

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

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 1.1 | Definition | 2 |
| 1.2 | Decision-making problems | 2 |
| 1.3 | History | 3 |
| 1.4 | Operations Research workflow | 3 |
| 1.5 | Mathematical programming problem | 6 |
| 1.5.1 | Multi-objective programming | 7 |
| 2 | Algorithms | 8 |
| 2.1 | Complexity | 8 |
| 2.2 | Definitions | 9 |
| 2.3 | Dynamic programming | 9 |
| 3 | Network optimization models | 11 |
| 3.1 | Introduction | 11 |
| 3.2 | Graph reachability problem | 15 |
| 3.3 | Minimum spanning tree problem | 16 |
| 3.4 | Graph shortest path problem | 18 |
| 3.4.1 | Dijkstra's algorithm | 18 |
| 3.4.2 | Floyd-Warshall's algorithm | 21 |
| 3.4.3 | Topological order algorithm | 23 |
| 3.4.4 | DAGs' dynamic programming algorithm | 24 |
| 3.4.5 | Project planning algorithm | 26 |
| 3.5 | Minimum network flow problem | 29 |
| 3.5.1 | Minimum cost flow problem | 33 |
| 3.5.2 | Assignment problem | 34 |
| 3.6 | Traveling salesman problem | 34 |
| 4 | Linear programming | 35 |
| 4.1 | Introduction | 35 |
| 4.2 | Geometry of linear programming | 37 |
| 4.3 | Basic feasible solutions | 39 |
| 4.4 | Simplex method | 42 |
| 4.4.1 | Optimality test | 42 |
| 4.4.2 | Selection of the adjacent vertex | 43 |

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.

This interdisciplinary discipline bridges the domains of applied mathematics, computer science, economics, and industrial engineering.

1.2 Decision-making problems

Definition

Decision-making problems entail the challenge of selecting a viable solution from an array of alternatives, guided by one or multiple criteria.

For more intricate decision-making problems, a mathematical modeling approach is employed. These problems can be categorized as follows:

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 on 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_{ij} , 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 directed 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

During World War II, groups of scientists were tasked with conducting research to determine the most effective methods for conducting military operations. In the years following the war, these techniques were declassified and began to be adopted on a broader scale, addressing various challenges in the realms of business, industry, and society. The post-war industrial expansion led to the growth of companies and organizations, which, in turn, gave rise to increasingly intricate decision-making dilemmas. This transformation was facilitated by the rapid advancements in Operations Research, numerical analysis methodologies, and the widespread adoption of computer technology.

1.4 Operations Research workflow



The fundamental stages involved in the examination of an Operations Research problem are as follows:

1. Problem definition: the initial step is to precisely articulate and understand the problem at hand.
2. Model construction: subsequently, a mathematical or computational model is constructed to encapsulate the problem's complexity.
3. Algorithm selection or development: to solve the model, an appropriate algorithm is chosen or developed, tailored to the specifics of the problem.
4. Implementation: the selected algorithm is either implemented or utilized within an existing software or program.

Upon completing this process, the results are thoroughly scrutinized, and feedback is integrated.

The resultant model, forged through this procedure, serves as a simplified representation of the real-world problem. To define it effectively, one must discern the fundamental components of the problem and delineate the principal 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 600 000 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 $j = 0, \dots, 8$.
- 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

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

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

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

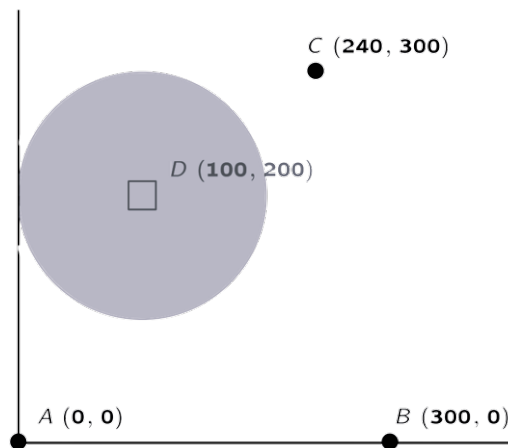
- Variable type: the variables are binary integer defined as $x_j \in \{0, 1\}$ $1 \leq j \leq 8$.

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:

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

- 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 |
|------------------------------------|-----------|-----------|-------------|
| <i>Mathematical programming</i> | single | one | no |
| <i>Multi-objective programming</i> | single | multiple | no |
| <i>Stochastic programming</i> | - | - | yes |
| <i>Game theory</i> | multiple | - | no |

We will delve into the realm of mathematical programming problems, where the primary aim typically revolves around minimizing or maximizing a specified function. Notably, it's worth mentioning that the maximization of a function $f(x)$ is essentially equivalent to the minimization of $-f(x)$. These problems are defined by the following characteristics:

- Decision variables $x \in \mathbb{R}^n$: these are numerical variables that serve as identifiers for potential solutions.
- Feasible region $X \subseteq \mathbb{R}^n$: this is the set of admissible values for the decision variables.
- Objective function $f : X \rightarrow \mathbb{R}$: this function quantitatively expresses the value of each feasible solution.

The core objective in solving a mathematical programming problem is to uncover a feasible solution that is globally optimal. In some cases, the problem may prove to be infeasible, unbounded, possess a unique optimal solution, or offer a multitude of optimal solutions. When dealing with particularly challenging problems, it may be necessary to settle for a feasible solution that represents a local optimum.

Mathematical programming can be categorized into the following classes:

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

1.5.1 Multi-objective programming

Multi-objective programming can be approached in various ways. Suppose our goal is to minimize $f_1(x)$ while simultaneously maximizing $f_2(x)$. In this context, we can:

1. Combine the objectives into a single objective problem by representing both objectives in the same units. This involves minimizing a weighted combination of the objectives, as follows:

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

with appropriate scalar values λ_1 and λ_2 .

2. Prioritize the primary objective and transform the other objective into a constraint. In this approach, the focus is on optimizing the primary objective function while ensuring that the secondary objective satisfies a specified constraint. This is achieved as follows:

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

with an appropriate constant value ε .

CHAPTER 2

Algorithms

2.1 Complexity

Definition

An *algorithm* is a step-by-step sequence of instructions designed to solve any given instance of a problem.

An *instance*, denoted as I , pertaining to a problem P , is a specific and unique case derived from the problem P .

The runtime of an algorithm is contingent on both the specific instance and the computer it runs on. To assess the algorithm's complexity while abstracting from hardware variations, we focus on evaluating its performance as a function of the instance's size, independently of the underlying hardware. For this purpose, we consider the count of elementary operations, presuming that each operation carries an equivalent cost. Since determining the precise count of elementary operations is often a formidable task, we resort to considering the asymptotic count of elementary operations in the worst-case scenario.

Definition

A function f is denoted as being *order of* g and expressed as:

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

If there exists a positive constant c such that $f(n)$ is less than or equal to $c \cdot g(n)$ for sufficiently large values of n .

We can categorize algorithms into two primary classes based on their worst-case complexity:

- Polynomial: characterized by a complexity of $O(n^d)$, where d is a constant.
- Exponential: demonstrating a complexity of $O(2^n)$.

Algorithms with a high-order polynomial complexity are generally considered inefficient in practical applications.

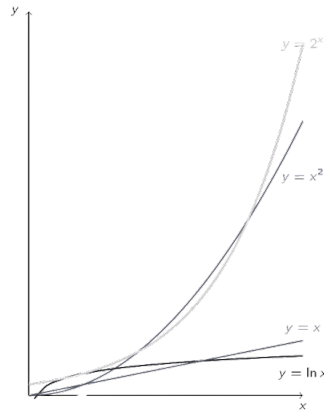


Figure 2.1: Plot of various algorithm's complexity

Definition

The size of an instance denoted as $|I|$ represents the number of bits required to represent that instance.

A problem P is considered *polynomially solvable* if there exists a polynomial time algorithm that can provide an optimal solution for any given instance.

For many discrete optimization problems, the most efficient algorithm available today demands a number of elementary operations that, in the worst-case scenario, grows exponentially with the size of the instance.

Definition

\mathcal{NP} -hard computational problems are, at the very least, as challenging as a broad spectrum of exceptionally difficult problems for which no polynomial time algorithm has been identified to date.

The \mathcal{NP} -hardness of a problem is a very strong evidence that is inherently difficult. However, this doesn't imply that it cannot potentially be solved using a polynomial time algorithm.

2.2 Definitions

Definition

An algorithm is considered *exact* when it is capable of delivering an optimal solution for every single instance.

In contrast, an algorithm is deemed *heuristic* when it is not guaranteed to provide an optimal solution for all instances.

A *greedy algorithm* progressively builds a feasible solution by consistently making locally optimal choices at each step, without revisiting or reevaluating prior selections.

2.3 Dynamic programming

Dynamic programming, introduced by Richard Bellman in 1953, is a versatile method employed to find optimal solutions consisting of a sequence of elementary decisions. This is achieved by resolving a series of recursive equations.

Dynamic programming is well-suited for a wide range of sequential decision problems as long as they adhere to the optimality property.

In contemporary applications, dynamic programming finds utility across various domains, including optimal control, equipment maintenance and replacement, and the selection of inspection points along a production line.

CHAPTER 3

Network optimization models

3.1 Introduction

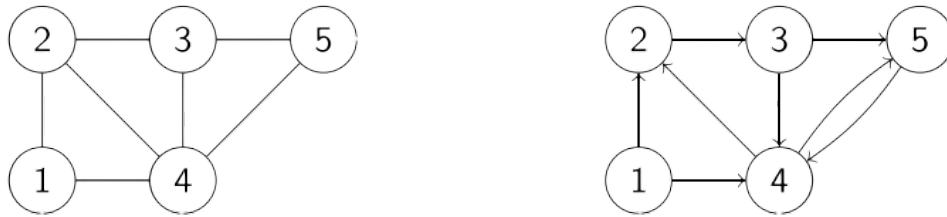
Numerous decision-making problems find their representation in the realm of graphs.

Definition

A graph is described as a pair $G = (N, E)$, comprising a set of nodes denoted as N and a set of edges or arcs E , where these edges link the nodes in pairs.

In the case of an undirected graph, an edge connecting nodes i and j is denoted as i, j , while in a directed graph, it is represented as (i, j) .

Example: Consider a road network connecting a total of "n" cities, which can be effectively represented by a graph. In this representation, each city corresponds to a node, and the connections between them are represented as edges.



The left graph is undirected and defined as follows:

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

The right graph 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

Nodes are considered *adjacent* when they are linked by an edge.

An edge labeled as e is deemed *incident* to a node v when node v serves as one of the endpoints of the edge e .

In undirected graphs, the *degree of a node* signifies the count of edges incident to that particular node.

In directed graphs, the *in-degree of a node* represents the number of arcs that have it as their successor, while the *out-degree of a node* denotes the quantity of arcs that have it as their predecessor.

Example: In the undirected graph, we observe that nodes 1 and 2 are adjacent, while node 1 and node 3 are not. The edge $\{1, 2\}$ is incident to nodes 1 and 2. Node 1 has a degree of 2, and node 4 has a degree of 4. In the directed graph, node 1 exhibits an in-degree of 0 and an out-degree of 2.



Definition

A *directed path* from $i \in N$ to $j \in N$ is a sequence of arcs $p = \langle \{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{k-1}, v_k\} \rangle$ 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 $N_1 \cap N_2 = \emptyset$ 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 $N_1 \cap N_2 = \emptyset$ 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 $m = \frac{n(n - 1)}{2}$ 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:

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

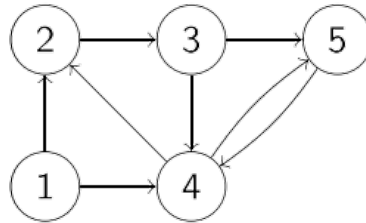
The best way to represent a sparse graph is by using lists of successors for each node.

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:

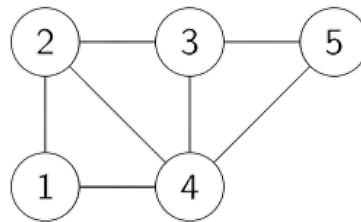
$$S(1) = \{2, 4\} \quad S(2) = \{3\} \quad S(3) = \{4, 5\} \quad S(4) = \{2, 5\} \quad S(5) = \{4\}$$



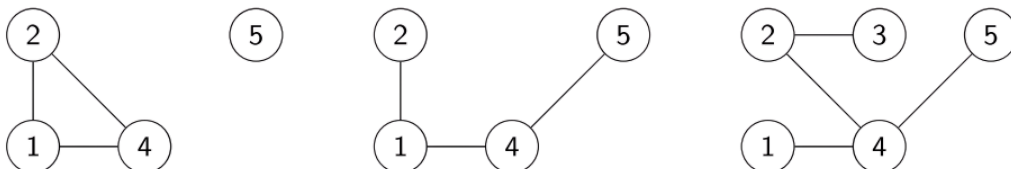
Definition

$G' = (N', E')$ 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:



From this graph, we can derive three distinct structures: a sub-graph, a tree, and a spanning tree. The modified graph below provides a visual representation of these structures:



Proposition (*Property*)

Every tree with n nodes has $n - 1$ edges.

Proof: 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. ■

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

```

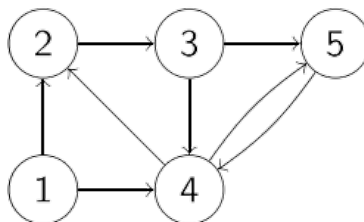
1:  $Q \leftarrow \{s\}$ 
2:  $M \leftarrow \{\emptyset\}$ 
3: while  $Q \neq \emptyset$  do
4:    $u \leftarrow \text{node in } Q$ 
5:    $Q \leftarrow Q - \{u\}$ 
6:    $M \leftarrow M \cup \{u\}$ 
7:   for  $(u, v) \in \delta^+(u)$  do
8:     if  $v \notin M$  and  $v \notin Q$  then
9:        $Q \leftarrow Q \cup \{v\}$ 
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. $Q = \{2\}$ $M = \emptyset$
2. $Q = \{3\}$ $M = \{2\}$
3. $Q = \{4, 5\}$ $M = \{2, 3\}$
4. $Q = \{5\}$ $M = \{2, 3, 4\}$
5. $Q = \emptyset$ $M = \{2, 3, 4, 5\}$

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 .

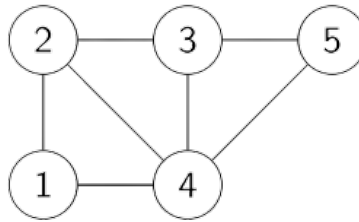
Theorem

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.

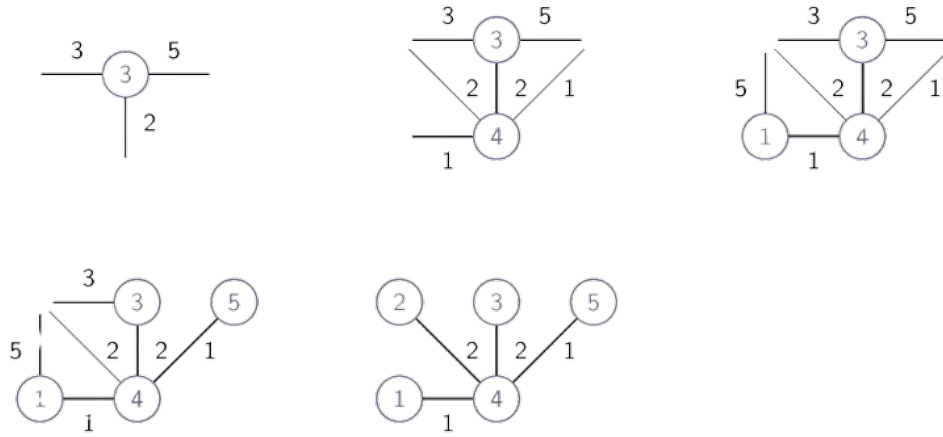
Example : Apply the Prim's algorithm to the following graph:



We select the node 3 as starting node, and so we have $S = \{3\}$ $T = \{\emptyset\}$, 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 $T \subseteq E$ 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:  $T \leftarrow \{\emptyset\}$ 
3: while  $|T| < n - 1$  do
4:    $\{u, v\} \leftarrow$  edge in  $\delta(S)$  of minimum cost
5:    $S \leftarrow S \cup \{v\}$ 
6:    $T \leftarrow T \cup \{u, v\}$ 
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

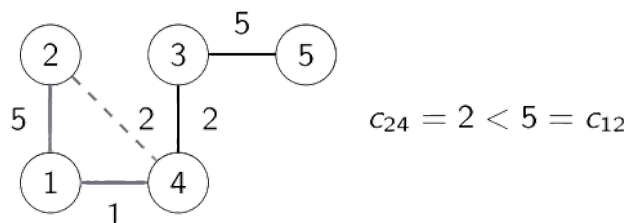
Prim's algorithm is greedy.

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



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

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 .

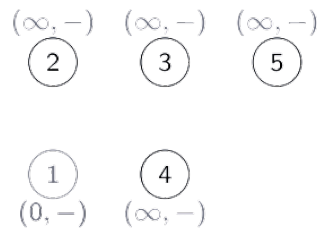
Each value c_j represents the cost of arc $(i, j) \in A$. Node s is the origin, or source, and t is the destination, or sink.

Definition

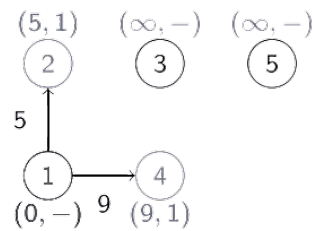
A path is *simple* if no node is visited more than once.

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 .

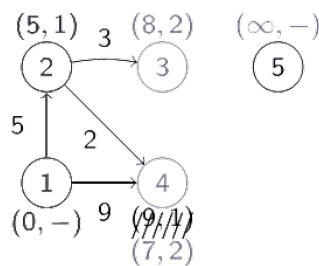
Example : Given a graph, and selecting 1 as the initial node, we set the following labels.



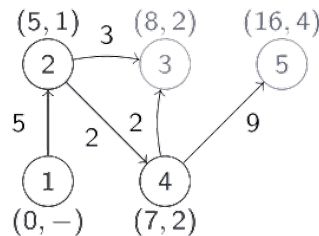
Next, we check all the arcs going from the starting node to other nodes, and we update the nodes label (the node one will always have no predecessor and null cost).



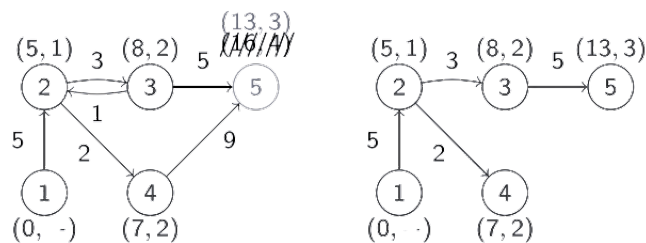
Now, we move to the node 2 and check all the reachable nodes. In this case we found the shortest path from the initial node to 4, so we update the label of 4.



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:  $S \leftarrow \emptyset$ 
2:  $X \leftarrow \{s\}$ 
3: for  $u \in N$  do
4:    $L_u \leftarrow +\infty$ 
5: end for
6:  $L_s \leftarrow 0$ 
7: while  $|S| \neq n$  do
8:    $u \leftarrow \operatorname{argmin}\{L_i | i \in X\}$ 
9:    $X \leftarrow X - \{u\}$ 
10:   $S \leftarrow S \cup \{u\}$ 
11:  for  $(u, v) \in \delta^+(u)$  such that  $L_v > L_u + c_{uv}$  do
12:     $L_v \leftarrow L_u + c_{uv}$ 
13:     $\text{pred}_v \leftarrow u$ 
14:     $X \leftarrow X \cup \{v\}$ 
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: $S = \{s, i_2, \dots, i_k\}$ 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 \emptyset 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 π from s to u has $c(\pi) \geq c(\emptyset)$. 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:

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

■

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:

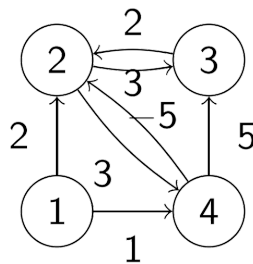
- d_{ij} , that is the cost of the shortest path from i to j .
- p_{ij} , that is the predecessor of j in that shortest path from i to j .

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:

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

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  $i \in N$  do
2:   for  $j \in N$  do
3:      $p_{ij} \leftarrow i$ 
4:      $d_{ij} \leftarrow \begin{cases} 0 & i = j \\ c_{ij} & i \neq j \wedge (i, j) \in A \\ +\infty & \text{otherwise} \end{cases}$ 
5:   end for
6: end for
7: for  $u \in N$  do
8:   for  $i \in N - \{u\}$  do
9:     for  $j \in N - \{u\}$  do
10:      if  $d_{iu} + d_{uj} < d_{ij}$  then
11:         $p_{ij} \leftarrow p_{uj}$ 
12:         $d_{ij} \leftarrow d_{iu} + d_{uj}$ 
13:      end if
14:    end for
15:  end for
16:  for  $i \in N$  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 $1, \dots, u$. ■

3.4.3 Topological order algorithm

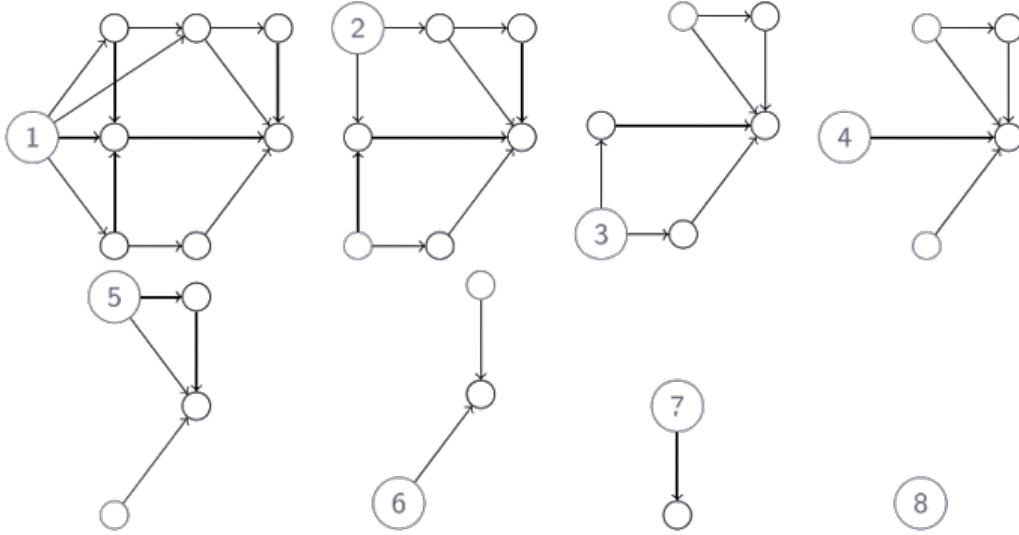
Given a directed acyclic graph $G = (N, A)$ with a cost $c_{ij} \in \mathbb{R}$ for each $(i, j) \in A$, and nodes s and t , determine the shortest (longest) path from s to t . The DAGs have the following property called topological order: the nodes of any directed acyclic graph G can be ordered topologically, 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 $\delta^-(v) = \emptyset$.
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 π_t , with at least 2 arcs, can be subdivided into two parts: π_t and (i, t) , where π_t is the shortest sub-path from s to i . For each node $i = 1, \dots, t$, 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 π_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 π_t .

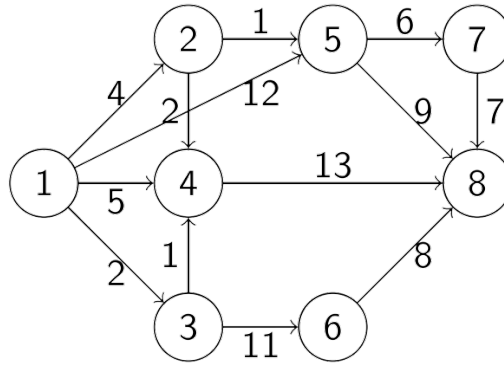
For DAGs whose nodes are topologically ordered L_{t-1}, \dots, L_1 satisfy the same type of recursive relations:

$$L_{t-1} = \min_{i < t-1} \{L_i + c_{i,t-1}\}; \dots; L_2 = \min_{i=1} \{L_i + c_{i2}\} = L_1 + c_{12}; L_1 = 0$$

which can be solved in the reverse order:

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

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** $j = 2, \dots, n$ **do**
 - 4: $L_j \leftarrow \min \{L_i + c_{ij} \mid (i, j) \in \delta^-(j) \wedge i < j\}$
 - 5: $\text{pred}_j \leftarrow v$ such that $(v, j) = \text{argmin} \{L_i + c_{ij} \mid (i, j) \in \delta^-(j) \wedge i < j\}$
 - 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:

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

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 , π_t , there exists $i < j$ such that the path can be subdivided into two parts: π_i and (i, t) , where π_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 $d_i \geq 0, i = 1, \dots, m$.

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 .

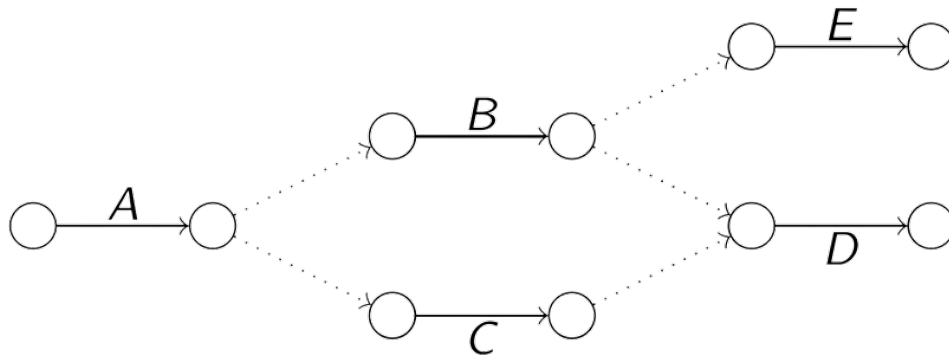
A project can be represented by a directed graph $G = (N, A)$ where each arc corresponds to an activity, and its arc length represents the duration of the corresponding activity. To account for the precedence constraints, the arcs must be positioned so that

$$A_i \propto A_j$$

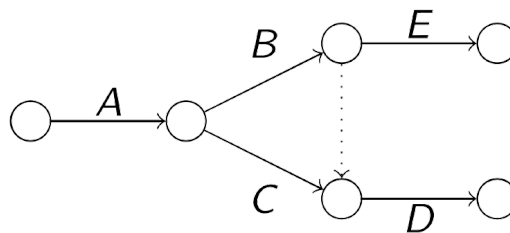
implies that there exists a directed path where the arc associated to A_i precedes the arc associated to A_j . Therefore, a node v marks an event corresponding to the end of all the activities $(i, v) \in \delta^-(v)$ and, hence, the possible beginning of all those $(v, j) \in \delta^+(v)$. We can also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

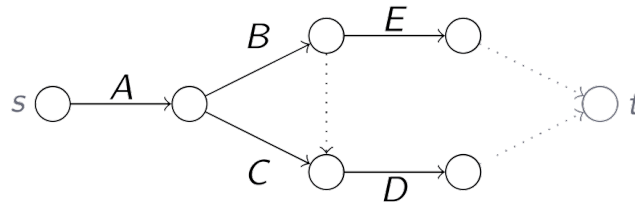
Example : A project can be defined in the following way:



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 we 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: $\sigma_{ij} = T_{max_j} - T_{max_i} - d_{ij}$.

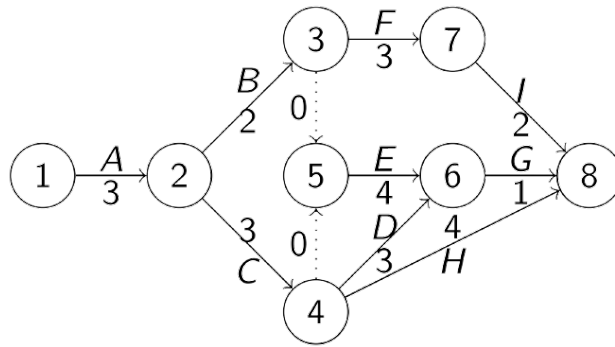
Example : Consider the following project:

| 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 |

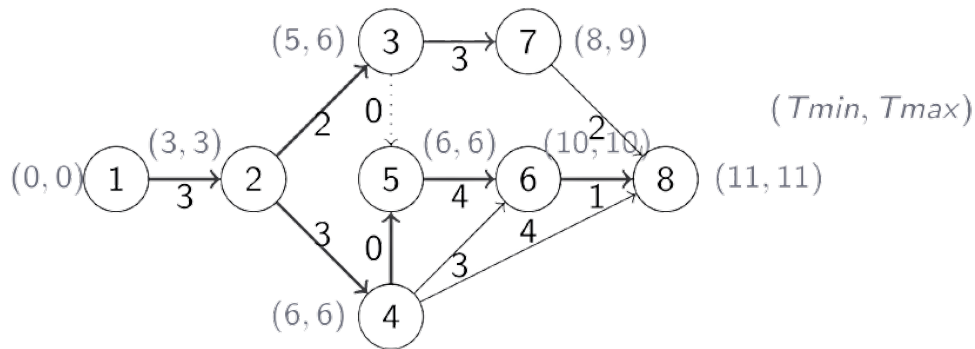
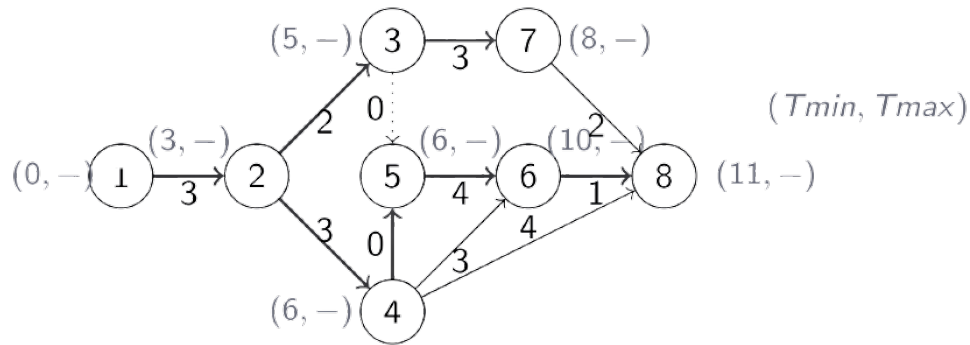
With these precedence constraints:

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

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** $j = 2, \dots, n$ **do**
 - 4: $T_{min_j} \leftarrow \max\{T_{min_i} + d_{ij} \mid (i, j) \in \delta^-(j)\}$
 - 5: **end for**
 - 6: $T_{max_n} \leftarrow T_{min_n}$
 - 7: **for** $i = n - 1, \dots, 1$ **do**
 - 8: $T_{max_i} \leftarrow \min\{T_{max_j} - d_{ij} \mid (i, j) \in \delta^+(i)\}$
 - 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*.

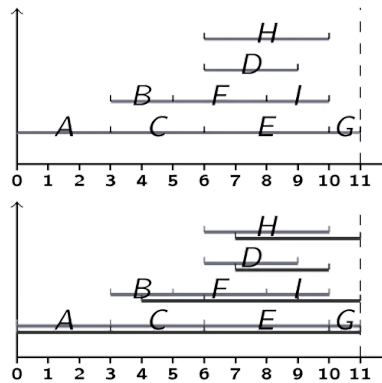
A *critical path* is an $s - t$ path only composed of critical activities (it always exists).

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* \mathbf{x} from s to t is a vector $\mathbf{x} \in \mathbb{R}^m$ 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 φ 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:

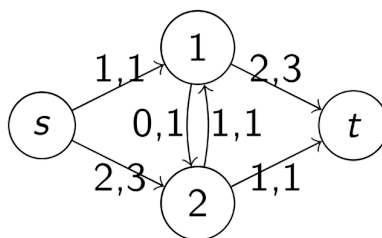
$$k(S) = \sum_{(i,j) \in \delta^+(S)} k_{ij}$$

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 \mathbf{x} is $\varphi = \varphi(\{s\})$.

Example: The value of the feasible flow \mathbf{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 \mathbf{x} and try to iteratively increase its value φ 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

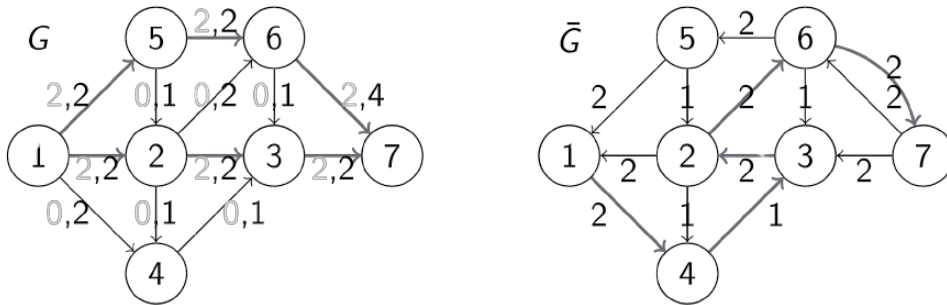
A path P from s to t is an *augmenting path* with respect to the current feasible flow x if $x_{ij} < k_{ij}$ for any forward arc $x_{ij} > 0$ for any backward arc.

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

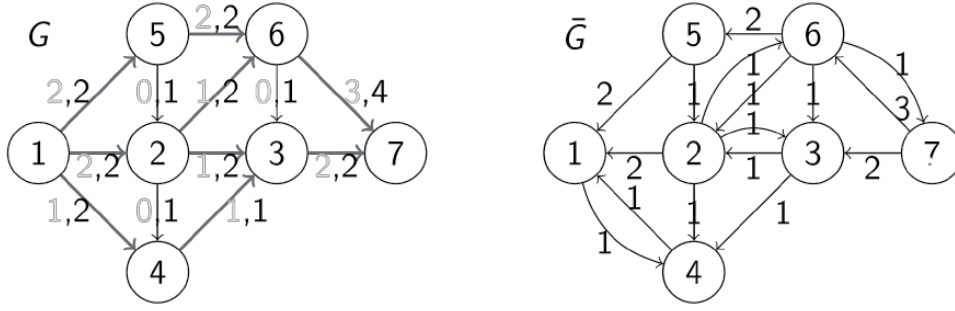
- If $(i, j) \in A$ is not empty $(i, j) \in \bar{A}$ with $\bar{k}_{ij} = x_{ij} > 0$.
- If $(i, j) \in A$ is not saturated $(i, j) \in \bar{A}$ with $\bar{k}_{ij} = k_{ij} - x_{ij} > 0$

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

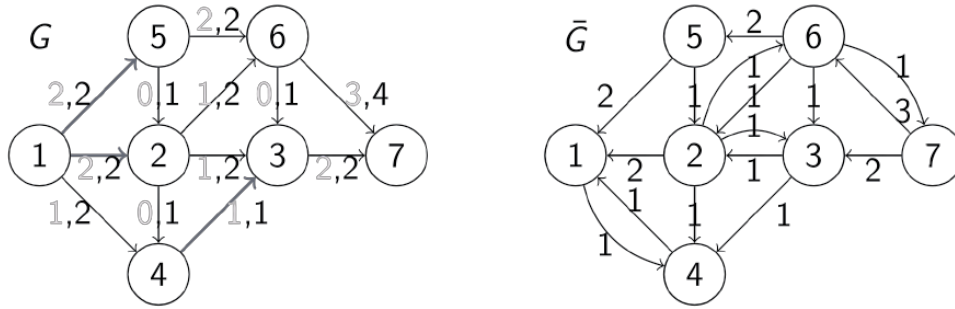
Example: The residual capacity of the following graph is computed in this way:



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 \mathbf{x} has a maximum value if and only if t is not reachable from s in the residual network associated to \mathbf{x} . If there is an augmenting path, then \mathbf{x} is not optimal for maximum value. If t is not reachable from s , then there is a cut of \bar{G} such that $\delta_G^+(S^*) = \emptyset$. By definition of \bar{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, $\phi(S) < k(S)$, $\forall \mathbf{x}$ feasible, $\forall S \subset V$ with $s \in S$, $t \notin S$. Then the flow \mathbf{x} has maximum value and the cut induced by S^* minimum capacity. ■

Theorem

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:  $x \leftarrow 0$ 
2:  $\phi \leftarrow 0$ 
3: optimum  $\leftarrow$  false
4: while optimum = true do
5:   Build residual network  $\overline{G}$  associated to  $x$ 
6:    $P \leftarrow$  path from  $s$  to  $t$  in  $\overline{G}$ 
7:   if  $P$  is not defined then
8:     optimum  $\leftarrow$  true
9:   else
10:     $\delta \leftarrow \min\{\overline{k}_{ij} | (i, j) \in P\}$ 
11:     $\phi \leftarrow \phi + \delta$ 
12:    for  $(i, j) \in P$  do
13:      if  $(i, j)$  is a forward arc then
14:         $x_{ij} \leftarrow x_{ij} + \delta$ 
15:      else
16:         $x_{ij} \leftarrow x_{ij} - \delta$ 
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

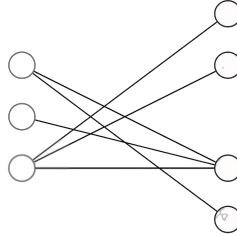
Given a network with a unit cost c_{ij} associated to each arc (i, j) and a value $\phi > 0$, determine a feasible flow from s to t of value ϕ and of minimum total cost.

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

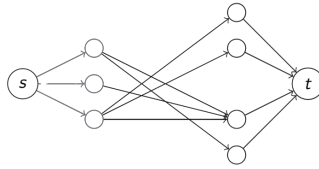
3.5.2 Assignment problem

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 φ and the matching containing φ edges.

3.6 Traveling salesman problem

Given a directed graph $G = (N, A)$ with cost $c_{ij} \in \mathbb{Z}$ for each arc $(i, j) \in A$, 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:

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

Where:

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

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

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

$$\min z = c^T x$$

such that $Ax \geq b$ and $x \geq 0$. In this notation we have that 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:

- $\max(c^T x) = \min(-c^T x)$
- $a^T x \leq b \implies \begin{cases} a^T x + s = b \\ s \geq 0 \end{cases}$ where s is called slack variable.
- $a^T x \geq b \implies \begin{cases} a^T x - s = b \\ s \geq 0 \end{cases}$ 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.
- $a^T x \leq b \Leftrightarrow -a^T x \geq -b$
- $a^T x \geq b \Leftrightarrow -a^T x \leq -b$
- $a^T x = b \Leftrightarrow \begin{cases} a^T x \geq b \\ a^T x \leq b \end{cases}$

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.

4.2 Geometry of linear programming

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 $H = \{x \in \mathbb{R}^n | a^T x = b\}$.

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

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

Definition

The feasible region 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:

$$[y_1, y_2] = \{x \in \mathbb{R}^n | x = \alpha y^1 + (1 - \alpha)y^2 \wedge \alpha \in [0, 1]\}$$

Property 4.1. 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 4.2. A non-empty polyhedron $P = \{x \in \mathbb{R}^n | Ax = b, x \geq 0\}$ (in standard form) or $P = \{x \in \mathbb{R}^n | Ax = b, x \geq 0\}$ (in canonical form) has a finite number ($n \geq 1$) of vertices.

Definition

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

Theorem

Every point x of a polyhedron P can be expressed as a convex combination of its vertices x^1, \dots, x^k plus (if needed) an unbounded feasible direction d of P :

$$x = \alpha_1 x^1 + \dots + \alpha_k x^k + d$$

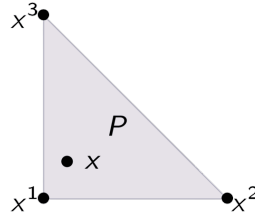
where the multipliers $\alpha_i \geq 0$ satisfies $\alpha_1 + \dots + \alpha_k = 1$.

Definition

A *polytope* is a bounded polyhedron, that is, it has the only unbounded feasible direction $d = 0$.

Every point x of a polytope P can be expressed as a convex combination of its vertices.

Example: The following point $x = \alpha_1 x^1 + \alpha_2 x^2 + \alpha_3 x^3$ can be also represented as $\alpha_1 + \alpha_2 + \alpha_3 = 1$ ($d = 0$).



The following theorem is known as the fundamental theorem of linear programming.

Theorem

Consider a linear programming problem $\min\{c^T x | x \in P\}$, where $P \subseteq \mathbb{R}^n$ is a non-empty polyhedron of the feasible solutions (in standard or canonical form). Then either there exists (at least) one optimal vertex or the value of the objective function is unbounded below on P .

Proof of case one: P has an unbounded feasible direction d such that $c^T d < 0$. P is unbounded and the values $z = c^T x$ tend to $-\infty$ along the direction d . ■

Proof of case two: P has no unbounded feasible direction d such that $c^T d < 0$, that is, for all of them we have $c^T d \geq 0$. Any point of P can be expressed as:

$$x = \sum_{i=1}^k \alpha_i x^i + d$$

where x^1, \dots, x^k are the vertices of P , $\alpha_i > 0$ with $\alpha_1 + \dots + \alpha_k = 1$, and $d = 0$, or d is an unbounded feasible direction. For any $x \in P$, we have $d = 0$ or $c^T d > 0$ and hence:

$$c^T x = c^T \left(\sum_{i=1}^k \alpha_i x^i + d \right) = \sum_{i=1}^k \alpha_i c^T x^i + c^T d \geq \min_{i=1, \dots, k} c^T x^i$$

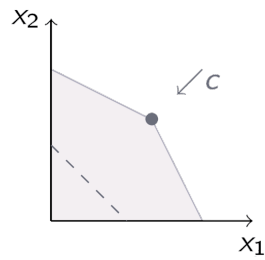
since $\alpha_i > 0$, for any i , and $\alpha_1 + \dots + \alpha_k = 1$. ■

From this theorem we can derive that an interior point $x \in P$ cannot be an optimal solution and that in an optimal vertex all feasible directions are worsening directions.

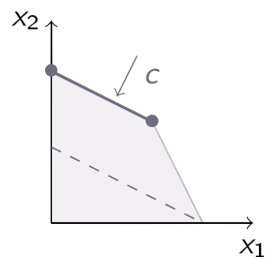
The theorem implies that, although the variables can take fractional values, linear programming can be viewed as combinatorial problems. Thus, we need to examine the vertices of the polyhedron of the feasible solutions. However, these are finite but often exponential and the graphical method is only applicable for $n \leq 3$.

The linear programs can be classified in four types:

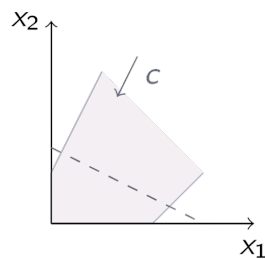
- A unique optimal solution



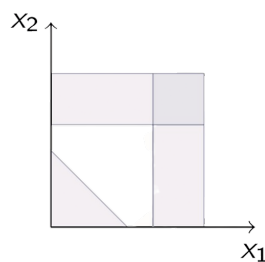
- Multiple optimal solutions



- Unbounded linear program



- Infeasible linear program



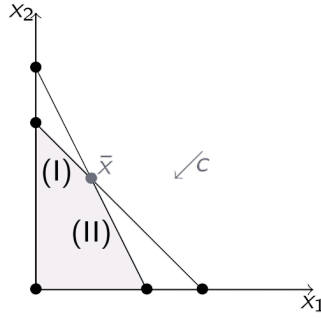
4.3 Basic feasible solutions

Due to the fundamental theorem of Linear Programming, to solve any linear program it suffices to consider the vertices of the polyhedron P of the feasible solutions. Since the geometrical definition of vertex cannot be exploited algorithmically, we need an algebraic characterization.

Example : The following linear program:

$$\begin{aligned} \min \quad & -x_1 - 3x_2 \\ \text{such that} \quad & x_1 + x_2 \leq 6 \\ & 2x_1 + x_2 \leq 8 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Its graphical representation is the following:



It is possible to see that a vertex corresponds to the intersection of the hyperplanes associated to n inequalities. In this example we have that the vertex \bar{x} is the intersection of the first two inequalities, that is the solution of the following system.

$$\begin{cases} x_1 + x_2 = 6 \\ 2x_1 + x_2 = 8 \end{cases}$$

If we want to use the same problem in standard form we have the following formulation:

$$\begin{aligned} \min \quad & -x_1 - 3x_2 \\ \text{such that} \quad & x_1 + x_2 + s_1 = 6 \\ & 2x_1 + x_2 + s_2 = 8 \\ & x_1, x_2, s_1, s_2 \geq 0 \end{aligned}$$

And to solve it we simply let $s_1 = s_2 = 0$ and solve the system found before. Using this formulation is also possible to compute all the intersections with the axis:

1. $x_1 = 0, x_2 = 0 \rightarrow s_1 = 6, s_2 = 8$
2. $x_1 = 0, s_1 = 0 \rightarrow x_2 = 6, s_2 = 2$
3. $x_1 = 0, s_2 = 0 \rightarrow x_2 = 8, s_1 = -2$
4. $x_2 = 0, s_1 = 0 \rightarrow x_1 = 6, s_2 = -4$
5. $x_2 = 0, s_2 = 0 \rightarrow x_1 = 4, s_1 = 2$
6. $s_1 = 0, s_2 = 0 \rightarrow x_1 = 2, x_2 = 4$

Note that the solutions three and four are infeasible since the values of s_1 and s_2 are negative.

Property 4.3. For any polyhedron $P = \{x \in \mathbb{R}^n | Ax = b, x \geq 0\}$:

- The facets (edges in \mathbb{R}^2) are obtained by setting one variable to 0
- The vertices are obtained by setting $n - m$ variables to 0.

Given a matrix $A \in \mathbb{R}^{m \times n}$ where $m \leq n$ we have that:

- If $m = n$, there is a unique solution of $Ax = b$.
- If $m < n$, there are ∞ solutions of $Ax = b$: the system has $n - m$ degrees of freedom. This means that $n - m$ variables can be fixed arbitrarily.

Definition

A *basis* of a matrix $A \in \mathbb{R}^{m \times n}$ with n variables and m constraints is a subset of m columns of A that are linearly independent and form an $m \times m$ non-singular matrix B .

Given the previous definition, we can permute the columns of A and then partition it to find the basis:

$$A = [B|N]$$

where B is a matrix $m \times m$ and N is a matrix $(n - m) \times m$.

Let $x^T = [x_B^T | x_N^T]$, where x_B^T has m components and x_N^T has $n - m$ components, then any system $Ax = b$ can be written as $Bx_B + Nx_N = b$ and for any set of values for x_N , if B is non-singular, we have:

$$x_B = B^{-1}b - B^{-1}Nx_N$$

Definition

A *basic solution* is a solution obtained by setting $x_N = 0$ and, consequently, letting $x_B = B^{-1}b$.

A basic solution with $x_B \geq 0$ is a *basic feasible solution*.

The variables in x_B are the *basic variables*.

The variables in x_N are the *non-basic variables*.

Theorem

$x \in \mathbb{R}^n$ is a basic feasible solution if and only if x is a vertex of:

$$P = \{x \in \mathbb{R}^n | Ax = b, x \geq 0\}$$

Example : Given the following linear program:

$$\begin{aligned} \min \quad & 2x_1 + x_2 + 5x_3 \\ \text{such that} \quad & x_1 + x_2 + x_3 + x_4 = 4 \\ & x_1 + x_5 = 2 \\ & x_3 + x_6 = 3 \\ & 3x_2 + x_3 + x_7 = 6 \\ & x_1, x_2, x_3, x_4, x_5, x_6, x_7 \geq 0 \end{aligned}$$

The matrices associated with the given constraints are:

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 3 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 4 \\ 2 \\ 3 \\ 6 \end{bmatrix}$$

We can see that the columns 4,5,6, and 7 are linearly independent. By choosing these three column we have

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = I = B^{-1}$$

The solution found in this case is $x_B = B^{-1}b = b$, that is a feasible solution. Instead, if we choose the columns 2,5,6, and 7 we have that the solution is $x_B = [4 \ 2 \ 3 \ -6]^T$, that is an infeasible solution.

The total number of feasible solution is computed as:

$$\text{number of feasible solutions} = \binom{n}{m}$$

4.4 Simplex method

The idea of the simplex method is that, given a linear program in standard form

$$\begin{aligned} \min \quad & z = c^T x \\ \text{such that} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

we will examine a sequence of basic feasible solutions with non-increasing objective function values until an optimal solution is reached, or the linear program is found to be unbounded.

4.4.1 Optimality test

Since a basic feasible solution is such that $x_B = B^{-1}b$, $x_N = 0$ it is possible to express objective function in terms of only the non-basic variables:

$$c^T x = c_B^T B^{-1}b + (c_N^T - c_B^T B^{-1}N) x_N$$

Definition

$$\bar{c}^T = c^T - C_B^T B^{-1}A = [c^T - C_B^T B^{-1}B, c^T - C_B^T B^{-1}N]$$

is the *vector of reduced costs* with respect to the basis B.

\bar{c}_j represents the change in the objective function value if non-basic x_j would be increased from 0 to 1 while keeping all other non-basic variables to 0. The solution value changes by:

$$\Delta z = \theta \cdot \bar{c}_j$$

Consider a linear program $\min\{c^T x | Ax = b, x \geq 0\}$ and a feasible basis B.

Proposition

If $\bar{c}_N \geq 0$ then the basic feasible solution (x_B^T, x_N^T) , where $x_B = B^{-1}b \geq 0$ and $x_N = 0$, of cost $c_B^T B^{-1}b$ is a global optimum.

Proof: $\bar{c}_N \geq 0$ implies that:

$$c^T x = c_B^T B^{-1} b + \bar{c}_N^T x_N \geq c_B^T B^{-1} b \quad \forall x \geq 0, Ax = b$$

■

his optimality condition is sufficient but in general not necessary.

Example: Given the following linear program:

$$\begin{aligned} \min \quad & -x_1 - x_2 \\ \text{such that} \quad & x_1 - x_2 + s_1 = 1 \\ & x_1 + x_2 + s_2 = 3 \\ & x_1, x_2, s_1, s_2 \geq 0 \end{aligned}$$

The matrices associated with the given constraints are:

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

Consider the solution $x_B = (x_1, s_2) = (1, 2)$, $x_n = (x_2, s_1)$, and $z = -1$, we have:

$$B = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad N = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$$

Therefore, we have that:

$$c_B^T = c_N^T = [-1 \quad 0] \quad B^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

and the reduced costs for x_2, s_1 are:

$$\bar{c}_N^T = c_N^T - c_B^T B^{-1} N = [-2 \quad 0]$$

Since $\bar{c}_2 = -2 < 0$ increasing x_2 to 1 we improve the solution by -2 .

4.4.2 Selection of the adjacent vertex

When moving from the current vertex to an adjacent vertex, we substitute one column of B with one column of N .