# Foundation Of Operations Research
## *Theory*

Christian Rossi

Academic Year 2023-2024

**Abstract**

Operations Research is the branch of applied mathematics dealing with quantitative methods to analyze and solve complex real-world decision-making problems.

The course covers some fundamental concepts and methods of Operations Research pertaining to graph optimization, linear programming and integer linear programming.

The emphasis is on optimization models and efficient algorithms with a wide range of important applications in engineering and management.

# Contents

# Chapter 1

# Introduction

## 1.1 Definition

**Definition**

*Operations Research* is the branch of mathematics in which mathematical models and quantitative methods are used to analyze complex decision-making problems and find near-optimal solutions.

It is an interdisciplinary field at the interface of applied mathematics, computer science, economics and industrial engineering.

## 1.2 Decision-making problems

**Definition**

The *decision-making problems* are problems in which we must choose a feasible solution among many alternatives based on one or several criteria.

The more complex decision-making problems are tackled via a mathematical modelling approach. Those problems can be classified in the following categories:

1. Assignment problem: given $m$ jobs and $m$ machines, suppose that each job can be executed by any machine and that $t_{ij}$ is the execution time of job $J_i$ one machine $M_j$. We want to decide which job assign to each machine to minimize the total execution time. Each job must be assigned to exactly one machine, and each machine to exactly one job. The number of feasible solution is equal to $m!$.

2. Network design: we want to decide how to connect $n$ cities via a collection of possible links to minimize the total link cost. Given a graph

$G = (N, E)$ with a node $i \in N$ for each city and an edge $\{i, j\} \in E$ of cost $c_i j$, select a subset of edges of minimum total cost, guaranteeing that all pairs of nodes are connected. The number of feasible solution is equal to $2^{|E|}$.

3. Shortest path: given a direct graph that represents a road network with distances (traveling times) for each arc, determine the shortest path between two points (nodes).

4. Personnel scheduling: determine the week schedule for the hospital personnel, to minimize the number of people involved while meeting the daily requirements.

5. Service management: determine how many desks to open at a given time of the day so that the average customer waiting time does not exceed a certain value.

6. Multi-criteria problem: decide which laptop to buy considering the price, the weight and the performance.

7. Maximum clique (community detection in social networks): determine the complete sub-graph of a graph, with the maximum number of vertices.

## 1.3 History

In the World War II, teams of scientists were asked to do research on the most efficient way to conduct the operations. In the decades after the war, the techniques became public and began to be applied more widely to problems in business, industry and society. During the industrial boom, the substantial increase in the size of the companies and organizations gave rise to more complex decision-making problems thanks to fast progress in Operations Research and in numerical analysis methodologies and advent and diffusion of computers.

## 1.4 Operations Research workflow


The main steps in studying an Operations Research problem are:

1. Define the problem.
2. Build the model.
3. Select or develop an appropriate algorithm.
4. Implement it or use an existing program.

After all this process we need to analyze the results with feedbacks.

The model obtained with this process is a simplified representation of a real-world problem. To define it we must identify the fundamental elements of the problem and the main relationships among them.

**Example :** A company produces three types of electronic devices: $D_1, D_2, D_3$, going through three main phases of the production process: assembly, refinement and quality control. The time required for each phase and product is:

| | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|
| Assembly | 80 | 70 | 120 |
| Refinement | 70 | 90 | 20 |
| Quality control | 40 | 30 | 20 |

The available resources within the planning horizon in minutes are:

| **Assembly** | **Refinement** | **Quality control** |
|---|---|---|
| 30 000 | 25 000 | 18 000 |

The unary product for each product in:

| $D_1$ | $D_2$ | $D_3$ |
|---|---|---|
| 1600 | 1000 | 2000 |

The main assumption is that the company can sell whatever it produces.

The mathematical model that describes the problem given before is the following:

- Decision variables: $x_j$ is the number of devices $D_j$ produced for $j = 1, 2, 3$.
- Objective function: we need to maximize the earning, so we have:

$$\max [1.6x_1 + 1x_2 + 2x_3]$$

- Constraints: they are on the production limit of each phase, that are:

$$80x_1 + 70x_2 + 120x_3 \leq 30000$$

$$70x_1 + 90x_2 + 20x_3 \leq 25000$$

$$40x_1 + 30x_2 + 20x_3 \leq 18000$$

- Variable type: the variables must be non-negative values, so we have $x_1, x_2, x_3 \geq 0$.

**Example:** An insurance company must decide which investments to select out of a given set of possible assets.

| **Investments** | **Area** | **Capital** ($c_j$) | **Return** ($r_j$) |
|---|---|---|---|
| A (automotive) | Germany | 150000 | 11% |
| B (automotive) | Italy | 150000 | 9% |
| C (ICT) | USA | 60000 | 13% |
| D (ICT) | Italy | 100000 | 10% |
| E (real estate) | Italy | 125000 | 8% |
| F (real estate) | France | 100000 | 7% |
| G (treasury bonds) | Italy | 50000 | 3% |
| H (treasury bonds) | UK | 80000 | 5% |

The available capital is 600000 euro. It is required to take at most five different investments. It is also required to take at maximum three investments in Italy and maximum three abroad.

The mathematical model that describes the problem given before is the following:

- Decision variables: boolean value to communicate if the investment is selected or not: $x_j = 1$ if the $j$-th investment is selected and $x_j = 0$ otherwise, for $j = 0, \ldots, 8$.
- Objective function: we need to maximize the expected return, so we have:

$$\max \left[ \sum_{j=1}^{8} c_j r_j x_j \right]$$

- Constraints: there is a constraint on the capital that insurance

$$\sum_{j=1}^{8} c_j x_j \leq 800$$

There is a constraint also on the max number of general investment and on the region they are coming from formalized asked

$$\sum_{j=1}^{8} x_j \leq 5$$

$$x_2 + x_4 + x_5 + x_7 \leq 3$$

$$x_1 + x_3 + x_6 + x_8 \leq 3$$

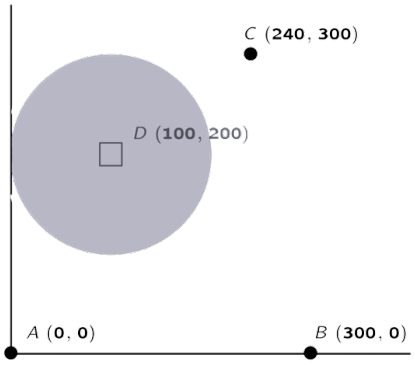
- Variable type: the variables are binary integer defined as $x_j \in \{0, 1\}$ $1 \leq j \leq 8$.

The variant requires that if any of the ICT investment is selected, then at least one of the treasury bond must be select. This requires one new constraint that is:

$$\frac{x_3 + x_4}{2} \leq x_7 + x_8$$

It is divided by two because if both ICT are selected at least one treasury bound must be selected and not two.

**Example :** Consider three oil pits, located in positions $A = (0, 0)$, $B = (300, 0)$, and $C = (240, 300)$, from which oil is extracted.




Connect them to a refinery with pipelines whose cost is proportional to the square of their length. The refinery must be at least 100 km away from point $D = (100, 200)$, but the oil pipelines can cross the corresponding forbidden zone. Give a mathematical model to decide where to locate the refinery to minimize the total pipeline cost.

- Decision variables: the coordinates of the refinery $x_1, x_2$.

- Objective function: we need to minimize the cost, so we have:

$$\begin{aligned}\min z = &\left[(x_1 - 0)^2 + (x_2 - 0)^2\right] \\ &+ \left[(x_1 - 300)^2 + (x_2 - 0)^2\right] \\ &+ \left[(x_1 - 240)^2 + (x_2 - 300)^2\right]\end{aligned}$$

- Constraints: there is a constraint on the location that is

$$\sqrt{(x_1 - 100)^2 + (x_2 - 100)^2} \geq 100$$

- Variable type: $x_1, x_2 \in \mathbb{R}$

## 1.5 Mathematical programming problem

| | **Decisions** | **Objective** | **Uncertainty** |
|---|---|---|---|
| *Mathematical programming* | single | one | - |
| *Multi-objective programming* | single | multiple | - |
| *Stochastic programming* | - | - | ✓ |
| *Game theory* | multiple | - | - |

We will analyze the mathematical programming problems. The optimization usually requires to minimizing or maximizing a given function. Note that maximizing $f(x)$ is the same problem of minimizing $-f(x)$. The problems are characterized by:

- Decision variables $\boldsymbol{x} \in \mathbb{R}^n$: numerical variables that identify a solution.
- Feasible region $\boldsymbol{X} \subseteq \mathbb{R}^n$.
- Objective function $f : \boldsymbol{X} \to \mathbb{R}$: expresses in quantitative terms the value of each feasible solution.

Solving a mathematical programming problem consists in finding a feasible solution which is globally optimum. It may happen that the problem is infeasible, unbounded, has a single optimal solution or has numerous optimal solutions. When the problem is very hard we must settle for a feasible solution that is a local optimum.

Mathematical programming can be classified in:

1. Linear Programming.
2. Integer Linear Programming.
3. Nonlinear Programming.

### 1.5.1 Multi-objective programming

Multi-objective programming can be taken into account in different ways. Suppose we wish to minimize $f_1(x)$ and maximize $f_2(x)$, we can:

1. Turn it into a single objective problem by expressing the two objectives in terms of the same unit:

   $$\min \lambda_1 f_1(x) - \lambda_2 f_2(x)$$

   for appropriate scalars $\lambda_1$ and $\lambda_2$.

2. Optimize the primary objective function and turn the other objective into a constraint:

   $$\max_{x \in \boldsymbol{X}} f_2(x) \quad f_1(x) \leq \varepsilon$$

   for an appropriate constant $\varepsilon$.

# Chapter 2

# Algorithms

## 2.1 Complexity

**Definition**

An *algorithm* is a sequence of instructions that allows to solve any instance of a problem.

An *instance* $I$ of a problem $P$ is a special case of $P$.

The execution time of an algorithm depends on the instance and on the computer. We want to evaluate the complexity of the algorithm as a function of the size of the instance independently of the hardware. Thus, we consider the number of elementary operations, and we assume that all have the same cost. Since it is usually hard to determine the exact number of elementary operations we consider the asymptotic number of elementary operations in the worst case.

**Definition**

A function $f$ is *ordered* of $g$, written:

$$f(n) = O(g(n))$$

If $\exists c > 0$ such that $f(n) \leq cg(n)$, for $n$ sufficiently large.

We can distinguish between two main classes of algorithms according to their worst-case complexity:

- Polynomial: $O(n^d)$ for a given constant $d$.
- Exponential: $O(2^n)$.

Algorithms with a high order polynomial complexity are not efficient in practice.

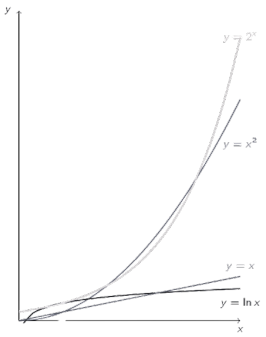



Figure 2.1: Plot of various algorithm's complexity

**Definition**

The *size of an instance* $I$ ($|I|$) is the number of bits needed to describe the instance.

A problem $P$ is *polynomially solvable* if there is a polynomial time algorithm providing an optimal solution for every instance.

For many discrete optimization problems, the best algorithm known today requires a number of elementary operations which grows, in the worst case, exponentially in the size of the instance.

**Definition**

$\mathcal{NP}$-hard computational problems are at least as difficult as a wide range of very challenging problems for which no polynomial time algorithm is known to date.

The $\mathcal{NP}$-hardness of a problem is a very strong evidence that is inherently difficult. This does not mean that it cannot admit a polynomial time algorithm.

## 2.2 Definitions

**Definition**

An algorithm is *exact* if it provides an optimal solution for every instance.

An algorithm is *heuristic* if it does not provide an optimal solution for every instance.

A *greedy algorithm* constructs a feasible solution iteratively by making at each step a locally optimal choice, without reconsidering previous choices.

## 2.3 Dynamic programming

Dynamic programming was proposed by Richard Bellman in 1953 as a general technique in which an optimal solution, composed of a sequence of elementary decisions, is determined by solving a set of recursive equations.

So, dynamic programming is applicable to any sequential decision problem for which the optimality property is satisfied.

It is used nowadays in multiple sectors: optimal control, equipment maintenance and replacement, selection of inspection points along a production line.

# Chapter 3

# Network optimization models

## 3.1 Introduction

Many decision-making problems can be formulated in terms of graphs.

**Definition**

A *graph* is a pair $G = (N, E)$, with a set of nodes $N$ and a set of edges or arcs $E \subseteq N \times N$ connecting them pairwise.

An edge connecting the nodes $i$ and $j$ is represented by $\{i, j\}$ or $(i, j)$ if the graph is *undirected* or *directed* respectively.

**Example :** A road network which connects $n$ cities can be modelled by a graph where a city corresponds to a node, and a connection corresponds to an edge.


The graph on the left is undirected and defined as:

- $N = \{1, 2, 3, 4, 5\}$
- $E = \{\{1, 2\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$

The graph on the right is directed and defined as:

- $N = \{1, 2, 3, 4, 5\}$
- $E' = \{(1, 2), (1, 4), (2, 3), (2, 4), (3, 4), (3, 5), (4, 5)\}$

**Definition**

Two nodes are *adjacent* if they are connected by an edge.

An edge $e$ is *incident* in a node $v$ if $v$ is an endpoint of $e$.

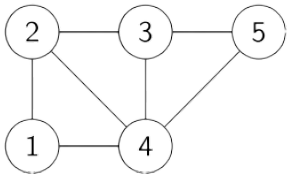
In undirected graphs, the *degree* of a node is the number of incident edges in a given node.

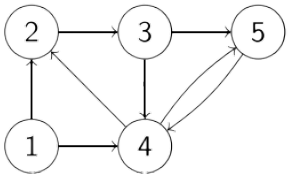
In directed graph, the *in-degree* of a node is the number of arcs that have it as successor.

In directed graph, the *out-degree* of a node is the number of arcs that have it as predecessor.

**Example:** In the undirected graph, we have that nodes 1 and 2 are adjacent and 1 and 3 are not. The edge $\{1, 2\}$ is incident in nodes 1 and 2. Node 1 has a degree 2, and node 4 has a degree of 4.

In the directed graph, the node 1 has an in-degree equal to 0, and an out-degree equal to 2.




**Definition**

A *directed path from* $i \in N$ *to* $j \in N$ is a sequence of arcs $p = \langle\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}\rangle$ connecting nodes $v_1$ and $v_k$.

Nodes $u$ and $v$ are *connected* if there is a path connecting them. A graph $(N, E)$ is connected if $u, v$ are connected for any $u, v \in N$.

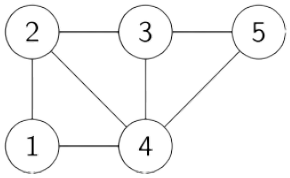
A graph is *strongly connected* if $u$ and $v$ are connected by a directed path for any $u, v \in N$.

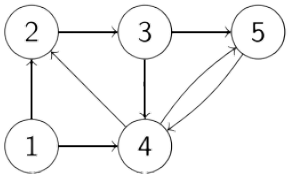
A *cycle* or circuit is a path with $v_1 = v_k$.

**Example:** The undirected graph has a path $\langle\{2, 3\}, \{3, 4\}, \{4, 5\}\rangle$ from node 2 to node 5. So we say those nodes are connected.

The directed graph has a directed path $\langle(3, 5), (5, 4), (4, 2), (2, 3), (3, 4)\rangle$ from node 3 to node 4. So we say those nodes are not strongly connected.

In the undirected graph $\langle\{2, 3\}, \{3, 5\}, \{5, 4\}, \{4, 2\}\rangle$ is a cycle. In the directed graph $\langle(2, 3), (3, 4), (4, 2)\rangle$ is a circuit.



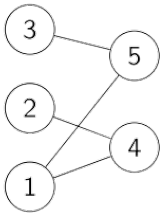



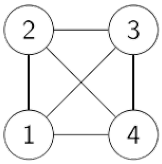

**Definition**

A graph is *bipartite* if there is a partition $N = N_1 \cup N_2$ with $N_1 \cap N_1 = \varnothing$ such that no edge connects nodes in the same subset.

A graph is *complete* if $E = \{\{v_i, v_j\} | v_i, v_j \in N \wedge i \leq j\}$

**Example:** The graphic on the left is bipartite because we can find two subsets of nodes such that $N = N_1 \cup N_2$ with $N_1 \cap N_1 = \varnothing$ that are: $N_1 = \{1, 2, 3\}$ and $N_2 = \{4, 5\}$. The graph on the right is a complete graph because all the nodes are connected with each other.




**Definition**

Given a directed graph $G = (N, A)$ and $S \subset NM$, the *outgoing cut* induced by $S$ is:

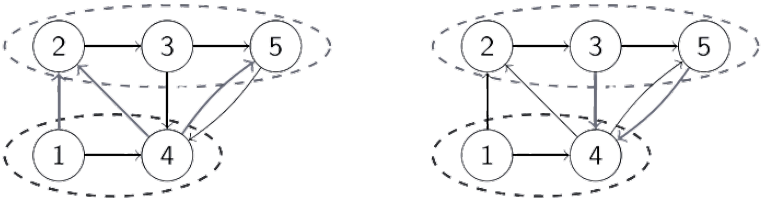
$$\delta^+(S) = \{(u, v) \in A | u \in S \wedge v \in N - S\}$$

Given a directed graph $G = (N, A)$ and $S \subset NM$, the *incoming cut* induced by $S$ is:

$$\delta^-(S) = \{(u, v) \in A | v \in S \wedge u \in N - S\}$$

**Example:** In the following graph we can note that:

- $\delta^+(\{1, 4\}) = \{(1, 2), (4, 2), (4, 5)\}$
- $\delta^-(\{1, 4\}) = \{(3, 4), (5, 4)\}$




An undirected graph with $n$ nodes has at most $m = n(n-1)$ arcs. A directed graph with $n$ nodes has at most $m = \dfrac{n(n-1)}{2}$ arcs.

**Definition**

Given $m$, the number of arcs or edges, and $n$, the number of nodes of the graph, we have that a graph is called *dense* if:

$$m \approx n^2$$

Given $m$, the number of arcs or edges, and $n$, the number of nodes of the graph, we have that a graph is called *sparse* if:

$$m \ll n^2$$

The best way to represent a dense graph is by using an $n \times n$ adjacency matrix, that is defined in the following way:

$$\begin{cases} a_{ij} = 1 & \text{if } (i,j) \in A \\ a_{ij} = 0 & \text{otherwise} \end{cases}$$

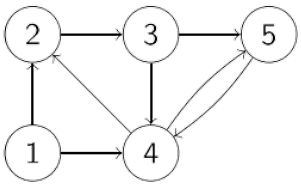
The best way to represent a sparse graph is by using lists of successors for each node.

**Example:** The adjacency matrix for the following graph is:

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

And the list of successor is:

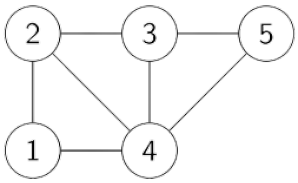
$$S(1) = \{2,4\} \quad S(2) = \{3\} \quad S(3) = \{4,5\} \quad S(4) = \{2,5\} \quad S(5) = \{4\}$$



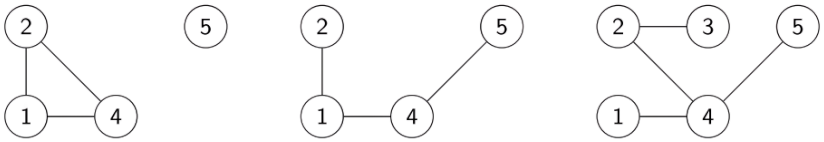

**Definition**

$G' = (N', E')$ is a *sub-graph* of $G = (N, E)$ if $N' \subseteq N$ and $E' \subseteq E$.
A *tree* $G_T = (N', T)$ *of* $G$ is a connected and acyclic sub-graph of $G$.
$G_T = (N', T)$ is a *spanning tree of* $G$ if it contains all nodes in $G$.
The *leaves* of a tree are the nodes of degree one.

**Example :** Given the following graph:




We can obtain: a sub-graph, a tree, and a spanning tree:




The trees have the following properties:

1. Every tree with $n$ nodes has $n - 1$ edges.
2. Any pair of nodes in a tree is connected via a unique path.
3. By adding a new edge to a tree, we create a unique cycle.
4. Let $G_T = (N, T)$ be a spanning tree of $G = (N, E)$. Consider an edge $e \notin T$ and the unique cycle $C$ of $T \cup \{e\}$. For each edge $f \in C - \{e\}$, the sub-graph $T \cup \{e\} - \{f\}$ is also a spanning tree of $G$.
5. Let $F$ be a partial tree contained in an optimal tree of $G$. Consider $e\{u, v\} \in \delta(S)$ of minimum cost, then there exists a minimum cost spanning tree of $G$ containing $e$.

6. If $c_j \geq 0$ for all $(i, j) \in A$, there is at least one shortest path which is simple.

7. The directed graph G representing any project is acyclic (it is a DAG).

8. The minimum overall project duration is the length of the longest path from $s$ to $t$.

9. Given a feasible flow $x$ from $s$ to $t$, for each cut $\delta(S)$ separating $s$ from $t$, we have $\varphi(S) = \varphi(\{s\})$.

10. For every feasible flow $x$ from $s$ to $t$ and every cut $\delta(S)$, with $S \subseteq V$, separating $s$ from $t$, we have $\varphi(S) \leq k(S)$. In other words, the value of the flow is less or equal to the capacity of the cut.

**Proof Property one :** We will demonstrate this property with a proof by induction. For the base case we have that the claim holds for $n = 1$. For the inductive step we have to show that, if this is true for trees with $n$ nodes, then it is also true for those with $n+1$ nodes. Let $T_1$ be a tree with $n + 1$ nodes and recall that any tree with $n \geq 2$ nodes has at least two leaves. By deleting one of the leaf and its incident edge we obtain a tree $T_2$ with $n$ nodes. By induction hypothesis, $T_2$ has $n - 1$ edges. Therefore, the tree $T_1$ has $n - 1 + 1 = n$ edges. ■

**Proof Property five :** By contradiction, assume $T^* \subseteq E$ is a minimum cost spanning tree with $F \subseteq T^*$ and $e \notin T^*$. Adding edge $e$ to $T^*$ creates the cycle $C$. Let $f \in \delta(S) \cap C$:

- If $c_e = c_f$, then $T^* \cup \{e\} - \{f\}$ is also optimal since it has same cost of $T^*$.
- If $c_e < c_f$, then $c\,(T^* \cup \{e\} - \{f\}) < C(T^*)$, hence $T^*$ is not optimal. ■

**Proof Property seven :** By contradiction, if $A_{i1} \propto A_{12}, \ldots, A_{jk} \propto A_{ki}$ there would be a logical inconsistency. ■

**Proof Property eight :** Since any $s - t$ path represents a sequence of activities that must be executed in the specified order, its length provides a lower bound on the minimum overall project duration. ■

**Proof Property ten :** By the definition of value of the flow through the cut $\delta(S)$ we have:

$$\varphi(S) = \sum_{(i,j)\in\delta^+(S)} x_{ij} - \sum_{(i,j)\in\delta^-(S)} x_{ij}$$

and because $0 \leq x_{ij} \leq k_{ij}$, for any $(i,j) \in A$ we also have

$$\sum_{(i,j)\in\delta^+(S)} x_{ij} - \sum_{(i,j)\in\delta^-(S)} x_{ij} \leq \sum_{(i,j)\in\delta^+(S)} k_{ij} = k(S)$$

Then, $\varphi(S) \leq k(S)$. ■

## 3.2 Graph reachability problem

Given the directed graph $G = (N, A)$ and a node $s$, determine all the nodes that are reachable from $s$.

**Algorithm 1** Graph reachability problem

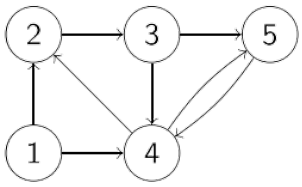
```
1: Q ← {s}
2: M ← {∅}
3: while Q ≠ 0 do
4:     u ← node in Q
5:     Q ← Q − {u}
6:     M ← M ∪ {u}
7:     for (u, v) ∈ δ+(u) do
8:         if v ∉ M and v ∉ Q then
9:             Q ← Q ∪ {v}
10:        end if
11:    end for
12: end while
```

The worst case complexity of the previous algorithm is $O(n^2)$.

**Example:** Given the following graph and $s = 2$ the algorithm makes the following steps:

1. $Q = \{2\}$ $M = \varnothing$
2. $Q = \{3\}$ $M = \{2\}$
3. $Q = \{4, 5\}$ $M = \{2, 3\}$
4. $Q = \{5\}$ $M = \{2, 3, 4\}$
5. $Q = \varnothing$ $M = \{2, 3, 4, 5\}$

So the nodes $\{2, 3, 4, 5\}$ are reachable from node two.




## 3.3 Minimum spanning tree problem

Given an undirected graph $G = (N, E)$ and a cost function, find a spanning tree $G_T = (N, T)$ of minimum total cost:

$$\min_{T \in X} \sum_{e \in T} c_e$$

where $X$ is the set of all spanning trees of $G$.

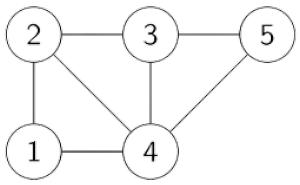
**Theorem (*Cayley*)**

A complete graph with $n$ nodes ($n \geq 1$) has $n^{(n-2)}$ spanning trees.

To find the spanning tree with the minimum total cost we can use an algorithm that iteratively builds the spanning tree. The idea of the algorithm is:

1. Select any node arbitrarily, and connect it to the nearest distinct node.
2. Identify the unconnected node that is closest to a connected node, and then connect these two nodes. Repeat this step until all nodes have been connected.

**Example :** Apply the Prim's algorithm to the following graph:

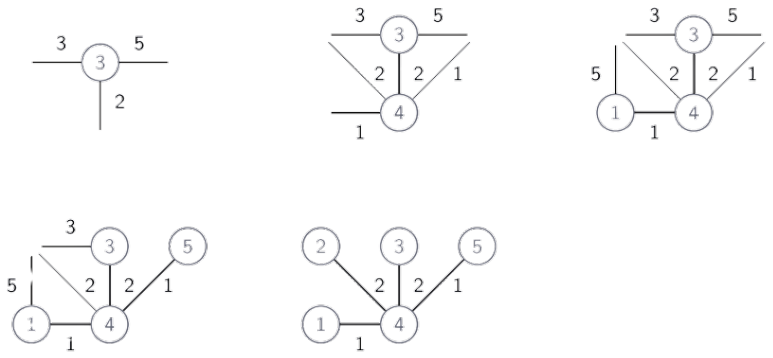



We select the node 3 as starting node, and so we have $S = \{3\}\, T = \{\varnothing\}$, then:

- The edge with minimum cost is the one that connects the nodes 3 and 4. Now we have: $S = \{3, 4\}$ and $T = \{\{3, 4\}\}$.
- The edge with minimum cost is the one that connects the nodes 1 and 4. Now we have: $S = \{1, 3, 4\}$ and $T = \{\{3, 4\}, \{1, 4\}\}$.

- The edge with minimum cost is the one that connects the nodes 4 and 5. Now we have: $S = \{1, 3, 4, 5\}$ and $T = \{\{3, 4\}, \{1, 4\}, \{4, 5\}\}$.
- The edge with minimum cost is the one that connects the nodes 4 and 5. Now we have: $S = N$ and $T = \{\{3, 4\}, \{1, 4\}, \{4, 5\}, \{2, 4\}\}$.

The total cost in this case is equal to $c(T) = 6$. Graphically, we have the following graphs:




Given a connected graph $G = (N, E)$ with edge cost the algorithm outputs $T \subseteq N$ of edges of $G$ such that $G_T = (N, T)$ is a minimum cost spanning tree of $G$.

**Algorithm 2** Prim's algorithm for the minimum cost spanning tree problem

1: $S \leftarrow \{u\}$
2: $T \leftarrow \{\varnothing\}$
3: **while** $|T| < n - 1$ **do**
4: $\quad \{u, v\} \leftarrow$ edge in$\delta(S)$of minimum cost
5: $\quad S \leftarrow S \cup \{v\}$
6: $\quad T \leftarrow T \cup \{u, v\}$
7: **end while**

Where $u \in S$ and $v \in N - S$. The worst-case complexity is $O(n^2)$.

**Proposition**

Prim's algorithm is exact.

The exactness does not depend on the choice of the first node nor on the selected edge of minimum cost in $\delta(S)$.

**Proposition**

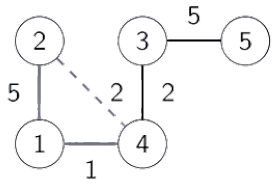
Prim's algorithm is greedy.

At each step a minimum cost edge is selected among those in the cut $\delta(S)$ induced by the current set of nodes $S$.

**Definition**

Given a spanning tree $T$, an edge $e \notin T$ is *cost decreasing* if when $e$ is added to $T$ it creates a cycle $C$ with $C \subseteq T \cup \{e\}$ and $\exists f \in C - \{e\}$ such that $c_e < c_f$.

**Example:** Given the following graph




$c_{24} = 2 < 5 = c_{12}$

Because $c(T \cup \{e\} - \{f\}) = c(T) + c_e - c_f$, if $e$ is cost decreasing, then

$$c(T \cup \{e\} - \{f\}) < c(T)$$

**Theorem (*Tree optimaility condition*)**

A tree $T$ is of minimum total cost if and only if no cost decreasing edge exists.

**Proof of direct implication:** If a cost-decreasing edge exists, then T is not of minimum total cost. ■

**Proof of inverse implication:** If no cost-decreasing edge exists, then $T$ is of minimum total cost. Let $T^*$ be a minimum cost spanning tree found by Prim's algorithm. It can be verified that, by exchanging one edge at a time, $T^*$ can be iteratively transformed into $T$ without modifying the total cost. Thus, $T$ is also optimal. ■

The optimality condition allows us to verify whether a spanning tree $T$ is optimal: it is sufficient to check that each $e \in E - T$ is not a cost-decreasing edge.

## 3.4 Graph shortest path problem

### 3.4.1 Dijkstra's algorithm

Given a directed graph $G = (N, A)$ with a cost $c_j \in R$ for each arc $(i, j) \in A$, and two nodes $s$ and $t$, determine a minimum cost path from $s$ to $t$.

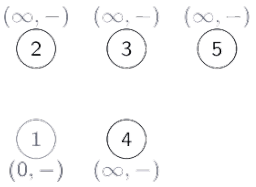
Each value $c_j$ represents the cost of arc $(i, j) \in A$. Node $s$ is the origin, or source, and $t$ is the destination, or sink.

**Definition**

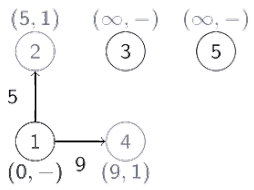
A path is *simple* if no node is visited more than once.

The input of Dijkstra's algorithm is a graph $G = (N, A)$ with non-negative arc costs and $s \in N$. This algorithm will give us the shortest paths from $s$ to all other nodes of $G$. The idea behind it is to consider the nodes in increasing order of cost of the shortest path from $s$ to any one of the other nodes. To each node $j \in N$, we assign a label $L_j$ which corresponds to the cost of a minimum cost path from $s$ to $j$ and a label $pred_j$ that is the predecessor of $j$ in the shortest path from $s$ to $j$. Note that this algorithm is greedy with respect to path from $s$ to $j$.

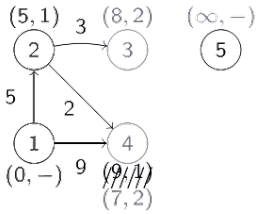
**Example :** Given a graph, and selecting 1 as the initial node, we set the following labels.



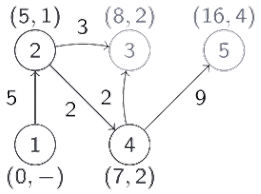

Next, we check all the arcs going from the starting node to other nodes, and we update the nodes label (the node one will always have no predecessor and null cost).



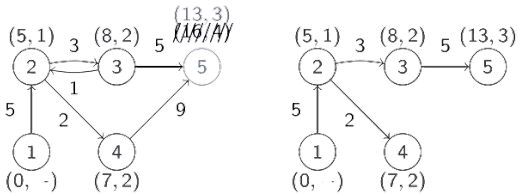

Now, we move to the node 2 and check all the reachable nodes. In this case we found the shortest path from the initial node to 4, so we update the label of 4.




Now, we move to the node 4 that is the closest node to $s$, and we check all the arcs going in other nodes.




We do the same for the remaining nodes, again in increasing order of cost from the start.




It is now possible to retrieve backward the shortest path to any node of the graph using the predecessor label. For instance, we have that the shortest path from $s$ to 5 has a cost of 13 and the path is $(1, 2, 3, 5)$.

The inputs of the algorithm are: a graph $G = (N, E)$ with non-negative arc costs, $s \in N$.

**Algorithm 3** Dijkstra's algorithm for the graph shortest path problem

```
1: S ← ∅
2: X ← {s}
3: for u ∈ N do
4:     L_u ← +∞
5: end for
6: L_s ← 0
7: while |S| ≠ n do
8:     u ← argmin{L_i | i ∈ X}
9:     X ← X − {u}
10:    S ← S ∪ {u}
11:    for (u, v) ∈ δ⁺(u) such that L_v > L_u + c_uv do
12:        L_v ← L_u + c_uv
13:        pred_v ← u
14:        X ← X ∪ {v}
15:    end for
16: end while
```

The worst case complexity of this algorithm is $O(n^3)$.

**Proposition**

Dijkstra's algorithm is exact.

**Proof:** At the $k$-th step: $S = \{s, i_2, \ldots, i_k\}$ and

$$L_j = \begin{cases} \text{cost of a minimum path from } s \text{ to } j,\ j \in S \\ \text{cost of a minimum path with all intermediate nodes in } S,\ j \notin S \end{cases}$$

By induction on the number $k$ of steps:

- Base case: it is easy to see that the statement holds for $k = 1$, since

$$S = \{s\},\ L_s = 0,\ L_j = +\infty,\ \forall j \neq S$$

- Inductive step: we must prove that, if the statement holds at the $k$-th step, it must also hold for the $(k+1)$-th step.

In the $(k+1)$-th step let $u \notin S$be the node that is inserted in $S$ and $\varnothing$ the path from $s$ to $u$ such that:

$$L_v + c_{uv} \leq L_i + C_{ij},\ \forall (i,j) \in \delta^+(S)$$

Let us verify that every path $\pi$ from $s$ to $u$ has $c(\pi) \geq c(\varnothing)$. There exist $i \in S$ and $j \notin S$ such that:

$$\pi = \pi_1 \cup \{(i,j)\} \cup \pi_2$$

Where $(i, j)$ is the first arc in $\pi \cap \delta^+(S)$. Moreover:

$$c(\pi) = c(\pi_1) + c_{ij} + c(\pi_2) \geq L_i + c_{ij}$$

Because $c_{ij} \geq 0$, thus, $c(\pi_2) \geq 0$, and by the induction assumption, $c(\pi_1) \geq L_i$. Finally, by the choice of $(v, u)$ we have:

$$L_i + c_{ij} \geq L_v + c_{vu} = c(\varnothing)$$

■

We can note that:

- A set of the shortest paths from $s$ to all the nodes $j$ can be retrieved via the vector of predecessors.
- The union of a set of the shortest paths from node $s$ to all the other nodes of $G$ is the shortest path trees rooted at $s$. Such shortest path trees have nothing to do with minimum cost spanning trees.
- Dijkstra's algorithm does not work when there are arcs with negative cost.

### 3.4.2 Floyd-Warshall's algorithm

If the graph $G$ contains a circuit of negative cost, the shortest path problem may not be well-defined. Each time the circuit appears, the cost decreases. There is no finite shortest path from $s$ to $t$. Floyd-Warshall's algorithm detects the presence of circuits with negative cost. It provides a set of the shortest paths between all pairs of nodes, even when there are arcs with negative cost. It is based on iteratively applying a triangular operation. The algorithm uses two $n \times n$ matrices, D, P, whose elements correspond, at the end of the algorithm to:

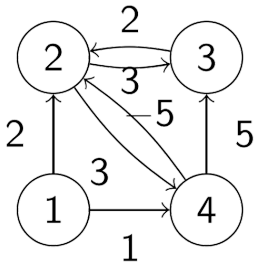
- $d_{ij}$, that is the cost of the shortest path from $i$ to $j$.
- $p_{ij}$, that is the predecessor of $j$ in that shortest path from $i$ to $j$.

**Definition**

The *triangular operation* states that for each pair of nodes $i, j$ with $i \neq u$ and $j \neq u$ (including case $i = j$), check whether when going from $i$ to $j$ it is more convenient to go via $u$, so if the following relation holds:

$$d_{iu} + d_{uj} < d_{ij}$$

**Example :** Given the following graph:




We have to initialize the matrices in the following way:

$$D = \begin{bmatrix} 0 & 2 & \infty & 1 \\ \infty & 0 & 3 & 3 \\ \infty & 2 & 0 & \infty \\ \infty & -5 & 5 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

The first iteration with $u = 1$ (we consider the first row and the first column) leave the matrices unchanged, in fact the triangular operation is always satisfied:

- $0 = d_{22} < d_{21} + d_{12} = \infty + 2$ (no changes).
- $3 = d_{23} < d_{21} + d_{13} = \infty + \infty$ (no changes).
- $3 = d_{24} < d_{21} + d_{14} = \infty + 1$ (no changes).
- $2 = d_{32} < d_{31} + d_{12} = \infty + 2$ (no changes).
- $0 = d_{33} < d_{31} + d_{13} = \infty + \infty$ (no changes).
- $\infty = d_{34} < d_{31} + d_{14} = \infty + 1$ (no changes).
- $-5 = d_{42} < d_{41} + d_{12} = \infty + 2$ (no changes).
- $5 = d_{43} < d_{41} + d_{13} = \infty + \infty$ (no changes).
- $0 = d_{44} < d_{41} + d_{14} = \infty + 1$ (no changes).

The second iteration with $u = 2$ (we consider the second row and the second column) changes the matrices and halts the algorithm because of the negative arc found:

- $0 = d_{11} < d_{12} + d_{21} = 2 + \infty$ (no changes).
- $\infty = d_{13} < d_{12} + d_{23} = 2 + 3$ ($p_{13} \leftarrow p_{23}$).
- $1 = d_{14} < d_{12} + d_{24} = 2 + 3$ (no changes).
- $\infty = d_{31} < d_{32} + d_{21} = 2 + \infty$ (no changes).
- $0 = d_{33} < d_{32} + d_{23} = 2 + 3$ (no changes).

- $\infty = d_{34} < d_{32} + d_{24} = 2 + 3$ ($p_{34} \leftarrow p_{24}$).
- $\infty = d_{41} < d_{42} + d_{21} = 5 + \infty$ (no changes).
- $5 = d_{43} < d_{42} + d_{23} = -5 + 3$ ($p_{43} \leftarrow p_{23}$).
- $0 = d_{44} < d_{42} + d_{24} = -5 + 3$ (negative cost circuit found).

The inputs of the algorithm are: a directed graph $G = (N, A)$ with an $n \times n$ cost matrix, $C = [c_{ij}]$.

**Algorithm 4** Walshall-Floyd's algorithm

```
1: for i ∈ N do
2:     for j ∈ N do
3:         p_ij ← i
4:         d_ij ← { 0      i = j
                  { c_ij   i ≠ j ∧ (i, j) ∈ A
                  { +∞     otherwise
5:     end for
6: end for
7: for u ∈ N do
8:     for i ∈ N − {u} do
9:         for j ∈ N − {u} do
10:            if d_iu + d_uj d_ij then
11:                p_ij ← p_uj
12:                d_ij ← d_iu + d_uj
13:            end if
14:        end for
15:    end for
16:    for i ∈ N do
17:        if d_ii < 0 then
18:            return
19:        end if
20:    end for
21: end for
```

Since in the worst case the triangular operation is executed for all nodes $u$ and for each pair of nodes $i$ and $j$, the overall complexity is $O(n^3)$.

**Proposition**

Floyd-Warshall's algorithm is exact.

**Proof:** Assume that the nodes of $G$ are numbered from 1 to $n$. Verify that, if the node index order is followed, after the $u$-th cycle the value $d_{ij}$ (for any $i$ and $j$) corresponds to the cost of the shortest path from $i$ to $j$ with only intermediate nodes in $1, \ldots, u$. ■

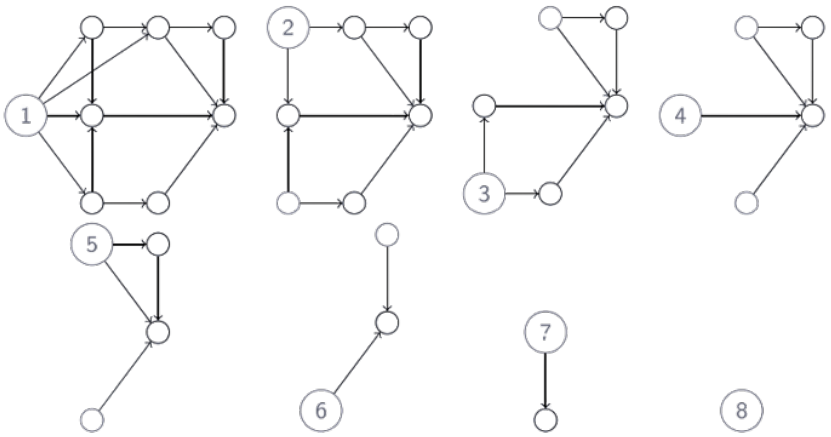
### 3.4.3 Topological order algorithm

Given a directed acyclic graph $G = (N, A)$ with a cost $c_i j \in \mathbb{R}$ for each $(i, j) \in A$, and nodes $s$ and $t$, determine the shortest (longest) path from $s$ to $t$. The DAGs have the following property called topological order: the nodes of any directed acyclic graph $G$ can be ordered topologically, that is, indexed so that for each arc $(i, j) \in A$ we have $i < j$. The topological order can be exploited in a very efficient dynamic programming algorithm to find shortest (or longest) paths in DAGs. Given $G = (N, A)$ represented via the lists of predecessors $\delta^-(v)$ and successors $\delta^+(v)$ for each node $v$ we have to follow these steps:

1. Assign the smallest positive integer not yet assigned to a node $v \in N$ with $\delta^-(v) = \varnothing$.
2. Delete the node $v$ with all its incident arcs.
3. Go back to the point one until there are nodes in the current sub-graph.

The complexity of this algorithm is $O(m)$ where $m = |A|$, because each node/arc is considered at most once.

**Example:** A graphical example of the algorithm is the following:



### 3.4.4 DAGs' dynamic programming algorithm

Any shortest path from 1 to $\pi_t$, with at least 2 arcs, can be subdivided into two parts: $\pi_t$ and $(i,t)$, where $\pi_t$ is the shortest sub-path from $s$ to $i$. For each node $i = 1,\ldots,t$, let $L_i$ be the cost of the shortest path from 1 to $i$. Then:

$$L_t = \min_{(i,t)\in\delta^-(t)}\{L_i + c_{it}\}$$

where the minimum is taken over all possible predecessors $i$ of $t$. If $G$ is directed, acyclic and topologically ordered, the only possible predecessors of $t$ in the shortest path $\pi_t$ from 1 to $t$ are those with index $i < t$. Thus,

$$L_t = \min_{i<t}\{L_i + c_{it}\}$$

In a graph with circuits, any node different from $t$ can be a predecessor of $t$ in $\pi_t$.

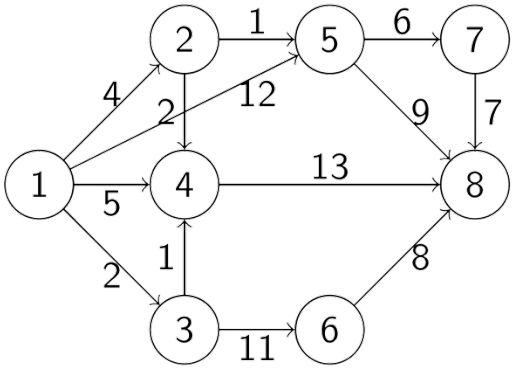
For DAGs whose nodes are topologically ordered $L_{t-1},\ldots,_L 1$ satisfy the same type of recursive relations:

$$L_{t-1} = \min_{i<t-1}\{L_i + c_{i,t-1}\};\ldots;L_2 = \min_{i=1}\{L_i + c_i 2\} = L_1 + c_{12}; L_1 = 0$$

which can be solved in the reverse order:

$$L_1 = 0; L_2 = L_1 + c_{12};\ldots;L_t = \min_{i<t-1}\{L_i + c_t\}$$

**Example :** Given the following graph:




The dynamic programming to find the shortest paths in DAGs follows this workflow:

- $L_1 = 0 \rightarrow \text{pred}_1 = 1$

- $L_2 = L_1 + c_{12} = 4 \rightarrow \text{pred}_2 = 1$
- $L_3 = L_1 + c_{13} = 2 \rightarrow \text{pred}_3 = 1$
- $L_4 = \min_{i=1,2,3}\{L_i + c_{i4}\} = \min 5, 6, 3 = 3 \rightarrow \text{pred}_4 = 3$
- $L_5 = \min_{i=1,2}\{L_i + c_{i5}\} = \min 12, 5 = 5 \rightarrow \text{pred}_5 = 2$
- $L_6 = L_3 + c_{36} = 13 \rightarrow \text{pred}_6 = 3$
- $L_7 = L_5 + c_{57} = 11 \rightarrow \text{pred}_7 = 5$
- $L_8 = \min_{i=4,5,6,7}\{L_i + c_{i8}\} = \min 16, 14, 21, 18 = 14 \rightarrow \text{pred}_8 = 5$

**Algorithm 5** Dynamic programming to find the shortest paths in DAGs

1: Sort the nodes of $G$ topologically
2: $L_1 \leftarrow 0$
3: **for** $j = 2, \ldots, n$ **do**
4: $\quad L_j \leftarrow \min\{L_i + c_{ij} | (i,j) \in \delta^-(j) \wedge i < j\}$
5: $\quad pred_j \leftarrow v$ such that $(v,j) = \text{argmin}\{L_i + c_{ij} | (i,j) \in \delta^-(j) \wedge i < j\}$
6: **end for**

Since the nodes are topologically ordered each node and each arc is considered exactly once, so we have a complexity of $O(m)$, where $m = |A|$.

It is possible to use the same algorithm to find the longest path with the formula:

$$L_t = \max_{i<t}\{L_i + c_{it}\}, \ldots$$

**Proposition**

The Dynamic Programming algorithm for DAGs is exact.

**Proof:** This is due to the optimality principle: for any shortest (longest) path from 1 to $t, \pi_t$, there exists $i < j$ such that the path can be subdivided into two parts: $\pi_i$ and $(i,t)$, where $\pi_i$ is a minimum (maximum) length from $s$ to $i$. ■

### 3.4.5 Project planning algorithm

**Definition**

A *project* consists of a set of $m$ activities with their duration: activity $A_i$ has an estimated duration $d_j \geq 0, i = 1, \ldots, m$.

Some pairs of activities are subject to a *precedence constraint*: $A_i \propto A_j$ indicates that $A_j$ can start only after the end of $A_i$.

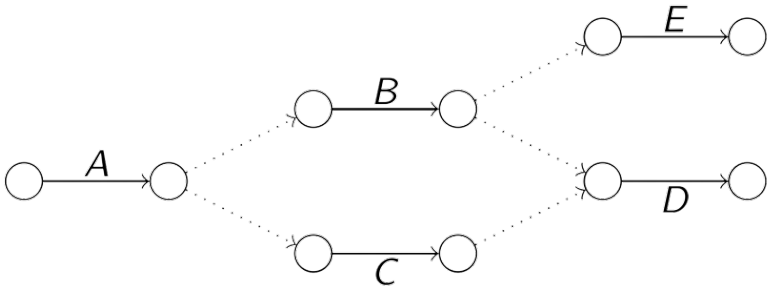
A project can be represented by a directed graph $G = (N, A)$ where each arc corresponds to an activity, and its arc length represents the duration of the corresponding activity. To account for the precedence constraints, the arcs must be positioned so that

$$A_i \propto A_j$$

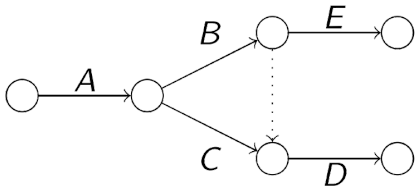
implies that there exists a directed path where the arc associated to $A_i$ precedes the arc associated to $A_j$. Therefore, a node $v$ marks an event corresponding to the end of all the activities $(i, v) \in \delta^-(v)$ and, hence, the possible beginning of all those $(v, j) \in \delta^+(v)$. We can also introduce new nodes so that graph $G$:

- Contains a unique initial node $s$ corresponding to the event beginning of the project.
- Contains a unique final node $t$ corresponding to the event end of the project.
- Does not contain multiple arcs (with same endpoints).

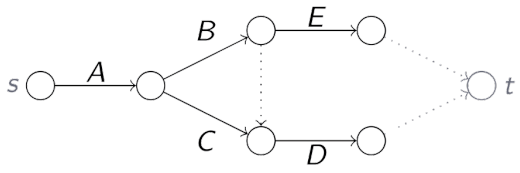
**Example:** A project can be defined in the following way:




This graph can be simplified by contracting some arcs, but we have to pay attention in introducing unwanted precedence constraints. The correct contracted graph is:




We can add the initial and the final nodes:




The problem to solve is: given a project, schedule the activities to minimize the overall project duration. To solve this problem with can use the critical path method that determines:

- A schedule that minimizes the overall project duration.
- The slack of each activity.

The input of the algorithm is the graph $G$ representing the project and find a topological order of the nodes. The steps are:

1. Consider the nodes by increasing indices and, for each $h \in N$, find the earliest time $T_{min_h}$ at which the event associated to node $h$ can occur ($T_{min_n}$ corresponds to the minimum project duration).
2. Consider the nodes by decreasing indices and, for each $h \in N$, find the latest time $T_{max_h}$, at which the event associated to node $h$ can occur without delaying the project completion date beyond $T_{min_n}$.
3. For each activity $(i, j) \in A$, find the slack: $\sigma_{ij} = T_{max_j} - T_{max_i} - d_{ij}$.

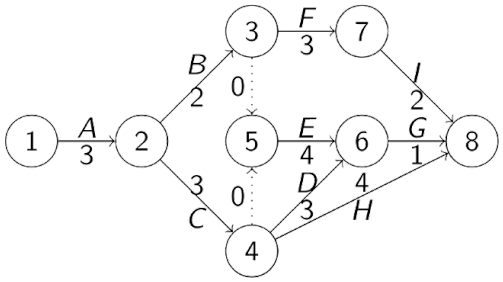
**Example :** Consider the following project:

| **Activity** | **Duration** | **Predecessors** |
|---|---|---|
| A | 3 | - |
| B | 2 | A |
| C | 3 | A |
| D | 3 | C |
| E | 4 | B,C |
| F | 3 | B |
| G | 1 | E,D |
| H | 4 | C |
| I | 2 | F |

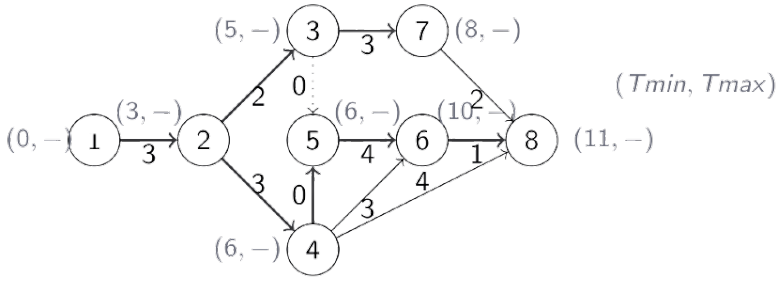
With these precedence constraints:

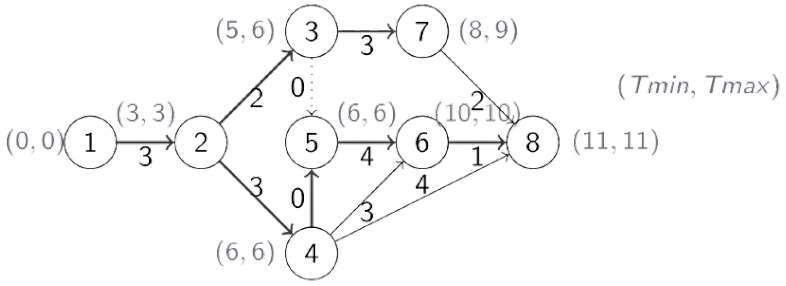
$$A \propto B, A \propto C, C \propto D, B \propto E, C \propto E, B \propto F, E \propto G, D \propto G, C \propto H, F \propto I$$

Determine the overall minimum duration of the project and the slack for each activity. We have to find the graph associated to the given problem, that is:




The first two phases are:




So, we have that the longest path that is critical is: $(1, 2, 4, 5, 6, 8)$.

This algorithm inputs are: a graph $G = (N, A)$, with $n = |N|$ and the duration $d_{ij}$ associated to each $(i, j) \in A$.

**Algorithm 6** Algorithm for the critical path method

1: Sort the nodes of $G$ topologically
2: $T_{min_1} \leftarrow 0$
3: **for** $j = 2, \ldots, n$ **do**
4: $\quad T_{min_j} \leftarrow \max\{T_{min_i} + d_{ij} | (i,j) \in \delta^-(j)\}$
5: **end for**
6: $T_{max_n} \leftarrow T_{min_n}$
7: **for** $i = n-1, \ldots, 1$ **do**
8: $\quad T_{max_j} \leftarrow \min\{T_{max_j} - d_{ij} | (i,j) \in \delta^+(j)\}$
9: **end for**

The complexity is $O(m)$, where $m = |A|$.

**Definition**

An activity $(i,j)$ with zero slack:

$$\sigma_{ij} = T_{max_j} - T_{min_i} - d_{ij} = 0$$

is *critical*.

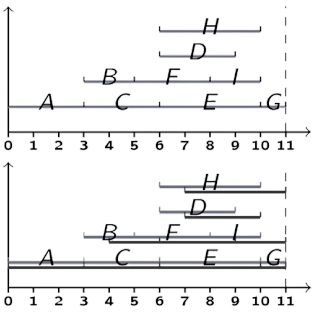
A *critical path* is an $s-t$ path only composed of critical activities (it always exists).

To find the slack of a project it is also possible to use Gantt charts, that were introduced in 1910 by Henry Gantt. It provides a temporal representations of the project.

**Example :** The result of the previous example are the following:

| $(i,j)$ | $d_{ij}$ | $T_{min_i}$ | $T_{max_j}$ |
|---|---|---|---|
| A | 3 | 0 | 3 |
| B | 2 | 3 | 6 |
| C | 3 | 3 | 6 |
| D | 3 | 6 | 10 |
| E | 4 | 6 | 10 |
| F | 3 | 5 | 9 |
| G | 1 | 10 | 11 |
| H | 4 | 6 | 11 |
| I | 2 | 8 | 11 |

And they can be represented with a Gantt chart as follows:




## 3.5 Minimum network flow problem

The network flows problems involves the distribution of a given product from a set of sources to a set of users to optimize a given objective function.

**Definition**

A network is a directed and connected graph $G = (V, A)$ with a source $s \in V$ and a sink $t \in V$, with $s \neq t$, and a capacity $k_{ij} \geq 0$ for each arc $(i, j) \in A$.

A *feasible flow* $\boldsymbol{x}$ from $s$ to $t$ is a vector $\boldsymbol{x} \in \mathbb{R}^m$ with a component $x_{ij}$ for each arc $(i, j) \in A$ satisfying the capacity constraints:

$$0 \leq x_{ij} \leq k_{ij} \quad \forall (i, j) \in A$$

and the flow balance constraints at each intermediate node $u \in V (u \neq s, t)$:

$$\sum_{(i,u) \in \delta^-(u)} x_{iu} = \sum_{(u,j) \in \delta^+(u)} x_{uj} \quad \forall u \in N - \{s, t\}$$

The *value of flow* $x$ is:

$$\varphi = \sum_{(s,j) \in \delta^+(s)} x_{sj}$$

Given a network and a feasible flow $x$, an arc $(i, j) \in A$ is *saturated* if $x_{ij} = k_{ij}$.

Given a network and a feasible flow $x$, an arc $(i, j) \in A$ is *empty* if $x_{ij} = 0$.

Given a network $G = (V, A)$ with an integer capacity $k_{ij}$ for each arc $(i, j) \in A$, and nodes $s, t \in V$, determine a feasible flow from $s$ to $t$ of maximum value.

If there are many sources/sinks with a unique type of product, dummy

nodes $s^*$ and $t^*$ can be added. The linear programming model of the problem is to maximize $\max \varphi$ such that:

$$\sum_{(u,j)\in\delta^+(u)} x_{uj} - \sum_{(i,u)\in\delta^-(u)} x_{iu} = \begin{cases} \varphi & u = s \\ -\varphi & u = t \\ 0 & \text{otherwise} \end{cases}$$

where $\varphi$ denotes the value of the feasible flow $x$, $0 \leq x_{ij} \leq k_{ij}$ with $\varphi, x_{ij} \in \mathbb{R}$, and $(i,j) \in A$.

**Definition**

A *cut separating s from t* is $\delta(S)$ of $G$ with $s \in S \subset V$ and $t \in V - S$. There are $2^{n-2}$ cuts separating $s$ from $t$, where $n = |V|$.

The *capacity of the cut* $\delta(S)$ induced by $S$ is equal to:

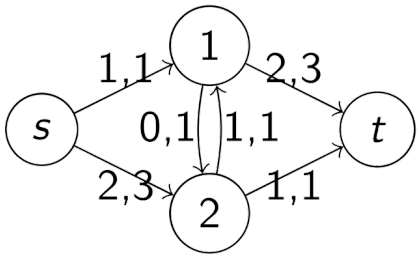
$$k(S) = \sum_{(i,j)\in\delta^+(S)} k_{ij}$$

Given a feasible flow $x$ from $s$ to $t$ and a cut $\delta(S)$ separating $s$ from $t$, the *value of the feasible flow x through the cut* $\delta(S)$ is:

$$\varphi(S) = \sum_{(i,j)\in\delta^+(S)} x_{ij} - \sum_{(i,j)\in\delta^-(S)} x_{ij}$$

With this notation the value of the flow $\boldsymbol{x}$ is $\varphi = \varphi(\{s\})$.

**Example:** The value of the feasible flow $\boldsymbol{x}$ through the cut $\delta(S)$ in the graph is:




$$\varphi(\{s,1\}) = 2 + 0 + 2 - 1 = 3$$

And the other values are:

$$\delta(\{s,1\}) = \{(s,2),(1,2),(1,t)\}$$

$$k(S) = 7$$

If $\varphi(S) = k(S)$ for a subset $S \subseteq V$ with $s \in S$ and $t \notin S$, then $x$ is a flow of maximum value and the cut $\delta(S)$ is of minimum capacity. The property $\varphi(S) \leq k(S)$ for any feasible flow $x$ and for any cut $\delta(S)$ separating $s$ from $t$, expresses a weak duality relationship between the two problems:

- Given $G = (V, A)$ with integer capacities on the arcs and $s, t \in V$, determine a feasible flow of maximum value.
- Given $G = (V, A)$ with integer arc capacities and $s, t \in V$, determine a cut (separating $s$ from $t$) of minimum capacity.

The idea of the Ford-Fulkerson's algorithm to find the network flows is the following. Start from a feasible flow $\boldsymbol{x}$ and try to iteratively increase its value $\varphi$ by sending, at each iteration, an additional amount of product along an undirected path from $s$ to $t$ with a strictly positive residual capacity.

If the arc $(i, j)$ is not saturated we can increase $x_{ij}$. If $(i, j)$ is not empty we can decrease $x_{ij}$ while respecting $0 \leq x_{ij} \leq k_{ij}$.

**Definition**

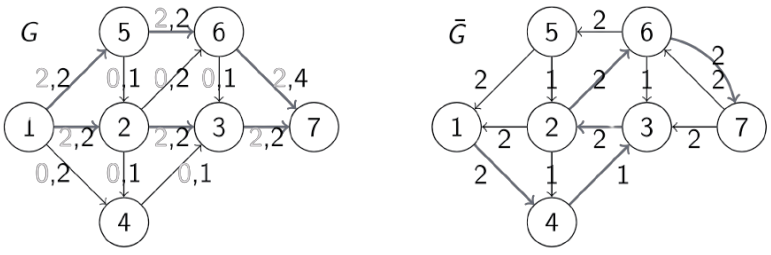
A path $P$ from $s$ to $t$ is an *augmenting path* with respect to the current feasible flow $x$ if $x_{ij} < k_{ij}$ for any forward arc $x_{ij} > 0$ for any backward arc.

Given a feasible flow $x$ for $G = (V, A)$, we construct the residual network $\overline{G} = (V, \overline{A})$ associated to $\boldsymbol{x}$, which accounts for all possible variations of $\boldsymbol{x}$:

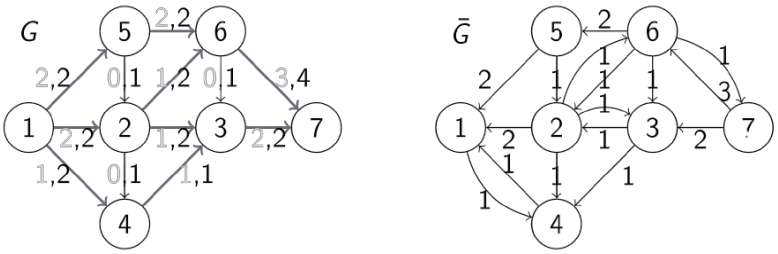
- If $(i, j) \in A$ is not empty $(i, j) \in \overline{A}$ with $\overline{k}_{ij} = x_{ij} > 0$.
- If $(i, j) \in A$ is not saturated $(i, j) \in \overline{A}$ with $\overline{k}_{ij} = k_{ij} - x_{ij} > 0$

where $\overline{k}_{ij}$ is called the residual capacity.

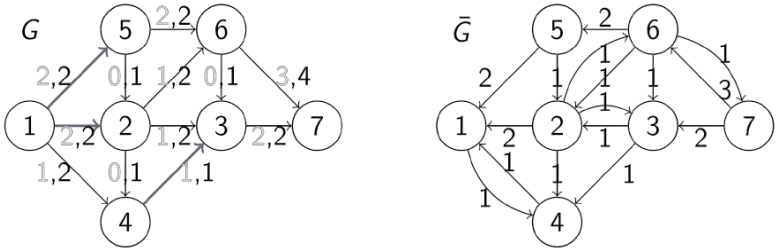
**Example :** The residual capacity of the following graph is computed in this way:




Then we update the flow, and we calculate again the residual capacity.




We do the same process again.




We cannot find other paths, so we now have obtained that the flow is $\phi = 5$.

**Proposition**

Ford-Fulkerson's algorithm is exact.

**Proof:** A feasible flow $\boldsymbol{x}$ has a maximum value if and only if $t$ is not reachable from $s$ in the residual network associated to $\boldsymbol{x}$. If there is an augmenting path, then $\boldsymbol{x}$ is not optimal for maximum value. If $t$ is not reachable from $s$, then there is a cut of $\overline{G}$ such that $\delta^+_{\overline{G}}(S^*) = \varnothing$. By definition of $\overline{G}$ we have that every $(i,j) \in \delta^+_G(S^*)$ is saturated and every $\delta^-_G(S^*)$ is empty. Therefore,

$$\phi(S^*) = \sum_{(i,j)\in\delta^+_G(S^*)} x_{ij} - \sum_{(i,j)\in\delta^-_G(S^*)} x_{ij} = \sum_{(i,j)\in\delta^+_G(S^*)} k_{ij} = k(S^*)$$

By weak duality, $\phi(S) < k(S)$, $\forall \boldsymbol{x}$ feasible, $\forall S \subset V$ with $s \in S$, $t \notin S$. Then the flow $\boldsymbol{x}$ has maximum value and the cut induced by $S^*$ minimum capacity. ■

**Theorem (*Strong duality*)**

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

The inputs of the algorithm are: a graph $G = (V, A)$ with capacity $k_{ij} > 0$ for any $(i, j) \in A$ such that $t \in N$.

**Algorithm 7** Ford-Fulkerson's algorithm

```
1: x ← 0
2: φ ← 0
3: optimum ← false
4: while optimum = true do
5:     Build residual network G̅ associated to x
6:     P ← path from s to t in G̅
7:     if P is not defined then
8:         optimum ← true
9:     else
10:        δ ← min{k̅_ij | (i, j) ∈ P}
11:        φ ← φ + δ
12:        for (i, j) ∈ P do
13:            if (i, j) is a forward arc then
14:                x_ij ← x_ij + δ
15:            else
16:                x_ij ← x_ij − δ
17:            end if
18:        end for
19:    end if
20: end while
```

The overall complexity of this algorithm is $O(m^2 k_{max})$. The space complexity of the algorithm is $O(m \log k_{max})$.

The algorithm can be made polynomial by looking for augmenting path with a minimum number of arcs.

### 3.5.1 Minimum cost flow problem

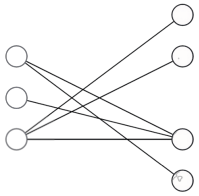
Given a network with a unit cost $c_{ij}$ associated to each arc $(i, j)$ and a value $\phi > 0$, determine a feasible flow from $s$ to $t$ of value $\phi$ and of minimum total cost.

The idea to solve this problem is to start from a feasible flow $x$ of value $\phi$ and send, an additional amount of product in the residual network along cycles of negative cost.

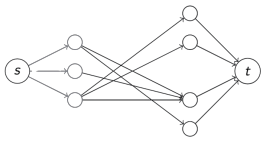
### 3.5.2 Assignment problem

**Definition**

Given an undirected bipartite graph $G = (V, E)$, a *matching* $M \subseteq E$ is a subset of non-adjacent edges.



Given a bipartite graph $G = (V, E)$, determine a matching with a maximum number of edges. This problem can be reduced to the problem of finding a feasible flow of maximum value from $s$ to $t$ in the following network.




There is a correspondence between the feasible flow of value $\varphi$ and the matching containing $\varphi$ edges.

## 3.6 Traveling salesman problem

Given a directed graph $G = (N, A)$ with cost $c_{ij} \in \mathbb{Z}$ for each are $(i, j) \in A$, determine a circuit of minimum total cost visiting every node exactly once.

**Definition**

A *Hamiltonian circuit* $C$ of $G$ is a circuit that visits every node exactly once.

So, denoting by $H$ the set of all Hamiltonian circuits of $G$, the problem amounts to:

$$\min_{C \in H} \sum_{(i,j) \in C} c_{ij}$$

This problem is $\mathcal{NP}$-hard.

# Chapter 4

# Linear programming

## 4.1 Introduction

**Definition**

A *linear programming problem* is an optimization problem:

$$\begin{aligned} &\min \ f(\boldsymbol{x}) \\ &\text{such that } \ \boldsymbol{x} \in \boldsymbol{X} \subseteq \mathbb{R}^n \to \mathbb{R} \end{aligned}$$

Where:

- The objective function $f : \boldsymbol{X} \to \mathbb{R}$ is linear.
- The feasible region $\boldsymbol{X} = \{\boldsymbol{x} \in \mathbb{R}^n | g_i(\boldsymbol{x}) r_i 0 \wedge i \in (1, m)\}$ with $r_i \in \{=, \geq, \leq\}$ and $g_i : \mathbb{R}^n \to \mathbb{R}$ linear functions, for $i = 1, \ldots, m$.

$\boldsymbol{x}^* \in \mathbb{R}^n$ is an *optimal solution* of the LP(1) if $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}), \forall \boldsymbol{x} \in \boldsymbol{X}$.

The general form of a linear programming problem is the following:

$$\min z = \boldsymbol{c}^T \boldsymbol{x}$$

such that $A\boldsymbol{x} \geq \boldsymbol{b}$ and $\boldsymbol{x} \geq \boldsymbol{0}$. In this notation we have that $\boldsymbol{x}$ is the vector of the decision variables. Note that the inequality for the matrix $A$ can be changed to an equality, and the constraints on the variable values can be removed in some problem. It is useful to transform all the linear programming problem in the general form found before. To do so we can use the following transformation rules:

- $\max(\boldsymbol{c}^T \boldsymbol{x}) = \min(-\boldsymbol{c}^T \boldsymbol{x})$

- $\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b} \implies \begin{cases} \boldsymbol{a}^T\boldsymbol{x} + s = \boldsymbol{b} \\ s \geq 0 \end{cases}$ where $s$ is called slack variable.
- $\boldsymbol{a}^T\boldsymbol{x} \geq \boldsymbol{b} \implies \begin{cases} \boldsymbol{a}^T\boldsymbol{x} - s = \boldsymbol{b} \\ s \geq 0 \end{cases}$ where $s$ is called surplus variable.
- If $x_{ij}$ is unrestricted in sign we have that $\begin{cases} x_j = x_j^+ - x_j^- \\ x_j = x_j^+, x_j^- \geq 0 \end{cases}$ After substituting $x_j$ with $x_j^+ - x_j^-$, we delete $x_j$ from the problem.
- $\boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b} \Leftrightarrow -\boldsymbol{a}^T\boldsymbol{x} \geq -\boldsymbol{b}$
- $\boldsymbol{a}^T\boldsymbol{x} \geq \boldsymbol{b} \Leftrightarrow -\boldsymbol{a}^T\boldsymbol{x} \leq -\boldsymbol{b}$
- $\boldsymbol{a}^T\boldsymbol{x} = \boldsymbol{b} \Leftrightarrow \begin{cases} \boldsymbol{a}^T\boldsymbol{x} \geq \boldsymbol{b} \\ \boldsymbol{a}^T\boldsymbol{x} \leq \boldsymbol{b} \end{cases}$

**Example:** Given the following linear programming model, rewrite it in the normal form:

$$\begin{aligned} \max f(x_1, x_2) = 2x_1 - 3x_2 \\ \text{such that } 4x_1 - 7x_2 \leq 5 \\ 6x_1 - 2x_2 \geq 4 \\ x_1 \geq 0, x_2 \in \mathbb{R} \end{aligned}$$

We can add two new variables $x_2 = x_3 - x_4$, with $x_3, x_4 \geq 0$, and we obtain:

$$\begin{aligned} \max f(x_1, x_2) = 2x_1 - 3x_2 \\ \text{such that } 4x_1 - 7x_3 + 7x_4 \leq 5 \\ 6x_1 - 2x_3 + 2x_4 \geq 4 \\ x_1, x_2, x_4 \geq 0 \end{aligned}$$

We now introduce slack and surplus variables $x_5$ and $x_6$, and we obtain:

$$\begin{aligned} \max f(x_1, x_2) = 2x_1 - 3x_2 \\ \text{such that } 4x_1 - 7x_3 + 7x_4 + x_5 = 5 \\ 6x_1 - 2x_3 + 2x_4 - x_6 = 4 \\ x_1, x_2, x_4, x_5, x_6 \geq 0 \end{aligned}$$

In the end we need to change the sign of the objective function:

$$\begin{aligned} \min f(x_1, x_2) = -2x_1 + 3x_2 \\ \text{such that } 4x_1 - 7x_3 + 7x_4 + x_5 = 5 \\ 6x_1 - 2x_3 + 2x_4 - x_6 = 4 \\ x_1, x_2, x_4, x_5, x_6 \geq 0 \end{aligned}$$

The assumptions of the linear programming models are:

1. Linearity of the objective function and constraints.
2. Proportionality of the objective function and constraints: the contribution of each variable is weighted on a constant.
3. Additivity of the objective function and constraints: the contribution of all variables is the sum of the single contributions.
4. Divisibility: the variables can take rational values.
5. Parameters are considered as constant which can be estimated with a sufficient degree of accuracy.

The linear programming sensitivity analysis allows evaluating how sensitive an optimal solution is with respect to small changes in the parameter values.

**Definition**

A *level curve* of a value $z$ of a function $f$ is the set of points in $\mathbb{R}^n$ where $f$ is constant and takes value $z$.

A *hyperplane* is defined as $H = \{\boldsymbol{x} \in \mathbb{R}^n | \boldsymbol{a}^T \boldsymbol{x} = \boldsymbol{b}\}$.

An *affine half-space* is defined as $H = \{\boldsymbol{x} \in \mathbb{R}^n | \boldsymbol{a}^T \boldsymbol{x} \leq \boldsymbol{b}\}$.

Each inequality constraint defines an affine half-space in the variable space.

**Definition**

The feasible region $\boldsymbol{X}$ of any linear programming is a *polyhedron* $P$.

A subset $S \subseteq \mathbb{R}^n$ is *convex* if for each pair $y_1, y_2 \in S$, $S$ contains the whole segment connecting $y_1$ and $y_2$.

The segment defined by $y_1, y_2 \in S$, defined by all *convex combinations* of $y_1$ and $y_2$ is:

$$[y_1, y_2] = \{\boldsymbol{x} \in \mathbb{R}^n | \boldsymbol{x} = \alpha y_1 + (1 - \alpha) y_2 \wedge \alpha \in [0, 1]\}$$

**Property :** A polyhedron $P$ is a convex set of $\mathbb{R}^n$

This happens because every half-space is convex, and the intersection of a finite number of convex sets is also a convex set.

**Definition**

A *vertex* of $P$ is a point $P$ which cannot be expressed as a convex combination of two other distinct points of $P$.

Algebraically, a vertex is defined as:

$$x = \alpha y_1 + (1-\alpha)y_2, \alpha \in [0,1], y_1, y_2 \in P \implies x = y_1 \vee x = y_2$$

**Property :** A non-empty polyhedron $P = \{x \in \mathbb{R}^n | A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}\}$ (in standard form) or $P = \{\boldsymbol{x} \in \mathbb{R}^n | A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \geq \boldsymbol{0}\}$ (in canonical form) has a finite number ($n \geq 1$) of vertices.

**Definition**

Given a problem $P$, a vector $\boldsymbol{d} \in \mathbb{R}^n$ with $\boldsymbol{d} \neq \boldsymbol{0}$ is an *unbounded feasible direction of* $P$ if, for every point $x_0 \in P$, the ray $\{\boldsymbol{x} \in \mathbb{R}^n | \boldsymbol{x} = \boldsymbol{x}_0 + \lambda \boldsymbol{d}, \lambda \geq 0\}$ is contained in $P$.