x_k, x_{j+1}) to a minimal path from x_0 to x_k for a predecessor x_k of x_{j+1}. This demonstrates the property by induction. Q.E.D.

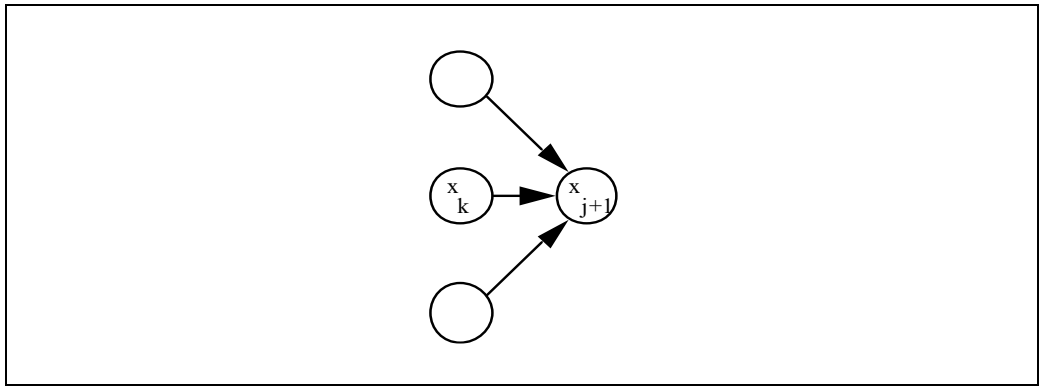


Figure 4.7 : Proof of theorem 5

4.6. Matrix method:

The matrix method allows to compute the value of the minimum paths between any pair of vertices in a graph without an absorbing circuit for a complexity of $O(n^3)$.

Algorithm

(i) Let $V^0 = (v^0_{ij})$ be the matrix of valuations of the edges in G.
$(v^0_{ij} = \infty$ if (x_i, x_j) does not belong to U)

(ii) For k = 1 to n,
For i = 1 to n,
For j = 1 to n,
do $v^k_{ij} = \min \{v^{k-1}_{ij}, v^{k-1}_{ik} + v^{k-1}_{kj}\}$

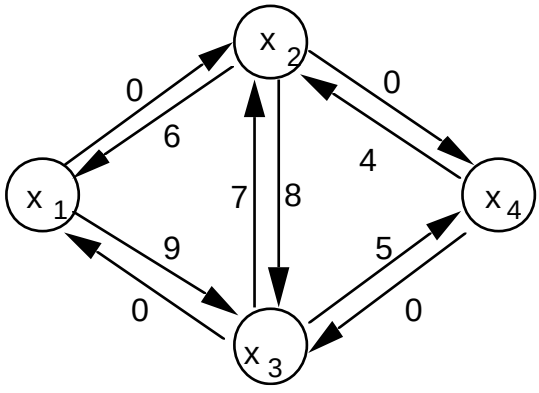


Figure 4-8: An example for Matrix method

$V^0 =$

∞	0	9	∞
6	∞	8	0
0	7	∞	5
∞	4	0	∞

$V^1 =$

∞	0	9	∞
6	6	8	0
0	0	9	5
∞	4	0	∞

$V^2 =$

6	0	8	0
6	6	8	0
0	0	8	0
10	4	0	4

$V^3 =$

6	0	8	0
6	6	8	0
0	0	8	0
0	0	0	0

$V^4 =$

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Figure 4.9 : Application of the Method

Theorem 5:

In the absence of an absorbing circuit, at the end of the k-th iteration, v^k_{ij} is the minimal value of an elementary path going from i to j and passing only through vertices 1, 2, ..., k.

Proof:

This result is immediately proven by induction. The absence of an absorbing circuit makes it possible to limit the search for minimal paths to elementary paths. An elementary path between i and j, if it exists, passing only through 1, 2, ..., k is either a path passing once through k or a path not passing through k. This explains the formula of the algorithm. Q.E.D.

Note: If there are absorbing circuits, negative terms appear on the main diagonal of one of the matrices.

4.7. Other pathfinding problems:

Other pathfinding problems can be solved similarly. We mention the following problems and their methods of resolution:

Maximum value paths: Ford and Bellman's algorithms still hold true by replacing MIN with MAX and initializing the λ_i values to $-\infty$ and λ_0 to 0.

Existence of a path from i to j: matrix method or boolean matrix raised to a power.

Counting the number of paths between i and j: M real matrix associated (number of paths of length 1), M^k (number of paths of length k).

Maximum reliability paths between i and j: The reliability of a path is the product of the valuations of the arcs on this path (reliability: probability of not breaking down). Ford or Bellman algorithms are used by replacing + with x and MIN with MAX. The λ_i values are initialized to 0, except λ_0 which is initialized to 1.

Maximum capacity paths: The maximum capacity of a path is the smallest valuation (≥ 0) on this path. Ford and Bellman algorithms are also used by replacing + with MAX and MIN with MAX. The λ_i values are initialized to -1, except λ_0 which is initialized to ∞.

5. SCHEDULING PROBLEMS :

5.1. Introduction :

Until 1958, Gantt charts (also known as bar charts) were commonly used for scheduling.

In that year, two graph-based methods were developed:

1 - The potential-tasks method, used during the construction of the FRANCE ocean liner.

2 - The PERT (Program Evaluation Review Technique) method, used during the development of the POLARIS missiles.

Both methods only consider potential constraints such as precedence and temporal localization, and do not manage resource constraints. They are based on finding maximal paths, known as critical paths. This was not too detrimental at the time, as these projects typically had a large number of tasks, were prestigious, and cost was secondary. The main goal was to complete the project as early as possible, which was facilitated by the critical path methods.

5.2. Conjunctive graphs, set of potentials:

In either method, a conjunctive graph is associated with the scheduling problem. A scheduling will be a set of potentials on the conjunctive graph.

Conjunctive graph:

> A conjunctive graph is a valued graph G = (X, U) with a root 0 and an anti-root n+1, such that there exists a positive-valued path between the root and any other vertex, and between any vertex other than the anti-root and the anti-root.

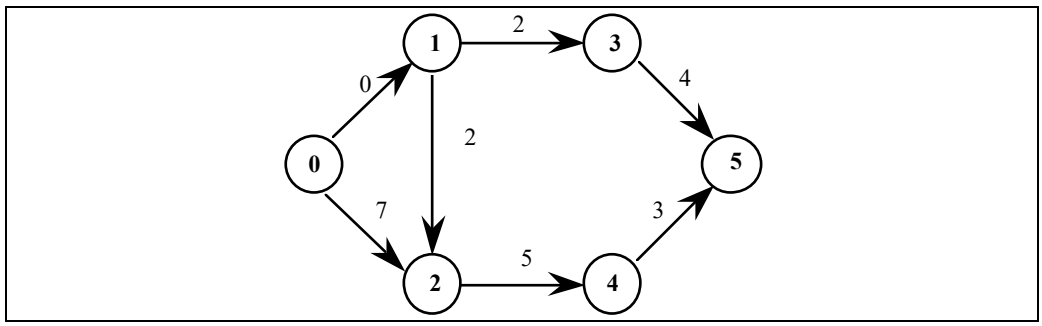


Figure 5.1 : example of conjuctive graph

Set of potentials:

A set of potentials on a conjunctive graph G = (X, U) is a function t from X to R such that:

1. t0 = 0,
2. vij ≤ tj - ti, for every arc (i, j) in U. For convenience, we write T = {ti | i ∈ X}.

Existence theorem:

A necessary and sufficient condition for the existence of a set of potentials on a conjunctive graph G = (X, U) is that the graph has no circuit with strictly positive value.

Proof: The condition is necessary:

If there exists a circuit $[i_1, i_2, ..., i_r, i_1]$ with strictly positive value, i.e., $v_{i1i2} + v_{i2i3} + \ldots + v_{iri1} > 0$, then for any set of potentials T $t_{i2} - t_{i1} \geq v_{i1i2}$, $t_{i3} - t_{i2} \geq v_{i2i3}$, ..., $t_{i1} - t_{ir} \geq v_{iri1}$.

Summing these inequalities, we get $0 > 0$, which is absurd.

The condition is sufficient:

If there is no circuit with strictly positive value, there exists a maximal path from 0 to i. Let ri be the value of such a path, and we will show that $R = \{ri / i \in X\}$ is a set of potentials. Indeed, the maximal value rj of a path from 0 to j is greater than ri + vij, which is the value of a path from 0 to j obtained by concatenating a maximal path from 0 to i with the arc (i, j). As a result, we have:

1. ro = 0,
2. rj - ri ≥ vij. Q.E.D.

Left-aligned and right-aligned set of potentials:

In the following, we assume that the graph has no circuit with strictly positive value, and we denote by l(i,j) the maximum value of a path from i to j.

We have seen that $R = \{r_i = l(0,i) / i \in X\}$ is a set of potentials (referred to as left-aligned or earliest). We will show that $F = \{ f_i = l(0,n+1) - l(i,n+1) / i \in X \}$ is also a set of potentials (referred to as right-aligned or latest). Let $t^* = l(0,n+1)$.

Proposition 1:

For any set of potentials $T = \{ti / i \in X\}$, $tn+1 \geq t^* = l(0,n+1)$. In particular, t* is the optimal duration of the scheduling. Moreover, for any i, we have: ri ≤ ti (ri: earliest start time). Finally, if T is an optimal scheduling of duration t*, then ti ≤ fi (fi: latest completion time).

Proof: Let $[i_o=0 , ... , i_r = n+1]$ be a path of maximum value between 0 and n+1, thus with a value of l(0,n+1) (such a path is called critical). $t_{i1} - t_{io} \geq v_{ioi1}$, $t_{i2} - t_{i1} \geq v_{i1i2}$, ..., $t_{ir} - t_{ir-1} \geq v_{ir-1ir}$. By summing, we obtain: $t_{ir} - t_{io} \geq l(0,n+1)$ and since i $i_r = n+1$ and $i_o = 0$, we have: $t_{n+1} \geq t^* = l(0,n+1)$.

Similarly, we can verify that $R = \{ri = l(0,i) / i \in X\}$ is a set of potentials with minimal difference between to and ti, hence ti ≥ ri. The minimal difference between ti and tn+1 is equal to l(i,n+1), therefore if $F = \{f_i = l(0,n+1)-l(i,n+1) / i \in X$ we have: $t_i \leq f_i$ si $t_{n+1} = t^*$. Q.E.D.

5.3. The potential-task method:

Tasks	Duration	Potential constraints
1	3	
2	7	
3	4	Task 1 precedes task 3
4	6	Tasks 1 and 2 precede task 4
5	5	Task 3 precedes task 5
6	3	Tasks 3 and 4 precede task 6
7	2	Task 6 precedes task 7

Figure 5-2 : example

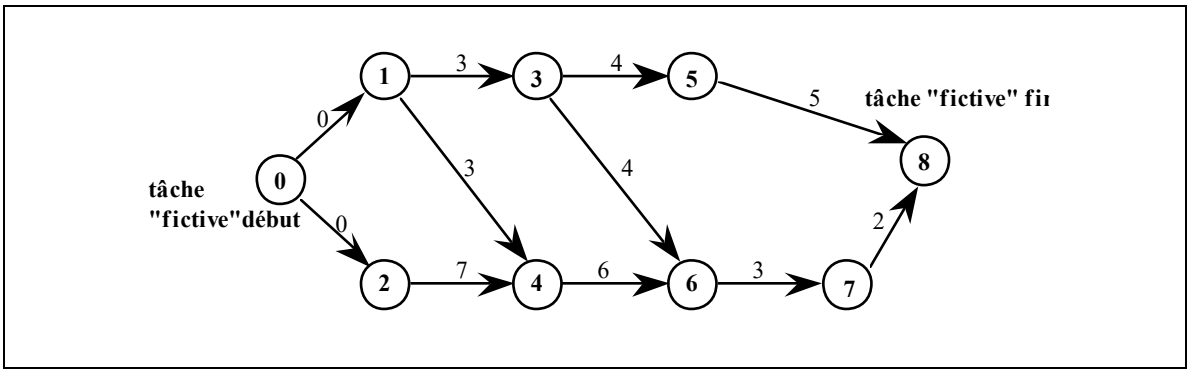


Figure 5.3 : The potential-task graph of the example

We associate canonically with this problem a graph G = (X, U) where X = {0, 1, 2, ..., n+1} (0: fictitious start task; n+1: fictitious end task), and U is associated with potential constraints, which include, on the one hand, the initial constraints, and on the other hand, the constraints due to tasks 0 and n+1: we connect any vertex without predecessor to 0 with an arc of valuation 0, and we connect any vertex i without successor to n+1 with an arc of valuation equal to the duration of task i.

The critical path:

The maximal paths between 0 and n+1, called critical paths, play a central role in this method. The tasks belonging to this critical path are called critical tasks. If a critical task is delayed by a certain delay, the scheduling will be delayed by the same delay. This is because critical tasks have the same earliest and latest start times, resulting in zero slack.

Scheduling: A scheduling is a set of potentials on the associated conjunction graph. Any scheduling has a duration greater than l(0,n+1) = t*, the value of the critical path. The most common approach is to calculate left and right-aligned schedules (earliest and latest start times) by solving two path problems: ri = l(0, i), fi = l(0,n+1) - l(i,n+1). Bellman's algorithm is typically used in the absence of cycles, or Ford's algorithm if there is a cycle, using the following formulas for earliest start times: r0 = 0, rj = max(ri + vij) (with i in U-(j)).

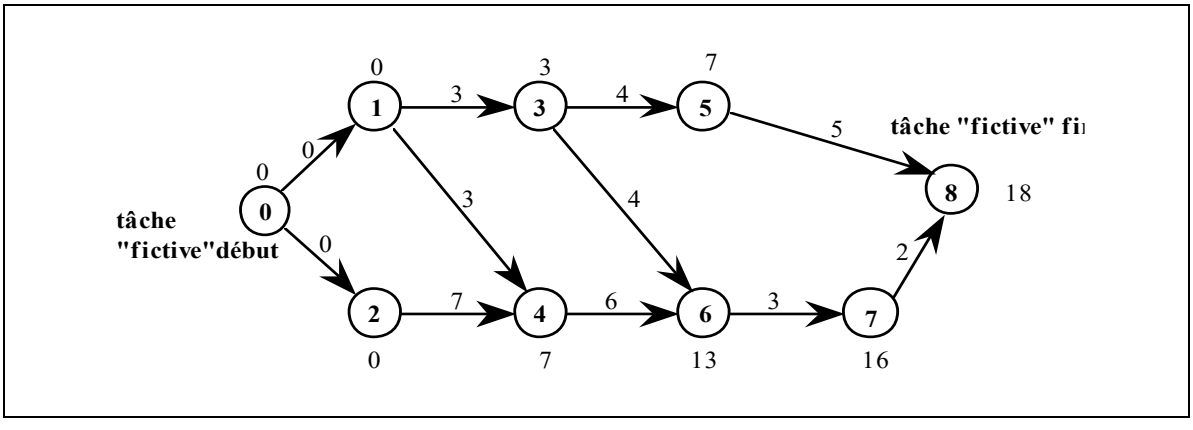


Figure 5.4 : Earliest start times

latest finish times: $f_{n+1} = t^*$, $f_i = \min (f_j - v_{ij})$ (with $j \in U^+(i)$).

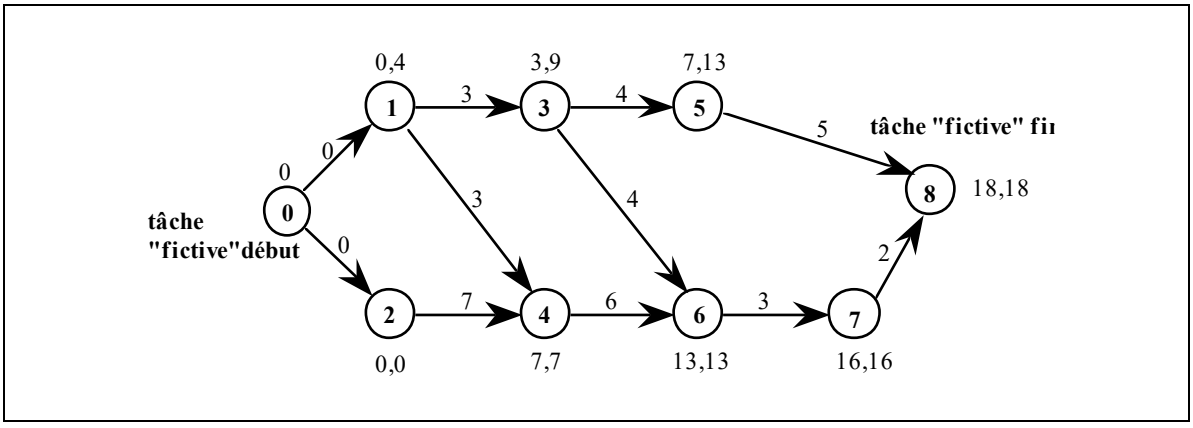


Figure 5.5 : Latest finish times

The different types of potential constraints:

We report below the different types of potential constraints, which include constraints of precedence and temporal localization.

Constraints of precedence:

i precedes j, calling ti (resp. tj) the start date of i (resp. j) and pi the duration of i, the constraint is written as tj >= ti + pi, i.e., tj - ti >= pi: we associate an arc (i, j) with a weight of pi.

Constraints of temporal localization:

Task i is available at date ai, ti >= ai or ti - t0 >= ai (t0 = 0). We associate an arc (0, i) with a weight of ai. Task i must be completed by date di: ti + pi <= di, which is written as: to - ti >= pi - di. We associate an arc (i, 0) with a weight of pi - di (this weight can be negative or zero).

Constraints of broad precedence:

Task j can start after task i with an additional setup time rij: tj >= ti + pi + rij, hence tj - ti >= pi + rij. Task j can start after task i has started by one third of its duration: tj >= ti + pi/3.

5.4. The PERT method :

We associate with each task i an event Di, representing the start of task i, and an event Fi, representing the end of task i. We also introduce an event D representing the start of the scheduling and an event F representing the end of the scheduling.

The conjunctive graph associated with this has vertices representing the set of events and arcs representing the set of tasks, along with fictitious arcs to represent the constraints.

Example :

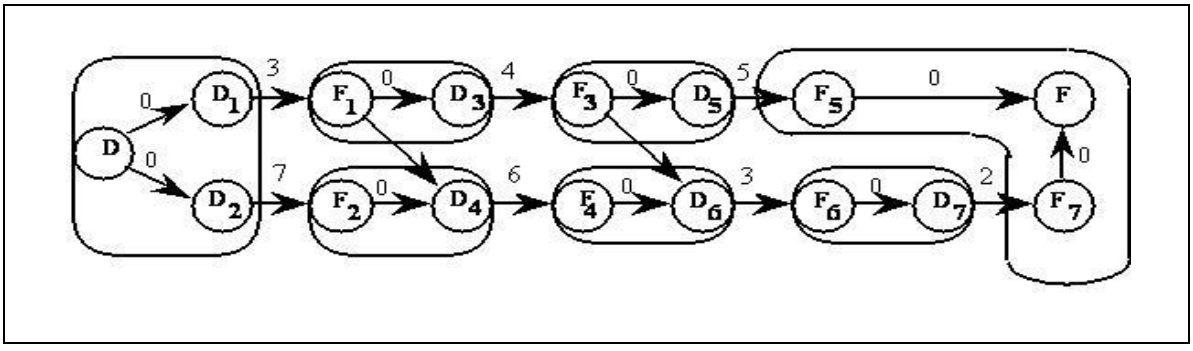


Figure 5.6 : expanded PERT graph

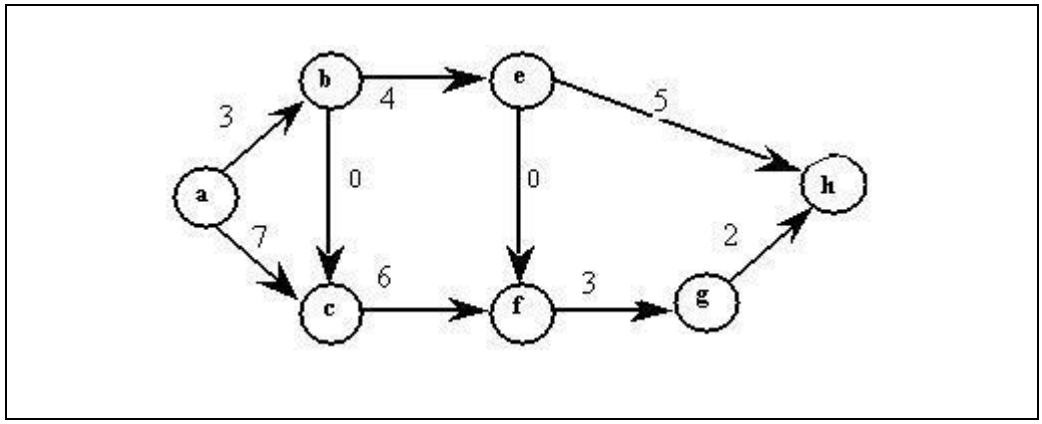


Figure 5.7 : simplified PERT graph

We can simplify the PERT graph by merging start or end events of tasks. However, the disadvantages of the simplified graph are:

• The lack of automation in its construction.

• The fact that it is not unique.

• The need to reconstruct the entire simplified graph if any constraints are modified (added or removed).

The advantage of the PERT graph is improved readability for non-specialists. In fact, a task with a certain duration is represented by an arrow.

6. MAXIMUM FLOW PROBLEM:

6.1. Introduction

In this chapter, we will study the problem of maximum flow. This problem arises when we need to maximize the amount of material transported from one or more sources to one or more destinations. It is modeled by a valued graph, called a transport network, which we define. The edges represent the possibilities of transport between two sites and are valued by their corresponding capacities. The problem of maximum flow is solved by an efficient fundamental algorithm due to FORD and FULKERSON, which we present and justify below. Finally, we discuss the polynomiality of this problem by referring to more recent research that has led to the development of more efficient algorithms.

6.2. **Transport Network:**

Definition :

We call a transport network a positively valued graph without loops, having a root s, a sink p, and containing the arc (p,s) of infinite valuation. The valuations of the arcs are called capacities.

An example of a transport network is reported in Figure 6.1 below.

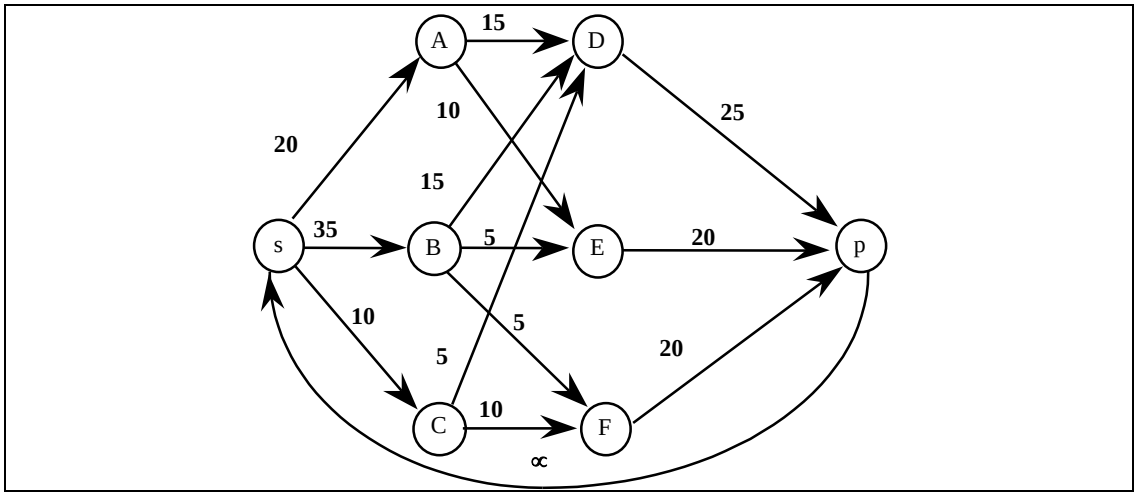


Figure 6.1 : Transport network

Definition :

A flow on a transport network G = (X, U) is a function f: U -> R that satisfies:

1)Capacity constraints: for every arc (i,j) in U: $0 \leq f_{ij} \leq c_{ij}$;

2)Conservation constraints (Kirchoff) (see figure 6.2): for every vertex i in X, we have : $\sum_{j \in U^+(i)} f_{ij} = \sum_{k \in U^-(i)} f_{ki}$

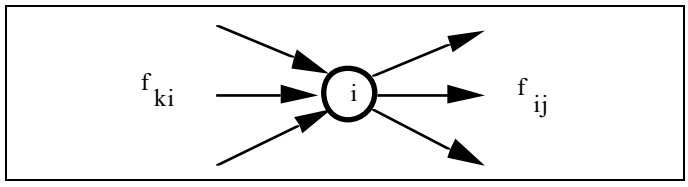


Figure 6.2 : Conservation constraint

A first example of a flow is the null flow. A second example of a flow is shown in Figure 6.3.

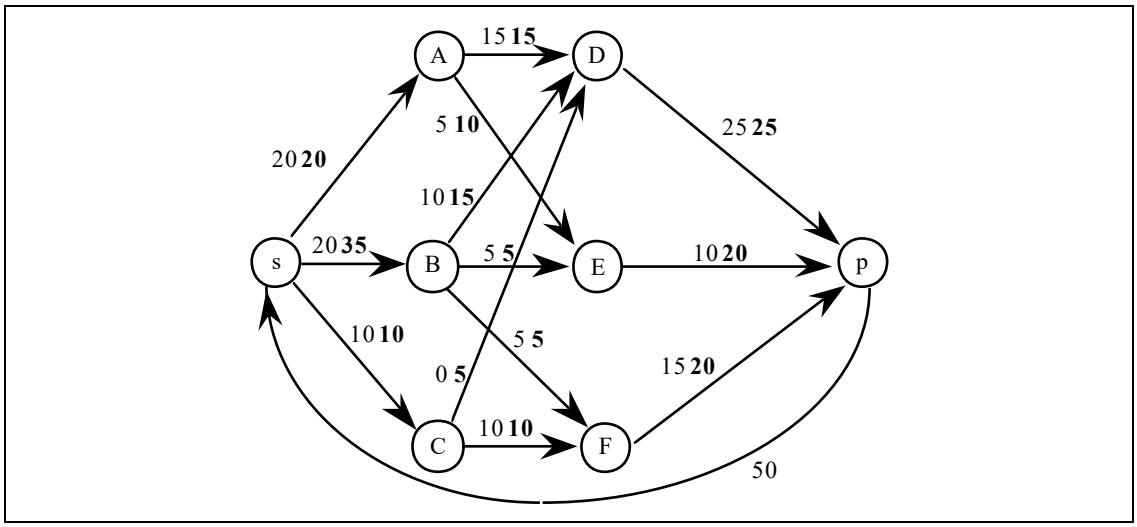


Figure 6.3 : A complete flow.

Three warehouses A, B, C, contain respectively 20, 35 et 10 tons of goods. There are demands of 25, 20 et 20 tons to destinations D, E and F. The unitary transportation costs are given in the following matrix. What would be the minimal cost transport plan ?

D	E	F
15	10	0
15	5	5
5	0	10

Problem: Determine a transport plan that allows for the maximum quantity of goods to be transported from origins to destinations. To solve this problem, the graph in Figure 6.1 is associated. More generally, the following problem must be solved:

Determine a maximum value flow on the arc (p, s), thus maximizing $f_{ps.}$

6.3. Lemma :

Before introducing the algorithms to maximize the flow, we show that the flow conservation property generalizes to a subset of vertices.

Lemma :

If Y is a subset of X, the flow outgoing from Y is equal to the flow incoming into Y.

Proof :

We sum the Kirchoff equation over all vertices in Y. The flows on the arcs that have one endpoint in Y and the other outside of Y appear on each side of the equation. Therefore, we can subtract them. What remains on one side of the equation is the sum of the flows on the arcs outgoing from Y and on the other side of the equation, the sum of the flows on the arcs incoming into Y. Thus, we have the result. Q.E.D.

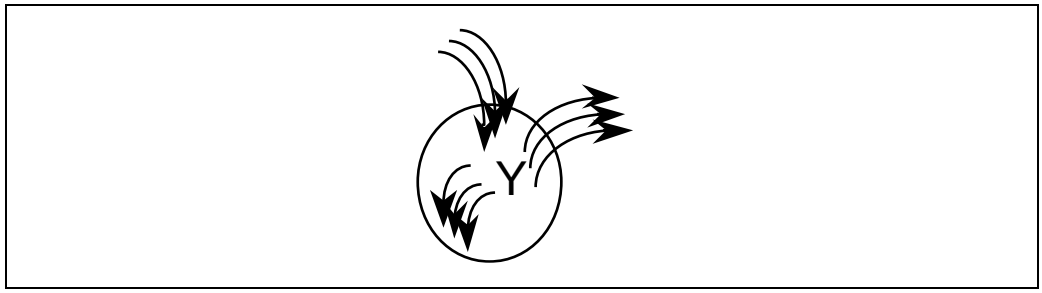


Figure 6.4: Proof of the lemma.

6.4. Complete flow :

6.4.1. Introduction :

A first idea to optimize the flow is to successively saturate the paths from s to p. We will obtain a so-called complete flow which, as we will see below, is not maximal but provides an excellent starting solution to apply the Ford-Fulkerson algorithm that we will present in the following section.

Définition :

> A flow is said to be complete if every path in the transportnetwork from s to p contains at least one saturated arc, that is an arc (i,j) such that fij = cij.

6.4.2. Algorithm for finding a complete flow:

We start with a flow f (for example, f = 0) and improve it step by step with a marking procedure:

(I) Mark s.

(II) Let i be a marked vertex not yet examined;

mark j if j is an unmarked successor of i with $f_{ij} < c_{ij}$.

The mark on j is +i.

(III) If p is marked, go to (IV).

If all marked vertices have been examined, the flow is complete END.

Otherwise, go to (II).

(IV) Improve the flow. Erase the marks (except for the one on s) and go to (I)

6.4.3. Operation on the previous example:

Successively, the following augmenting paths are found: sADp (15), sAEp (5), sBDp (10), sBEp (5), sBFp (5), and sCFp (10). The resulting flow has a value of 50 and is complete (see figure 6.3). Indeed, all paths passing through the arc sA are saturated because this arc is saturated; all paths passing through the arc sC are saturated because this arc is saturated; all paths passing through the arc sB are saturated because the arcs Dp, BE, and BF are saturated.

6.5. Ford-Fulkerson algorithm:

To construct a complete flow, we removed the augmenting paths. It is not optimal because there is still **an augmenting chain**. We explain below the notion of augmenting chain and report the Ford-Fulkerson algorithm, which allows searching for the augmenting chains and stops when there is no more augmenting chain. The flow is then optimal.

6.5.1. Augmenting chain:

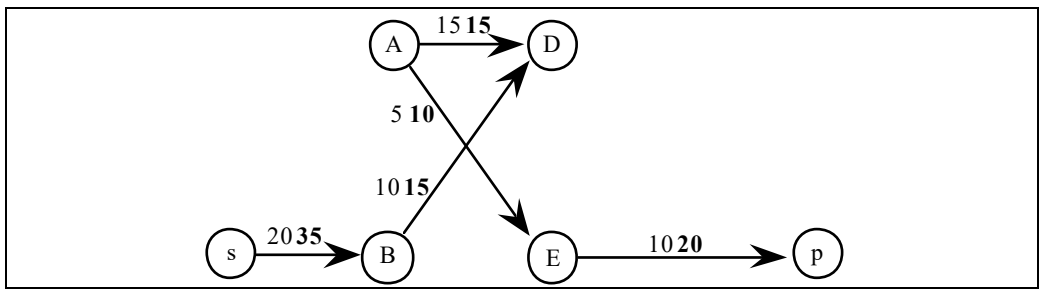


Figure 6.5: An augmenting chain

In a path from s to p, the direction of the arcs is respected. In a chain from s to p, on the other hand, the direction of the arcs is not necessarily respected. We can then distinguish between two types of arcs: arcs that respect the orientation (called forward arcs +) and arcs that do not respect the orientation (called backward arcs -). An augmenting chain is a chain going from s to p, whose forward arcs are unsaturated and whose backward arcs transport a strictly positive flow. It allows improving the flow by adding a quantity δ on the forward(+) arcs and removing a quantity δ on the backward(-)arcs.

The Ford-Fulkerson algorithm starts from an arbitrary flow (for example, the null flow or a complete flow). It searches for an augmenting chain. It improves the flow as long as there exists an augmenting chain and stops otherwise. The flow is then optimal.

6.5.2. Ford-Fulkerson algorithm:

(I) Mark the source s with *.

(II) Let i be a marked but unexamined vertex.

Consider all successors j of i:

Mark j with +i if it is unmarked and if $f_{ij} < c_{ij}$. Consider all predecessors k of i:

Mark k with -i if it is unmarked and if fki > 0.

(III) If p is marked, go to (IV).

If there are still unexamined marked vertices, go to (II).

Otherwise, the flow is optimal. END.

(IV) Improve the flow using the augmenting chain that allowed marking p.

Clear the marks, except for s, and go back to step (I).

6.5.3. Example :

(I) We mark s with * and start from the complete flow built previously (see figure 6.3).

(II) We examine s.

We mark B with +s, because only the arc sB is unsaturated.

(II) We examine B. BD is the only unsaturated arc leaving B, so we mark D with +B (successor of B).

B has no unmarked predecessor.

(II) We examine D.

There is no unsaturated arc leaving D.

However, the arc AD carries a non-zero flow. This means that A is marked -D (predecessor of D).

(II) We examine A.

We mark E with +A, as it is an unsaturated successor of A.

The only predecessor of A is already marked.

(II) We examine E.

We mark p with +E, i.e., as a successor of E.

(IV) Since p is marked, we have found an augmenting path. We use the marks "going back" to reconstruct this augmenting path: it is the path sBDAEp, whose four arcs in the + direction are unsaturated, while the arc in the - direction carries a non-zero flow.

This path allows us to improve the flow by 5 units. We clear the marks except for s and start again.

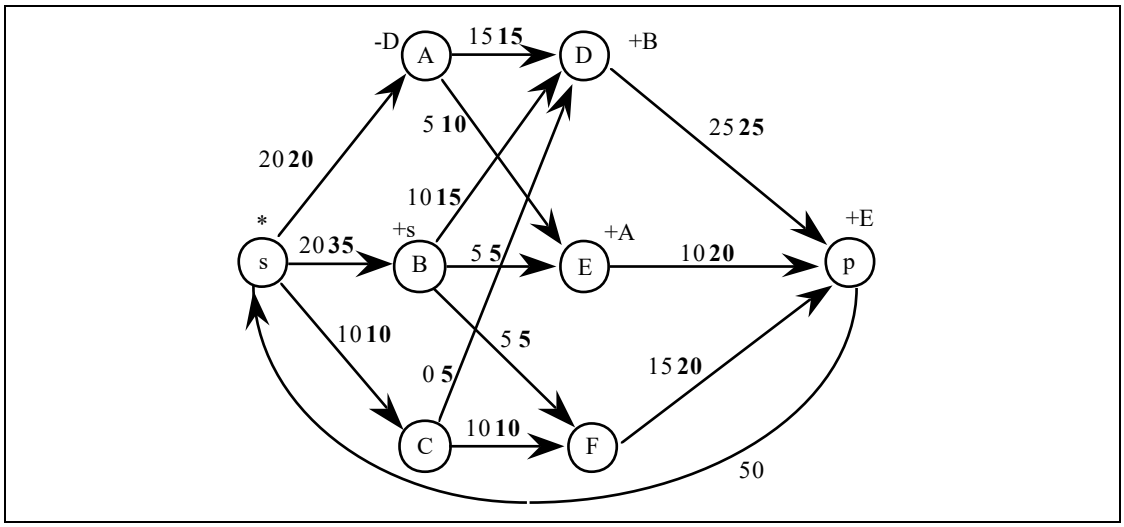


Figure 6.6 : Ford-Fulkerson marking

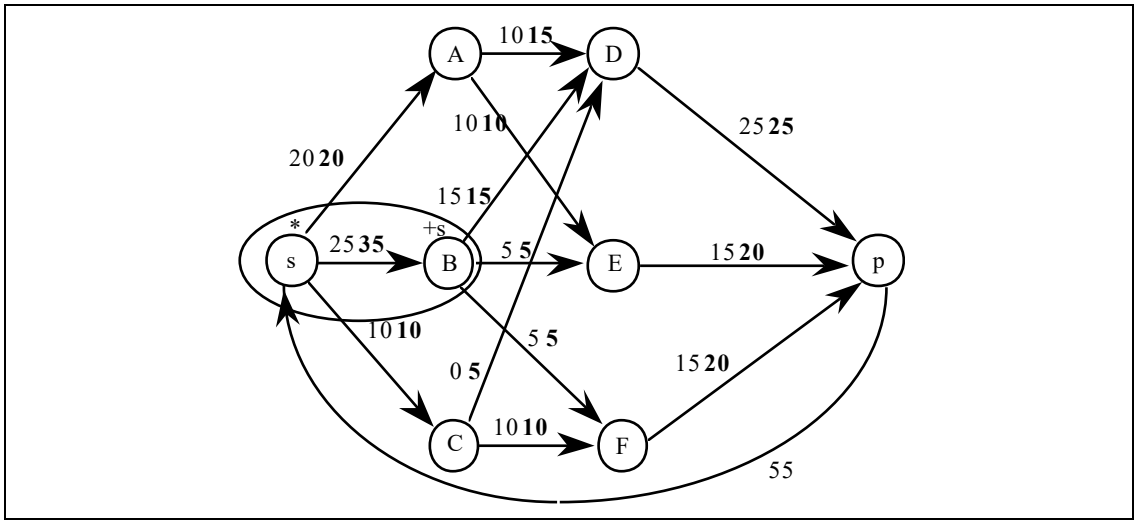


Figure 6.7 : Optimal flow.

(iii) This time, we mark the vertex B and it's finished because all the outgoing arcs from B are now saturated. The flow is maximal as we will show below in the general case.

6.5.4. Ford-Fulkerson's Theorem:

Definition :

> A cut is a set of vertices containing s and not containing p. The capacity of a cut is the sum of the capacities of the arcs leaving this cut.

Example : at the end of the previous algorithm, the set of marked vertices M = {s, B} is a cut that has the following property:

-all arcs leaving M are saturated;

- the arcs entering M, except for (p,s), have zero flow.

Ford-Fulkerson Theorem:

The minimum capacity of a cut is equal to the maximum flow.

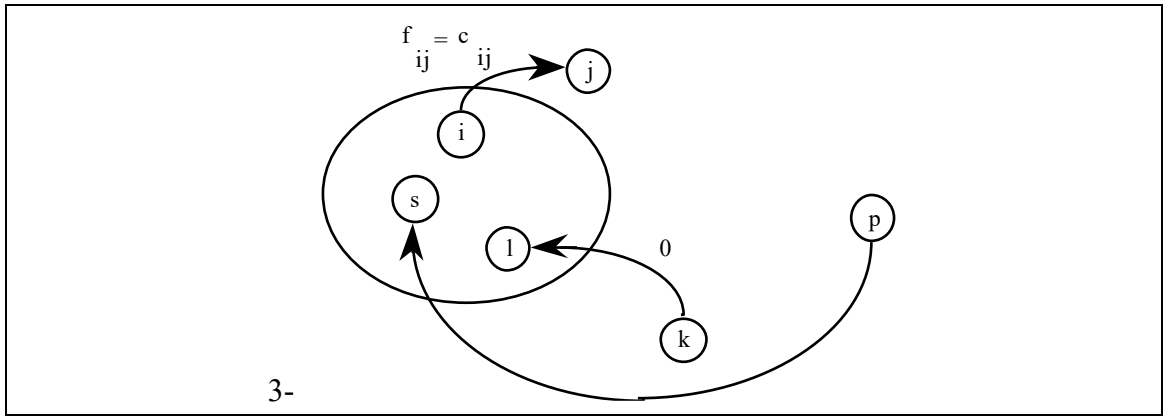
Proof :

It must be shown that: Max f_{ps} = Min C(Y) (with f, flow,Y $\subseteq$, s$\in$Y, p$\notin$Y)

1- Let's show that Max $f_{ps} \leq$ Min C(Y):

Let f be any flow and Y any cut, it suffices to show that: $f_{ps} \leq C(Y)$. The flow leaving Y is less than or equal to C(Y). The flow entering Y is greater than or equal to fps. Since the flow entering Y is equal to the flow leaving Y, we have: $f_{ps} \leq C(Y)$.

2- Let's show that Max $f_{ps} \geq$ Min C(Y): We will show that there exists a flow f ' and a cut M such that: $f'_{ps} = C(M)$. The inequality will result because Max $f_{ps} \geq f'_{ps}$ and Min C(Y) $\leq$ C(M). When we apply Ford-Fulkerson, at the end, if we designate by M the set of marked vertices and by f ' the flow, we have $C(M) = f'_{ps}$.



4- Figure 6.8 : Minimal cut.

After a finite number of iterations, Ford-Fulkerson algorithm stops. Indeed, each iteration improves the flow by at least one unit; this flow is bounded, for example, by C({s}). In the end, the outgoing arcs from M are saturated and the incoming arcs into M have zero flow. The flow entering M is f_{ps}. The flow leaving M is C(M). Hence, $C(M) = f_{ps}$. Q.E.D.

The following corollaries are immediately demonstrated:

1: A necessary and sufficient condition for a flow to be maximal is that there exists no augmenting path.

2: The Ford-Fulkerson algorithm constructs a maximal flow.

6.5.5. Complexity of the Ford-Fulkerson algorithm:

Figure 6.9 shows an example where the number of iterations of Ford-Fulkerson is equal to 2M if we successively choose the augmenting paths {s x y p} and {s y x p} of capacities 1. The number of augmentations can thus be exponential. The Ford-Fulkerson algorithm is not polynomial.

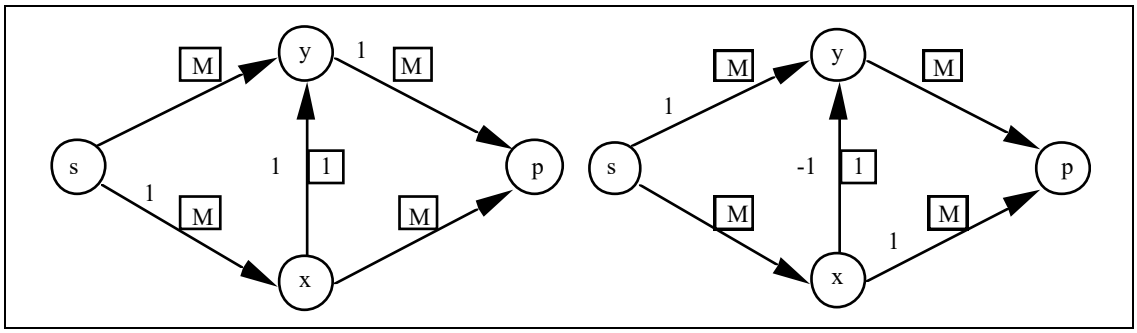


Figure 6.9 : Example.

However, if we apply the "first labeled, first examined" rule, the Ford-Fulkerson algorithm becomes polynomial with complexity $O(n^5)$: the number of augmenting paths is bounded by 1/4 (n^3 - n) according to the Edmonds-Karp theorem. This amounts to searching for the shortest augmenting paths. Finally, note that there are better algorithms, in particular, the KARZANOV algorithm with a global complexity of $O(n^3)$.

6.6. Maximum flow at minimum cost:

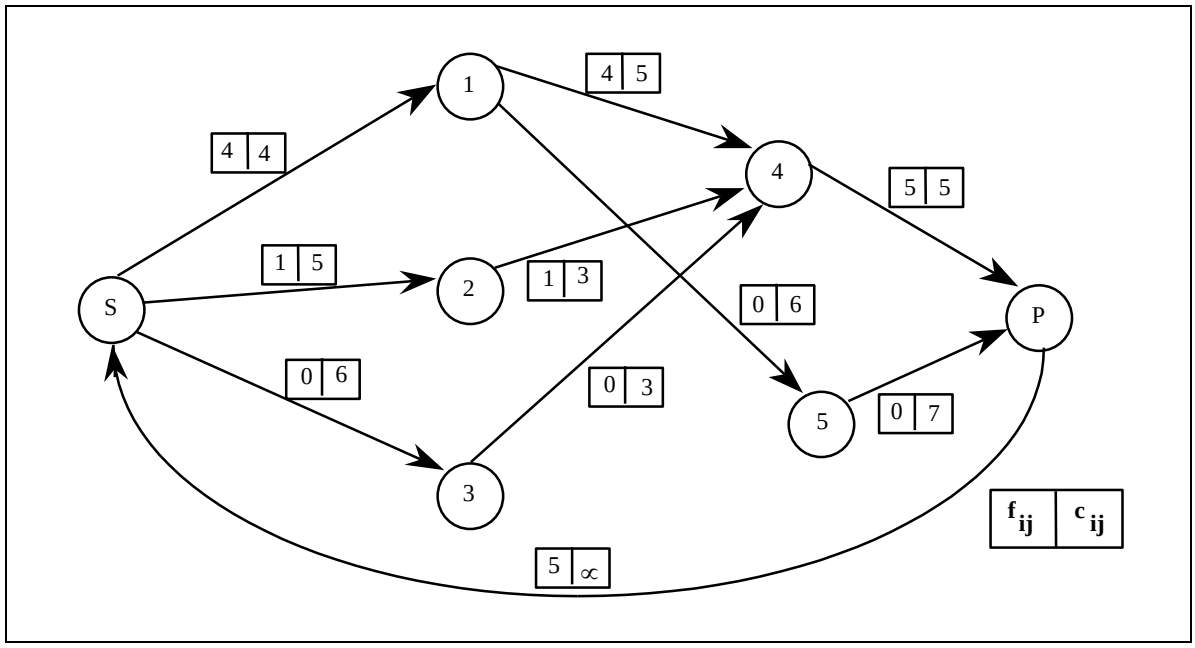


Figure 6.10 : Flow on a transport network

We return to the Ford-Fulkerson algorithm by showing that the concept of an augmenting chain on the initial graph is equivalent to the concept of an augmenting path on the residual graph. In

Figure 6.10, we have a flow on the transport network, and the corresponding residual graph is shown in Figure 6.11.

6.6.1. Maximum flow problem:

Definition of the residual graph $G^e(f)$:

For a pair consisting of a flow f and a transport network G, we associate the residual graph $G^e(f) = (X, U^e(f))$, defined by:

> for any arc (i, j) in G, there are two arcs in $G^e(f)$: one arc (i, j) of capacity $c_{ij} - f_{ij}$ and one arc (j, i) of capacity f_{ij} (if the capacity of an arc is zero in $G^e(f)$, the arc is not represented, as it is useless).

Note that adding (or subtracting) flow on an arc of the initial graph is equivalent to adding this flow on the corresponding (or inverse) arc of the residual graph. Augmenting chains on the initial graph correspond to paths on the residual graph. Therefore, the Ford-Fulkerson algorithm becomes the following algorithm :

Algorithm of the residual graph:

This algorithm constructs a flow of maximum value on the backward arc (p, s) in G.

(I) Initialize f on all arcs (for example, f = 0 if b(u) = 0 for all u).

(II) (II) Construct $G^e(f)$ and find a path from s to p in $G^e(f)$. Go to (III) if a path is found, otherwise END.

(III) (III) Improve the flow f along the obtained path and return to (II).

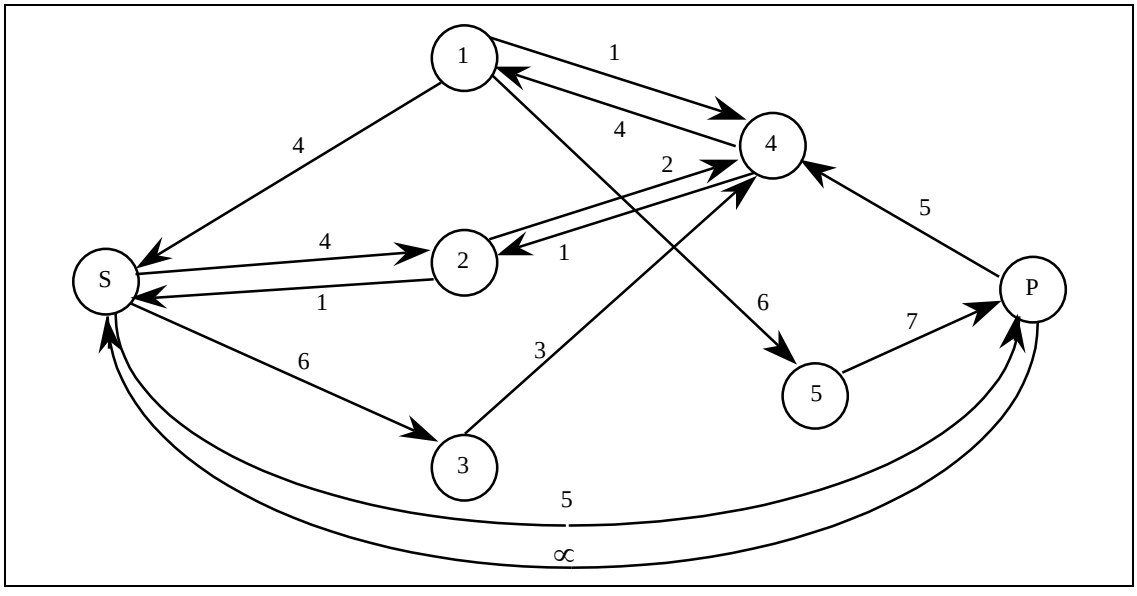


Figure 6.11 : First residual graph

Application of the algorithm to the previous example

On the first residual graph (Figure 6.11), we have the path [s, 3, 4, 1, 5, p] that allows us to improve the flow by 3. On the second residual graph (Figure 6.12), we have the path [s, 2, 4, 1, 5, p] that allows us to improve the flow by 1. The third residual graph (Figure 6.13) no longer contains a path from s to p; the flow is therefore maximum. The flows are equal to the capacities of the reverse arcs in this third residual graph (see Figure 6.14).

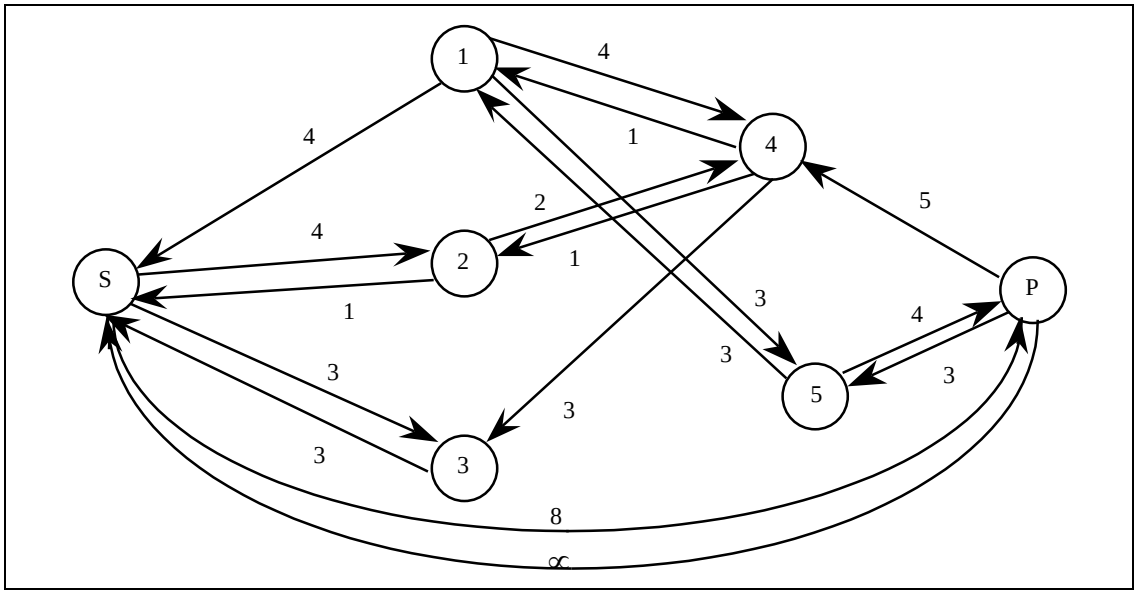


Figure 6.12 :Second residual graph

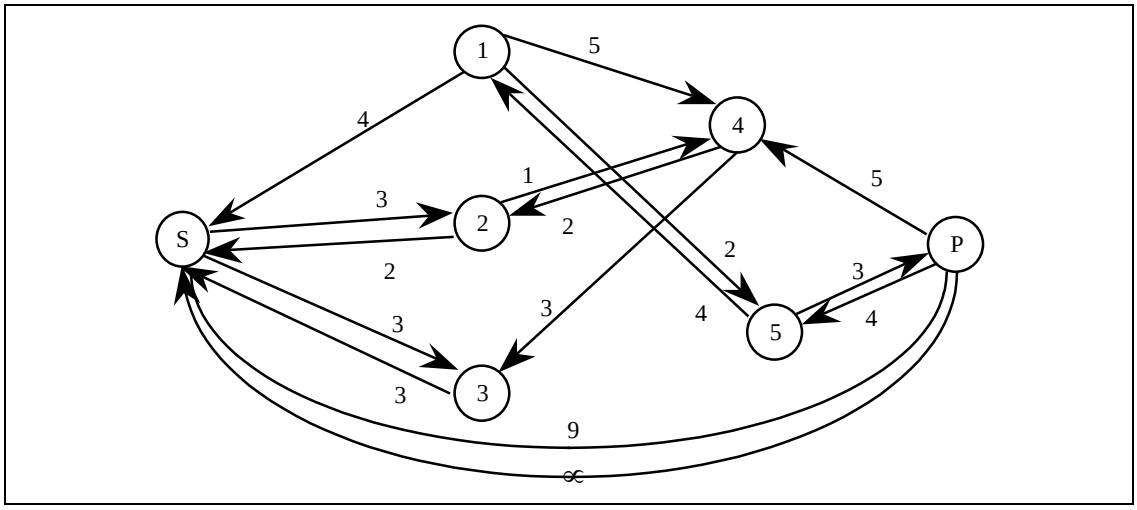


Figure 6.13: Third residual graph

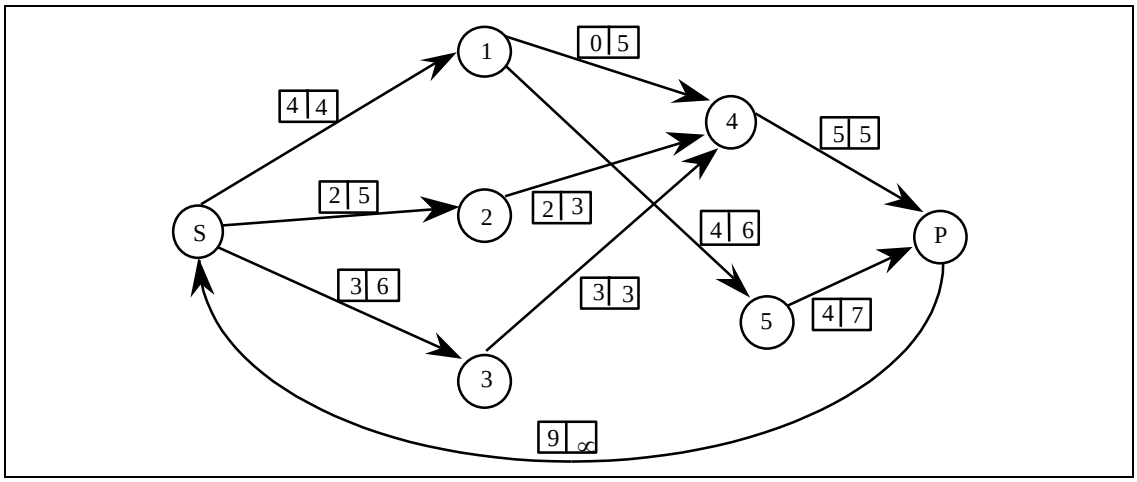


Figure 6.14 : Maximal flow.

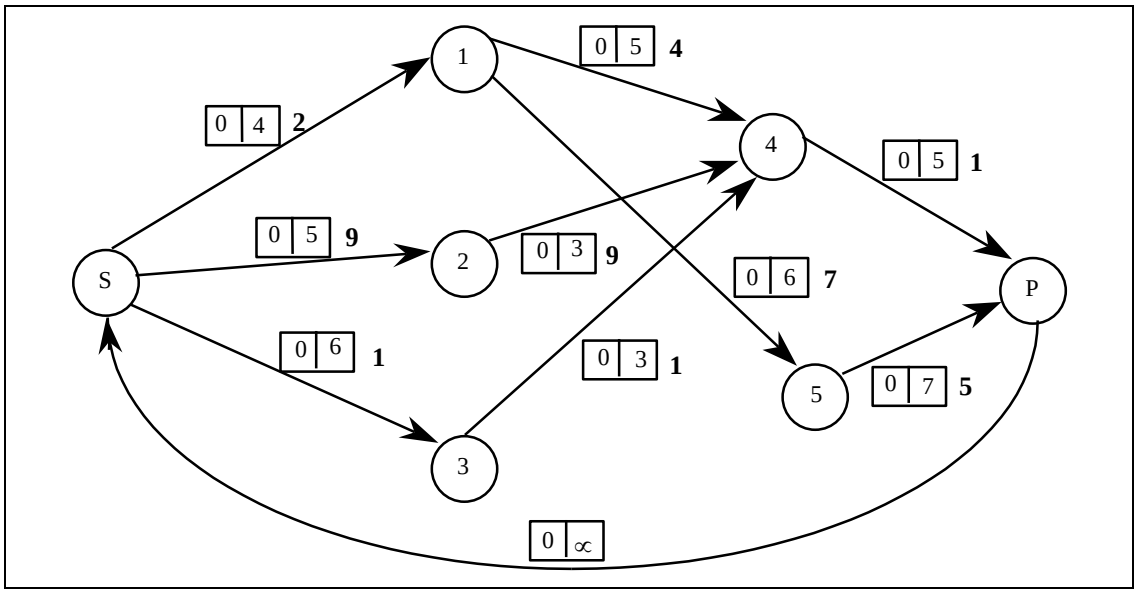


Figure 6.15: A problem with costs

6.6.2. The problem of maximum flow at minimum cost:

We will now modify this algorithm for the residual graph to take into account unit transport costs on the arcs. For each arc (i, j), we also have a cost d_{ij} (≥ 0). We must determine a maximum flow of minimal global cost, that is, which minimizes: $\sum_{(i,j)\in U} d_{ij} f_{ij}$

ROY's Algorithm :

(I) Set f = 0.
(II) Construct $G^e(f)$ by introducing a cost d_{ij} on arc (i, j) and a cost - d_{ij} on arc (j, i).
(III) (Look for a minimum cost path in $G^e(f)$ from s to t. If there is no path from s to t, then the flow is maximal of minimum cost, so FINISH.
(IV) Otherwise, improve the flow f and return to (II).

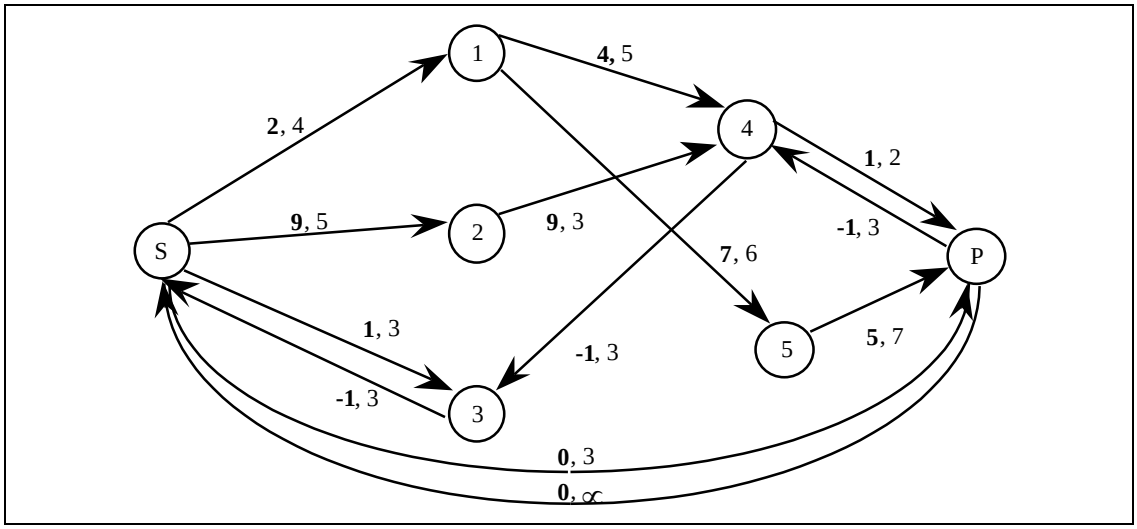


Figure 6.16 : Second residual graph.

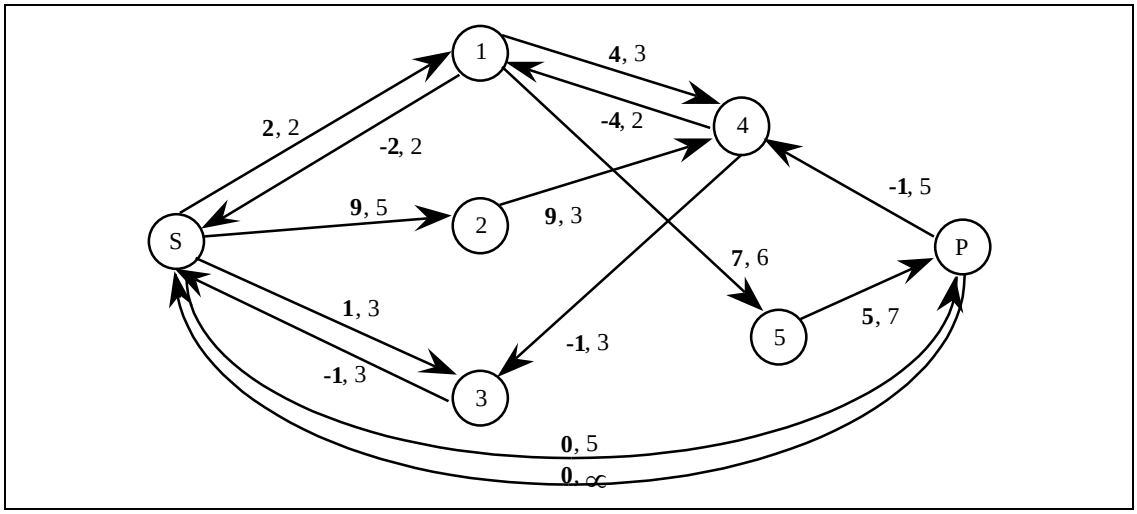


Figure 6.17 : Third residual graph.

Example :

The first residual graph coincides with the initial graph (see Figure 6.15). On this graph, the minimum cost path from s to t is [s, 3, 4, t] with a cost of 3. We then pass a flow of value 3 along this path and obtain a second residual graph (see Figure 6.16). On this graph, the minimum cost path from s to t is [s, 1, 4, t] with a cost of 7. We then pass a flow of value 2 along this path and obtain a third residual graph (see Figure 6.17). On this graph, the minimum cost path from s to t is [s, 1, 5, t] with a cost of 14. We then pass a flow of value 2 along this path and obtain a fourth residual graph (see Figure 6.18). On this graph, the minimum cost path from s to t is [s, 2, 4, 1, 5, t] with a cost of 26. We then pass a flow of value 2 along this path and obtain a final residual graph (see Figure 6.19). There is no longer a path from s to t.

The flow is maximum of minimum cost (see Figure 6.20). The cost of this flow is: 3x3 + 2x7 + 2x14 + 2x26 = 103.

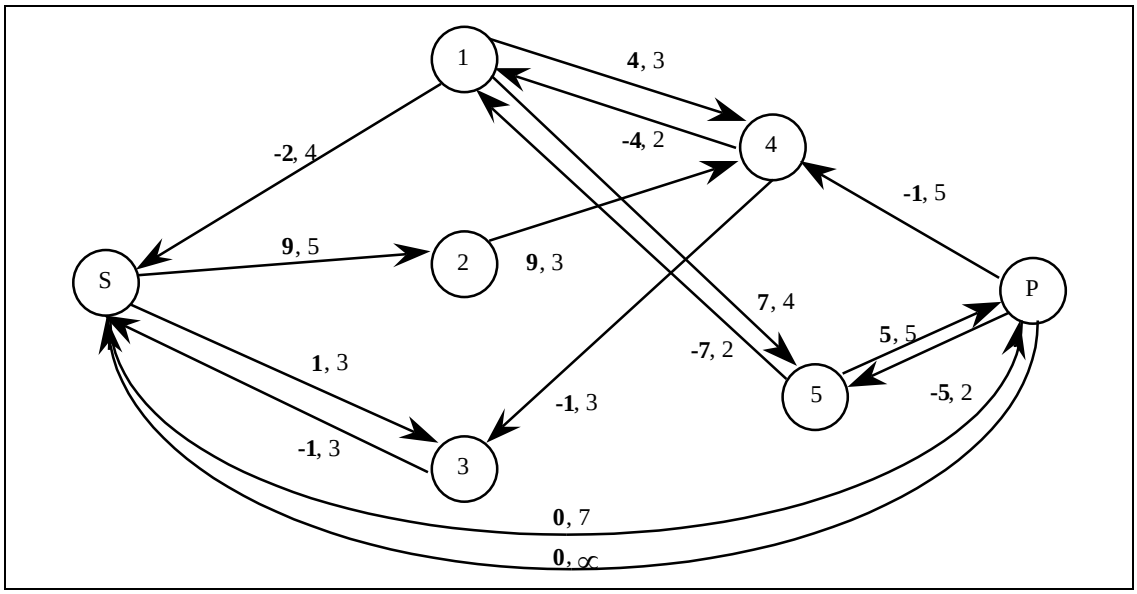


Figure 6.18 : Fourth residual graph.

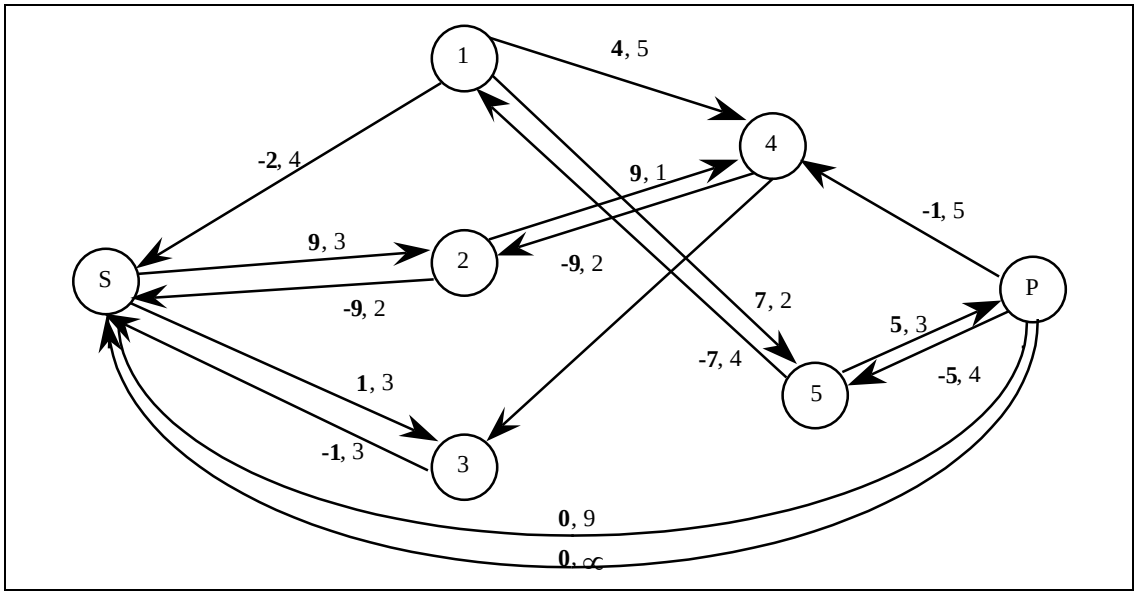


Figure 6.19 : Last residual graph.

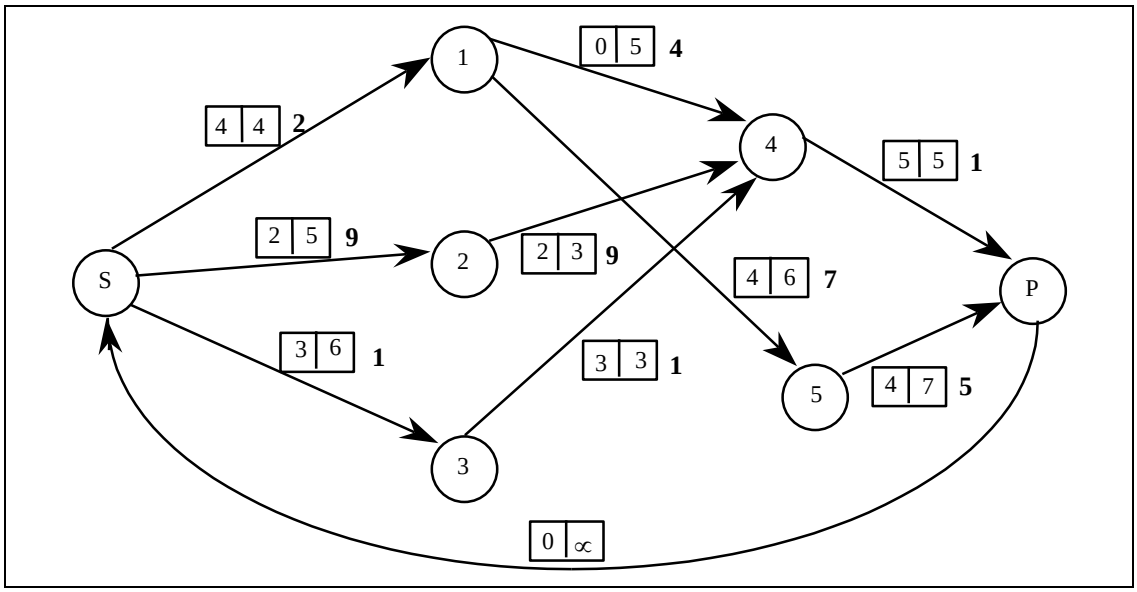


Figure 6.20 : Maximum flow of minimum cost.

7. THE PROBLEMS OF CHANNELED FLOWS AT MINIMUM COST:

7.1. Introduction :

In this chapter, we consider generalizations of the maximum flow problem by addressing minimum cost flow problems and channelized flow problems. A channelized flow is a flow that must satisfy, in addition to the Kirchhoff and capacity constraints, additional constraints called bounds: for each arc (i, j), a bound bij is associated, and the flow on arc (i, j) must be greater than or equal to bij. A flow is considered to be of minimum cost if a linear function of transportation costs is minimized. The basic tool remains the augmenting chain. We simplify it by introducing the concept of residual graph, which allows us to reduce the search for augmenting paths. We explain how to find a channelized flow and a minimum cost flow.

7.2. Channelized flow and residual graph:

Sometimes, an integer lower bound b(u) on the flow of arc u may not be zero, in which case the problem of finding a compatible flow (referred to as channelized flow) arises. It is reminded that c(u), the capacity of arc u, is an integer upper bound on the flow of arc u.

Definition:

A channelized flow is a function f from U to N (set of integers) that satisfies the Kirchhoff constraints, as well as the constraints of bounds and capacities. We seek a flow f such that for every u, $b(u) \leq f(u) \leq c(u)$. The Hoffman theorem provides a condition for the existence of a channelized flow.

Hoffman's Theorem:

A necessary and sufficient condition for the existence of a channelized flow in the network G = (X, U, b, c) is that for every $Y \subseteq X$, the sum of the bounds of the arcs entering Y is less than or equal to the sum of the capacities of the arcs leaving Y, i.e.:

$$\sum_{u \in U^-(Y)} b(u) \leq \sum_{u \in U^+(Y)} c(u)$$

Proof:

The condition is obviously necessary. Indeed, the flow entering Y is equal to the flow leaving Y. The incoming flow is greater than the sum of the bounds, and the outgoing flow is less than the sum of the capacities, so the inequality holds whenever there exists a flow.

The sufficiency of the condition is ensured by the algorithm for finding a compatible flow that we introduce and justify below. Q.E.D.

Algorithm for finding a compatible flow:

{Note: fu denotes the flow on arc u}

(I) Start with a flow f set to zero. (II) Search for an arc u such that fu < b(u) - If such an arc does not exist, then END 1; - Otherwise, set s as the terminal node of u and p as the initial node of u.

(III) Search for a path from s to p in the forward direction (+) where the arcs are unsaturated, and in the backward direction (-) where the flow on the arcs is strictly greater than the bound; if such a path does not exist, then END 2.

(IV) Use this path to improve the flow by attempting to satisfy the bound constraint on arc u, and return to (II).

This algorithm terminates either after finding a compatible flow (FIN1), or because the necessary condition of the theorem is not satisfied (FIN2). We will prove this result after running the algorithm on the example in Figure 7.2.

We find the following four augmenting paths successively:

• for s = x3 and p = x1, we have the path (x3, x4, x1) with a capacity of 2.

• for s = x3 and p = x1, we have the path (x3, x4, x5, x1) with a capacity of 2.

• for s = x2 and p = x1, we have the path (x2, x5, x1) with a capacity of 3.

• for s = x3 and p = x5, we have the path (x3, x4, x5) with a capacity of 1.

We then have a compatible flow.

Validity of the algorithm:

Validité de l'algorithme :

The algorithm terminates. It cannot loop indefinitely because the set of values is finite, and we are sure to never encounter the same flow between two steps. Indeed, $\sum_{u \in U} \max\{b_u - f_u, 0\}$ is strictly decreasing during the algorithm and bounded below by 0.

Since we start with a null flow and always respect capacity constraints, we can be sure that for every u: $f\mathrm{u} \leq \mathrm{c(u)}$.

If FIN1, then we have obtained a compatible flow.

If FIN2, then the marking algorithm for a path from s to p is blocked. Let's denote A as the set of vertices marked at the end of this phase. We verify that the flow entering A is strictly less than the sum of the capacities of the incoming arcs to A, and is equal to the sum of the capacities of the outgoing arcs from A (Refer to 7.1).

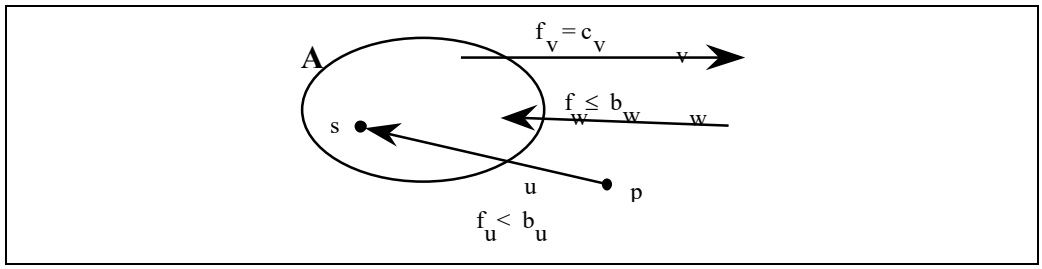


Figure 7.1 : Case of non-existence of a compatible flow (FIN2).

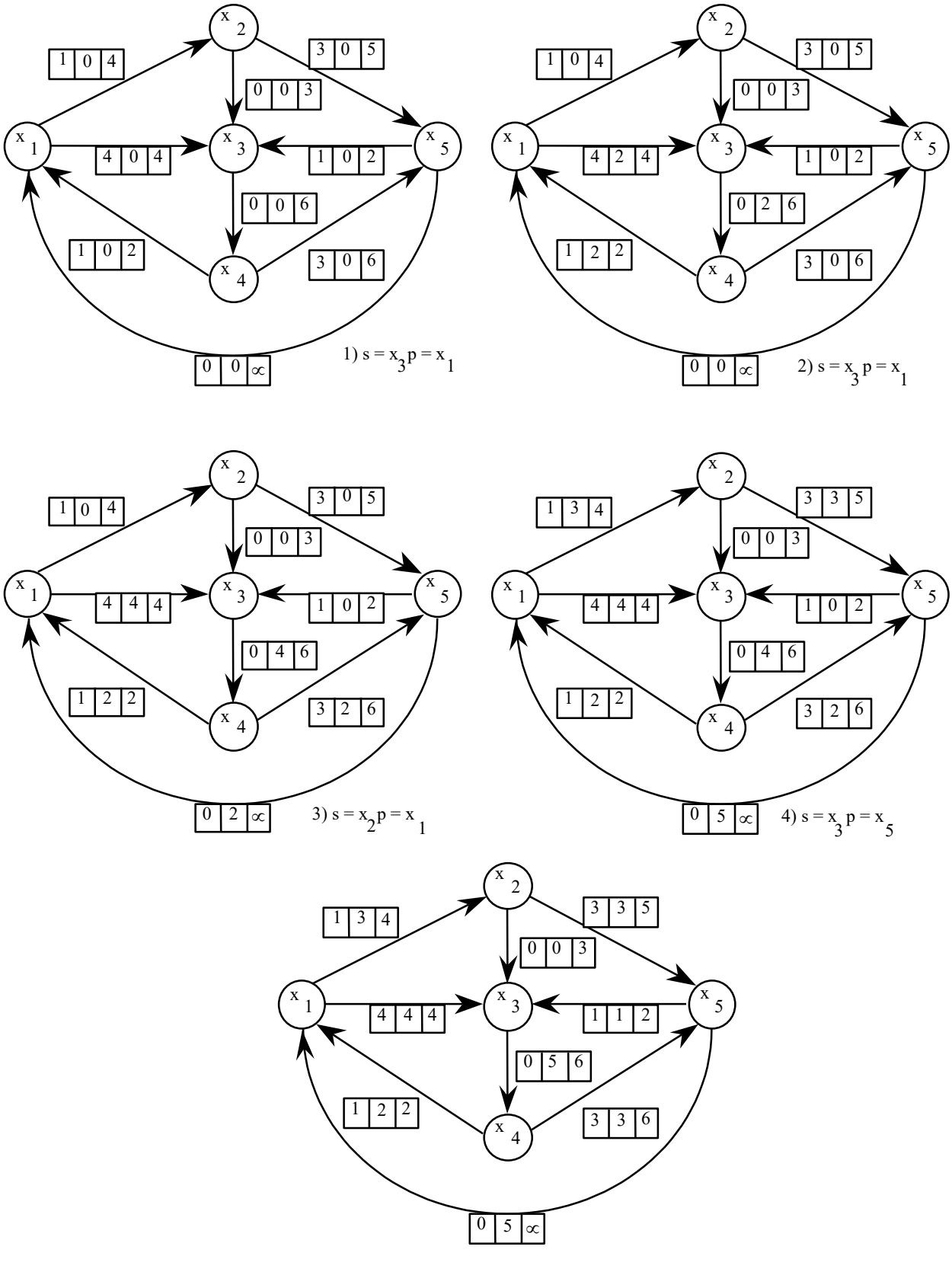


Figure 7.2 : application of algorithm

7.3. Minimum cost flow :

7.3.1. Decomposition of a flow on a basis of circuits :

Proposal:

A necessary and sufficient condition for a flow f to be positive is that there exists a family of circuits $\mu_1, \mu_2, ..., \mu p$ and positive integer coefficients $\lambda_1, \lambda_2, ..., \lambda p$ such that: $f = \lambda_1\mu_1 + \lambda_2\mu_2 + ... + \lambda p\mu p$.

Proof:

The condition is sufficient because each circuit can be considered as a positive flow. Let's show that the condition is also necessary. Starting from an arc with non-zero flow, we can find another arc with non-zero flow due to Kirchoff's condition. By iterating this process, we will eventually revisit a same vertex, and thus we have found a circuit μ_1. Let λ_1 be the minimum value of flows on this circuit; $f - \lambda_1\mu_1$ is still a positive flow, and we can apply the same reasoning to it to obtain all the components of f successively, as a new arc with zero flow appears each time. Q.E.D.

7.3.2. Gap bijection :

Definition of the residual graph Ge(f) (including bijections):

To the pair consisting of a flow f and a transportation network G, we associate the residual graph Ge(f) = (X, Ue(f)), defined as follows:

> **for each arc (i, j) in G, there are two corresponding arcs in Ge(f): an arc (i, j) with capacity cij - $f$ij, and an arc (j, i) with capacity $f$ij - bij (if the capacity of an arc is zero in Ge(f), it is not represented as it is unnecessary)**.

Let $f0$ be a compatible flow and Ge($f0$) be the associated residual graph. Let φ be a flow on Ge($f0$). We define a new compatible flow f on G using the notation $f = f0 \oplus \varphi$, where:

$$((f)_u = (f_0)u + (\varphi)_{u+} - (\varphi)u-$$

The cost of the flow f on G is equal to the cost of the flow $f0$ on G plus the cost of the flow φ on the residual graph. We aim to minimize the cost of f, where f is a compatible flow. The method we will discuss involves removing circuits with strictly negative costs in the residual graph, and is based on the optimality theorem, which we will prove.

7.3.3. Optimality theorem :

Optimality theorem:

A compatible flow $f0$ is of minimal cost if and only if Ge($f0$) does not contain any circuit with strictly negative cost.

Proof:

The condition is obviously necessary. If there exists a circuit with strictly negative cost, this circuit allows to construct a strictly better flow, as explained in the example below. The condition is also sufficient

Let $f0$ be a flow such that $G^e(f_0)$ does not contain any circuit with strictly negative cost, and let f be another compatible flow. We have $f = f_0 \oplus \varphi$, where φ decomposes over a base of circuits of $G^e(f_0)$: $\varphi = \lambda_1\mu_1 + \lambda_2\mu_2 + ... + \lambda_p\mu_p$.

Therefore, $Cost(\varphi) = \sum_{i\in[1,p]} \lambda_i \ Cost(\mu_i)$ is positive because the costs of all the circuits in the graph of the residual network are positive, and the λi coefficients are also positive. By using the equation $Cost(f) = Cost(f_0) + Cost(\varphi)$ on a $Cost(f) \geq Cost(f_0)$ which means that $f0$ is a compatible flow of minimal cost. Q.E.D.

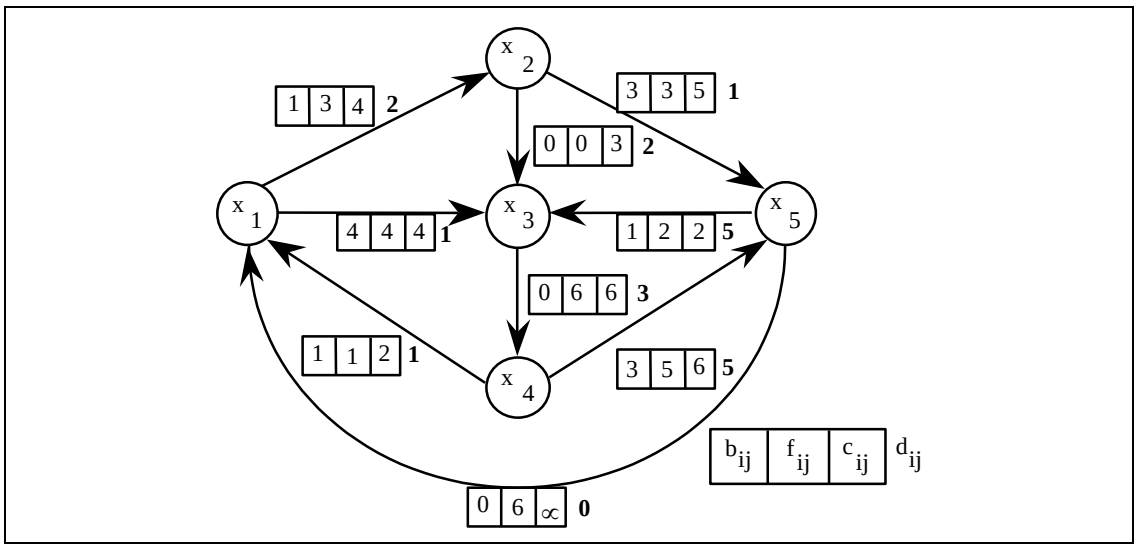
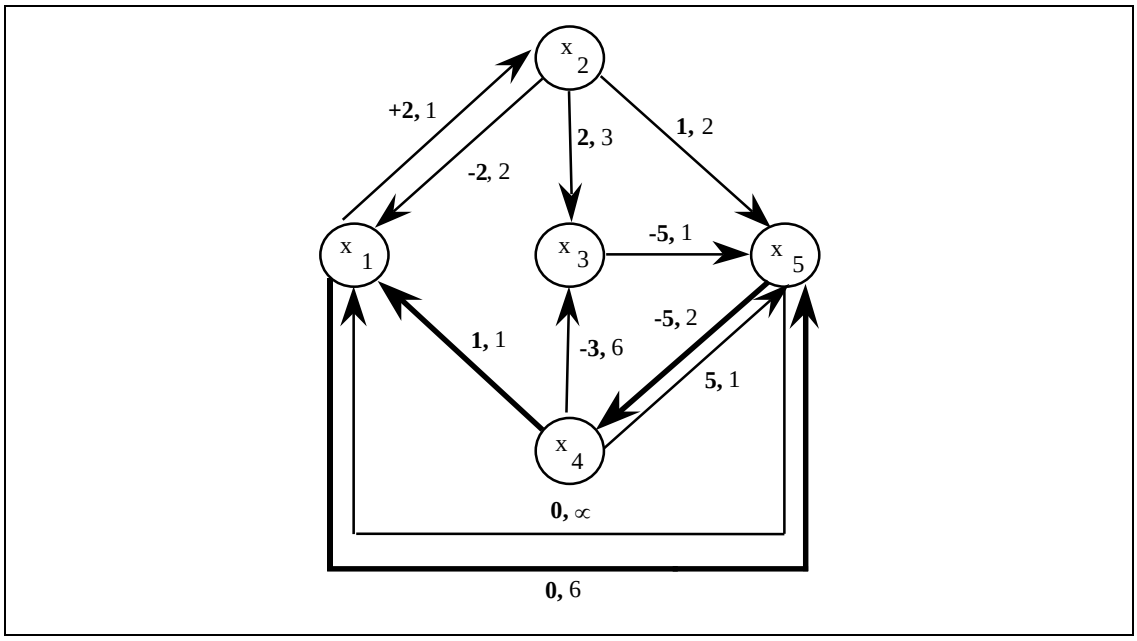
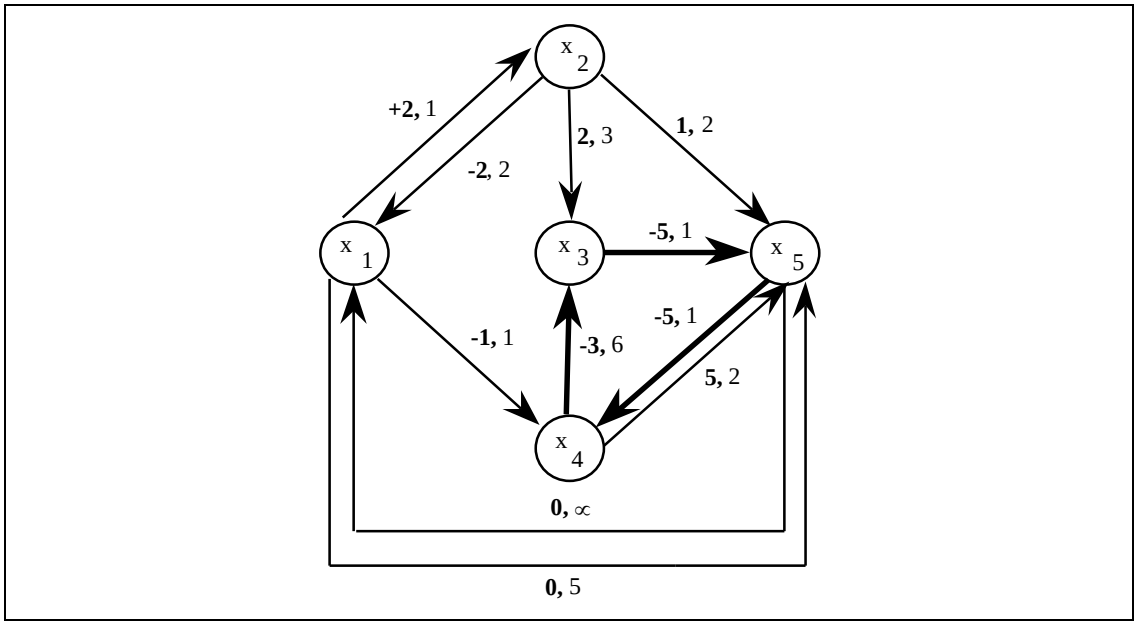


Figure 7.3 : Minimum-cost network flow problem

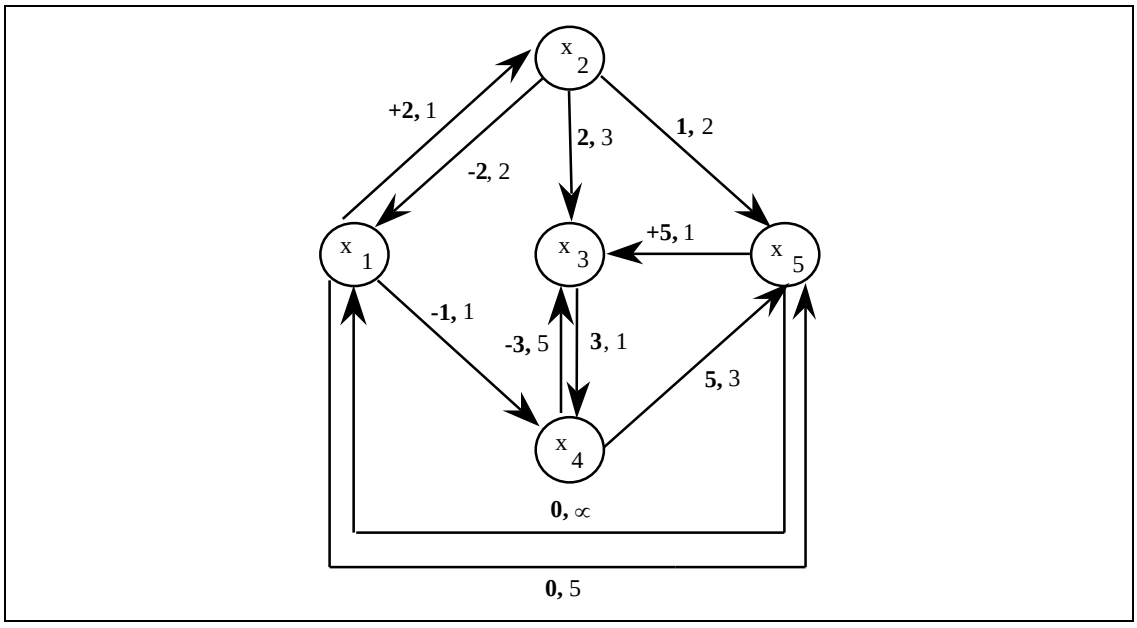
In Figure 7.3, we have an example of a compatible flow. We will use the previous result to find a minimum-cost compatible flow. In Figure 7.4, we have the associated residual graph. This residual graph contains a circuit with strictly negative cost: the circuit [x5, x4, x1]. We then send a flow of 1 through this circuit in the residual graph. In Figure 7.5, we have the updated residual graph. This residual graph contains a circuit with strictly negative cost: the circuit [x3, x5, x4]. We then send a flow of 1 through this circuit in the residual graph. We obtain a final residual graph where all circuits have positive or zero costs (see Figure 7.6). As a result, the resulting compatible flow, which is shown in Figure 7.7, is a minimum-cost compatible flow.



7.4 : first residual graph.



7.5 : second residual graph.



7.6 : final residual graph.

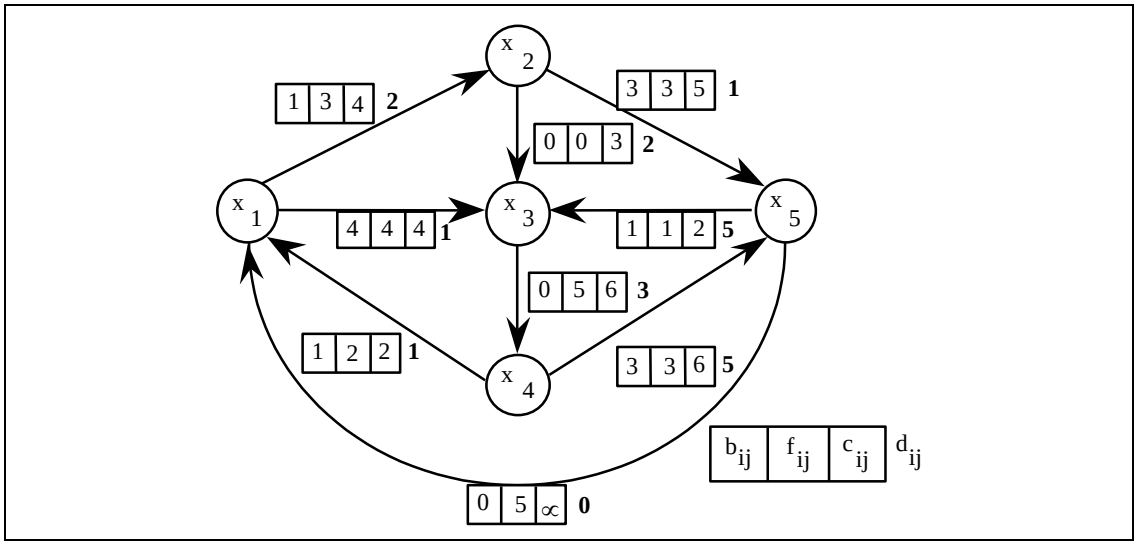


Figure 7.7 : minimum-cost network flow.

7.3.4. Proof of Roy's algorithm:

We conclude this chapter by using the optimality theorem to prove the validity of Roy's algorithm.

Proposition: Roy's algorithm determines a maximum flow with minimum cost.

Proof:

We show by induction that all the flows fk constructed during Roy's algorithm have minimal cost among flows with value vk. This will imply that the last flow obtained is a maximum flow with minimum cost.

The property holds for $f0 = 0$, which is the flow with zero value and minimal cost. Let's assume that fk has minimal cost among flows with value vk. We have $f_{k+1} = f_k \oplus \varepsilon\mu$, where μ is a circuit with minimal cost passing through the back edge in the residual graph $G^e(f_k)$, and ε is the maximum allowable value of the flow. Let f be an arbitrary flow with the same value on the back

edge as fk+1. We know that $f = f_k \oplus \varphi$, where φ is a flow on the residual graph $G^e(f_k)$. Applying the decomposition theorem, we have $\varphi = \lambda_1\mu_1 + \lambda_2\mu_2 + ... + \lambda_p\mu_p$ By using the fact that f_{k+1} and f have the same components on the back edge, we obtain:

$\sum_{i \in I} \lambda_i = \varepsilon$ where I is the set of circuits passing through the back edge.

We deduce that :

$\text{Cost}(f) = \sum_{i \in [1, p]} \lambda_i \text{Cost}(\mu_i) + \text{Cost}(f_k) \geq \sum_{i \in I} \lambda_i \text{Cost}(\mu_i) + \text{Cost}(f_k)$ since all circuits that do not pass through the back edge have a non-negative or zero cost due to the recurrence assumption, fk is of minimal cost among the flows of value v_k. Using $\text{Cost}(\mu_i) \geq \text{Cost}(\mu)$ for $i \in I$ and

$\sum_{i \in I} \lambda_i = \varepsilon$, we have : $\text{Cost}(f) \geq \varepsilon \text{Cost}(\mu) + \text{Cost}(f_k) = \text{Cost}(f_{k+1})$. This demonstrates the minimality of the cost of fk+1 and the result by induction. Hence, the validity of ROY's algorithm. Q.E.D.

7.4. Graph Algorithms and Linear Programming

The minimum cost flow problem can be modeled by the following linear program. The same applies to problems of pathfinding and matching. For example, finding a minimum-cost path from s to t is equivalent to finding a flow of value 1 from s to t with minimum cost. We have seen for the assignment problem, which is a matching problem, how to reduce it to a flow problem. Therefore, in principle, linear programming solvers could be used for all these problems, especially since the solution of the continuous linear program is integral. However, specific algorithms for each problem are much more efficient in practice, which justifies the study of their specific properties..

$$Min \quad \sum_{(i,j) \in U} d_{i,j} f_{i,j}$$

$$(a) \quad \sum_{j \ succ \ i} f_{i,j} = \sum_{k \ prec \ i} f_{k,i}; \quad \forall i \in V$$

$$(b) \quad 0 \leq f_{i,j} \leq C_{i,j}; \quad \forall (i, j) \in U$$