Foundations of Operations Research - Notes - v0.5.1

260236

November 2024

Preface

Every theory section in these notes has been taken from the sources:

- Course slides. [2]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on foundations of operations research. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

Contents

Algorithms

1 Introduction

Definition 1

Operations Research (OR), often shortened to the initialism `OR`, is the branch of mathematics in which **mathematical models** and **quantitative methods** (e.g. optimization, game theory, simulation) are **used to analyze complex decision-making problems** and **find (near-)optimal solutions**.

The overall and primary *goal* is to *help make better decisions.*

OR can be seen as an interdisciplinary field at the intersection of applied mathematics, computer science, economics, and industrial engineering.

Operations research is often concerned with **determining the extreme values of some real-world objective**: the *maximum* (of profit, performance, or yield) or *minimum* (of loss, risk, or cost). Originating in military efforts before World War II, its techniques have grown to concern problems in a variety of industries. [4]

Its origins date back to World War II: teams of scientists were asked to research the most efficient way to conduct operations (e.g., to optimize the allocation of scarce resources).

In the decades after the war, the techniques became public and were applied more widely to problems in business, industry, and society.

During the industrial boom, the substantial increase in the size of firms and organizations led to more complex decision problems.

There are favorable circumstances: rapid progress in OR and in numerical analysis methods, and the advent and spread of computers (available computing power and widespread software).

1.1 Decision-making problems

Decision-making problems are analyzed using mathematical models and quantitative methods.

Definition 2

Decision-making problems are problems in which we must **choose** a (feasible) **solution among a large number of alternatives based on one or several criteria**.

In other words, they are complex decision-making problems that are **addressed through a mathematical modeling approach** (mathematical models, algorithms, and computer implementations).

Some practical **examples** include assignment problem, network design, shortest paths, personnel scheduling, service management, multicriteria problem, and maximum clique.

1.1.1 Assignment problem

A mathematical definition of an **assignment problem** is as follows. 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.

	M_1	M_2	M_3
J_1	2	6	3
J_2	8	4	9
J_3	5	7	8

Table 1: Example of an assignment problem table.

The **main goal** is to **decide which job to assign to each machine in order to minimize the total execution time**. Also, (constraints) each job must be assigned to exactly one machine, and each machine must be assigned to exactly one job.

The **number** of feasible **solutions** is the permutations, then the **factorial of** m: $m!$.

1.1.2 Network design

The network design problem is characterized as **how to connect n cities** (offices) **via a collection of possible links** so as (*main goal*) **to minimize the total link cost**.

Using mathematical terms, 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.

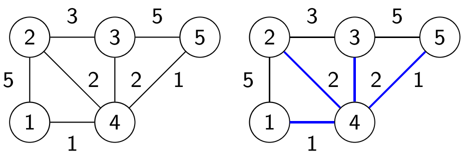


Figure 1: Examples of network design graphs.

The **number** of alternative **solutions** is at most 2^m, where $m = |E|$.

1.1.3 Shortest path

The shortest path problem is similar to network design. Given a direct path that represents a road network with distances (traveling times) for each arc, determine the shortest (fastest) path between two points (nodes).

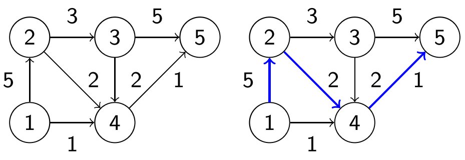


Figure 2: Examples of shortest path graphs.

1.1.4 Other problems

Other decision-making problems are:

- Personnel scheduling problem: determine the week schedule for the hospital personnel, so as to minimize the number of people involved (physicians, nurses, ...) while meeting the daily requirements.
- Service management problem: determine how many counters/desks to open at a given time of the day so that the average customer waiting time does not exceed a certain value (guarantee a given service quality).
- Multicriteria problem: decide which laptop to buy considering the price, the weight and the performance.
- Maximum clique problem: determine the complete subgraph of a graph, with maximum number of vertices.

1.2 Scheme of an OR study

The most important and common **steps** in operational research are:

1. **Problem**. Define the problem;
2. **Model**. Build the model;
3. **Algorithm**. Select or develop an appropriate algorithm;
4. **Implementation**. Implementing or using an efficient computer program;
5. **Results**. Analyze the results.

Definition 3

A mathematical **model** is a **simplified representation of a real-world problem**.

To define a mathematical model, it is necessary to identify the fundamental elements of the problem and the main relationships between them. But **how can we decide** the *number of decision makers*, the *number of objectives* and the *level of uncertainty in the parameters*? It depends on the environment. If we have:

- One decision maker, one object, then we will use **mathematical programming** (section 1.3, page 12).
- One decision maker, multiple objectives, then we will use **multi-objective programming** (section 1.4, page 15).
- Uncertainty greater than zero, then we will use **stochastic programming**.
- Multiple decision makers, then we will use **game theory**.

Example 1: production planning

A company produces 3 types of electronic devices: D_1, D_2, D_3; going through 3 main phases of the production process: assembly, refinement and quality control.
Time (in minutes) required for each phase and product:

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

Available resources within the planning horizon (depend on the workforce) in minutes:

Assembly	Refinement	Quality control
30′000	25′000	18′000

Unitary profit (in KEuro):

D_1	D_2	D_3
1.6	1	2

Assumption: the company can sell whatever it produces.
Give a mathematical model for determining a production plan which maximizes the total profit.

- **Decision variables**, x_j is equal to the number of devices D_j produced, for $j = 1, 2, 3$.
- **Objective function**: max $z = 1.6x_1 + 1x_2 + 2x_3$.
- **Constraints**: the production capacity limit for each phase:

$$\begin{array}{rcll} 80x_1 + 70x_2 + 120x_3 & \leq & 30'000 & \text{(assembly)} \\ 70x_1 + 90x_2 + 20x_3 & \leq & 25'000 & \text{(refinement)} \\ 40x_1 + 30x_2 + 20x_3 & \leq & 18'000 & \text{(quality control)} \end{array}$$

- **Non-negative variables**: $x_1, x_2, x_3 \geq 0$ may be fractional (real) values.

Example 2: portfolio selection problem

An insurance company must decide which investments to select out of a given set of possible assets (stocks, bonds, options, gold certificates, real estate, . . .).

Investments	area	capital (c_j $K€$)	expected return (r_j)
A	Germany	150	11%
B	Italy	150	9%
C	U.S.A.	60	13%
D	Italy	100	10%
E	Italy	125	8%
F	France	100	7%
G	Italy	50	3%
H	UK	80	5%

Legend:

- A and B: automotive
- C and D: ICT
- E and F: real estate
- G: short term treasury bounds
- H: long term treasury bounds

The available capital is: 600 KEuro.
At most 5 investments to avoid excessive fragmentation.
Geographic diversification to limit risk: ≤ 3 investments in Italy and ≤ 3 abroad.
Give a mathematical model for deciding which investments to select so as to maximize the expected return while satisfying the constraints.

- **Decision variables**, x_j is equal to 1 if j-th investment is selected and $x_j = 0$ otherwise, for $j = 1, \dots, 8$.
- **Objective function**: $\max z = \sum_{j=1}^{8} c_j\, r_j\, x_j$.
- **Constraints**:

$$\begin{array}{rcll} \sum_{j=1}^{8} c_j\, x_j & \leq & 600 & \text{(capital)} \\ \sum_{j=1}^{8} x_j & \leq & 5 & \text{(max 5 investments)} \\ x_2 + x_4 + x_5 + x_7 & \leq & 3 & \text{(max 3 in Italy)} \\ x_1 + x_3 + x_6 + x_8 & \leq & 3 & \text{(max 3 abroad)} \end{array}$$

- **Binary (integer) variables**: $x_j \in \{0, 1\}$ and $1 \leq j \leq 8$.

<u>**Possible variant**</u>. In order to limit the risk, if any of the ICT investment is selected then at least one of the treasury bond must be selected.

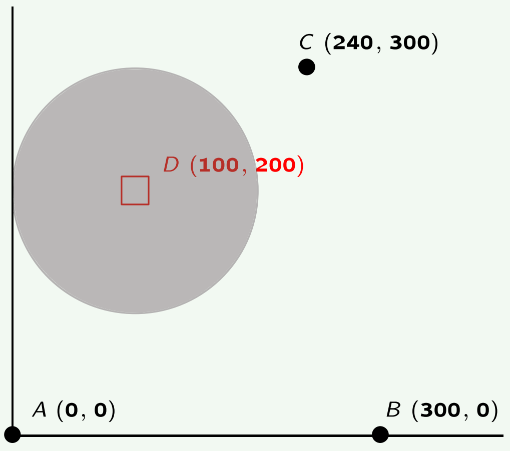
- **Objective function**: $\max z = \sum_{j=1}^{8} c_j\, r_j\, x_j$.
- **Constraints**:

$$\begin{array}{rcll} \sum_{j=1}^{8} c_j\, x_j & \leq & 600 & \text{(capital)} \\ \sum_{j=1}^{8} x_j & \leq & 5 & \text{(max 5 investments)} \\ x_2 + x_4 + x_5 + x_7 & \leq & 3 & \text{(max 3 in Italy)} \\ x_1 + x_3 + x_6 + x_8 & \leq & 3 & \text{(max 3 abroad)} \\ \frac{x_3 + x_4}{2} & \leq & x_7 + x_8 & \text{(investment in treasury bonds)} \end{array}$$

- **Binary (integer) variables**: $x_j \in \{0, 1\}$ and $1 \leq j \leq 8$.

Example 3: facility location

Consider 3 oil pits, located in positions A, B and C, 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 so as to minimize the total pipeline cost.

- **Decision variables**, x_1, x_2 cartesian coordinates of the refinery.
- **Objective function**:

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

- **Constraints**:

$$\sqrt{(x_1 - 100)^2 + (x_2 - 200)^2} \geq 100$$

- **Variables**: $x_1, x_2 \in \mathbb{R}$.

1.3 Mathematical programming/optimization

Mathematical Optimization or **Mathematical Programming** is the **selection of a best element**, with regard to some criteria, **from some set of available alternatives**.

In the more general approach, an optimization problem consists of **maximizing** or **minimizing a real function** by systematically choosing input values from within an allowed set and computing the value of the function. The generalization of optimization theory and techniques to other formulations constitutes a large area of applied mathematics.

Okay, how is it defined mathematically?

Mathematical Operation problems belong to the category of decision-making problems. They are characterized by a **single decision maker**, a **single objective**, and **reliable parameter estimates**. In mathematical language, we can say:

$$\text{opt } f(\mathbf{x}) \quad \text{with} \quad \mathbf{x} \in X \quad \text{and} \quad \text{opt } = \begin{Bmatrix} \min \\ \max \end{Bmatrix}$$

Where:

- $\mathbf{x} \in \mathbb{R}^n$ **decision variables**. They are numerical variables whose values identify a solution of the problem.
- $X \subseteq \mathbb{R}^n$ **feasible region**. Distinguishes between feasible and infeasible solutions (via constraints):

$$X = \left\{ \mathbf{x} \in \mathbb{R}^n \; : \; g_i(\mathbf{x}) \begin{Bmatrix} = \\ \leq \\ \geq \end{Bmatrix} 0, i = 1, \dots, m \right\}$$

- $f : X \to \mathbb{R}$ **objective function**.Expresses in quantitative terms the value or cost of each feasible solution.

Note an interesting observation:

$$\max \{ f(\mathbf{x}) : \; \mathbf{x} \in X \} = -\min \{ -f(\mathbf{x}) : \; \mathbf{x} \in X \}$$

More specifically, how can we solve these problems?

It depends on how hard the given problem is to solve.

- The problem has an **easy/medium level** of complexity. It makes sense to use the **Global Optima** technique. It consists in finding a feasible solution that is **globally optimum**, then a vector $\mathbf{x}^* \in X$ such that:

$$\begin{aligned} f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in X \quad \text{if opt } = \min \\ f(\mathbf{x}^*) \geq f(\mathbf{x}) \quad \forall \mathbf{x} \in X \quad \text{if opt } = \max \end{aligned}$$

  Unfortunately, this method is not perfect and it may happen that the given problem occurs:

- Is **infeasible**, so the feasible region is empty: $X = \emptyset$.
- Is **unbounded**: $\forall c \in \mathbb{R}$, $\exists \mathbf{x}_c \in X$ such that $f(\mathbf{x}_c) \le c$ or $f(\mathbf{x}_c) \ge c$.
- Has a **single optimal solution**.
- Has a **large number** (even an infinite number) **of optimal solutions** (with the same optimal value!).

- The problem has a **difficult/hard level** of complexity. Then the **Local Optima** is the best choice. It consists in finding a feasible solution that is **local optimum** (main different against global optima technique), then a vector $\widehat{\mathbf{x}} \in X$ such that:

$$\begin{array}{lll} f(\widehat{\mathbf{x}}) \le f(\mathbf{x}) & \forall \mathbf{x} \text{ with } \mathbf{x} \in X \text{ and } \|\mathbf{x} - \widehat{\mathbf{x}}\| \le \varepsilon & \text{if opt } = \min \\ f(\widehat{\mathbf{x}}) \ge f(\mathbf{x}) & \forall \mathbf{x} \text{ with } \mathbf{x} \in X \text{ and } \|\mathbf{x} - \widehat{\mathbf{x}}\| \le \varepsilon & \text{if opt } = \max \end{array}$$

  For an appropriate value $\varepsilon > 0$.

  In this case, it may happen that the given **problem has many local optima**.

Categories

A Mathematical Programming can be **categorized** depending on the feasible region:

- **Linear Programming (LP)**. The function f is linear:

$$X = \left\{ \mathbf{x} \in \mathbb{R}^n \ : \ g_i(\mathbf{x}) \begin{Bmatrix} = \\ \le \\ \ge \end{Bmatrix} 0, \ i = 1, \dots, m \right\} \text{ with } g_i \text{ linear } \forall i$$

  An **example** is the *production planning*.

- **Integer Linear Programming (ILP)**. The function f is linear:

$$X = \left\{ \mathbf{x} \in \mathbb{R}^n \ : \ g_i(\mathbf{x}) \begin{Bmatrix} = \\ \le \\ \ge \end{Bmatrix} 0, \ i = 1, \dots, m \right\} \cap \mathbb{Z}^n \text{ with } g_i \text{ linear } \forall i$$

  An **example** is the *portfolio selection* (finance). As we can see, the ILP technique is identical to LP with additional integrality constraints on the variables.

- **Nonlinear Programming (NLP)**. The function f is convex/regular or non convex/regular:

$$X = \left\{ \mathbf{x} \in \mathbb{R}^n \ : \ g_i(\mathbf{x}) \begin{Bmatrix} = \\ \le \\ \ge \end{Bmatrix} 0, \ i = 1, \dots, m \right\}$$

  With g_i convex/regular or not convex/regular $\forall i$.

  An **example** is the *facility location* (with g_i convex).

History of Mathematical Programming

It is correct to report the history of mathematical programming:

1826/27 Joseph Fourier presents a method to solve systems of linear inequalities (Fourier-Motzkin) and discusses some LPs with 2-3 variables.

1939 Leonid Kantorovitch lays the bases of LP (Nobel prize 1975).

1947 George Dantzig proposes independently LP and invents the simplex algorithm.

1958 Ralph Gomory proposes a cutting plane method for ILP problems.

1.4 Multi-objective programming

❓ How is it born?

Even though some real-word problems can be reduced to a matter of a single objective very often it is hard to define all the aspects in terms of a single objective. Defining multiple objectives often gives a better idea of the task.

Multi-objective programming (also known as **Multi-objective optimization** or **Pareto optimization**) is an **area of multiple-criteria decision making** that is concerned with mathematical optimization problems involving **more than one objective function to be optimized simultaneously**. Multi-objective is a type of vector optimization that has been applied in many fields of science, including engineering, economics and logistics where optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives.

Minimizing cost while maximizing comfort while buying a car, and maximizing performance whilst minimizing fuel consumption and emission of pollutants of a vehicle are **examples** of multi-objective optimization problems involving two and three objectives, respectively. [1]

Suppose to minimize $f_1(\mathbf{x})$ and maximize $f_2(\mathbf{x})$ (e.g. laptop: f_1 is cost and $f_2(\mathbf{x})$ is performance):

1. Turn it into a **single objective problem** by expressing the two objectives in terms of the same unit (e.g. monetary unit):

$$\min \lambda_1 f_1(\mathbf{x}) - \lambda_2 f_2(\mathbf{x})$$

   for appropriate scalars λ_1 and λ_2.

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

$$\max_{x \in \tilde{X}} f_2(\mathbf{x}) \quad \text{where} \quad \tilde{X} = \{\mathbf{x} \in X \ : \ f_1(\mathbf{x}) \leq \varepsilon\}$$

   for appropriate constant ε.

This is a simple introduction, the more detailed explanation will be explained in the following pages.

1.5 Mathematical Programming or Simulation?

Mathematical Programming and Simulation involves **building a model** and **designing an algorithm**. And the main differences are:

Mathematical Programming	**Simulation**
Problem can be "well" formalized.	Problem is difficult to formalize.
Algorithm yields a(n optimal) solution.	Algorithm simulates the evolution of the real system and allows to evaluate the performance of alternative solutions.
The results are "certain"	The results need to be interpreted.
Example: assignment.	Example: service counters.

Table 2: Major differences between Mathematical Programming and Simulation.

2 Graph and network optimization

2.1 Graphs

2.1.1 Definitions and characteristics

In the following section we will explain some basic concepts about the graphs. The terms used are important, and we will use the definition box to highlight these words.

Definition 1: graph

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

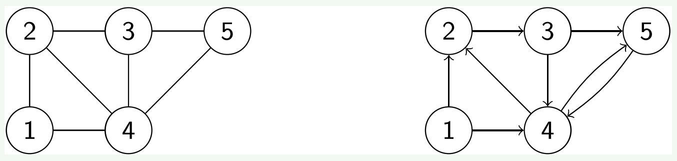
Definition 2: edge

An **edge** connecting the nodes i and j is represented by

- Graph undirected: $\{i, j\}$
- Graph directed: (i, j)

Example 1

For **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.



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

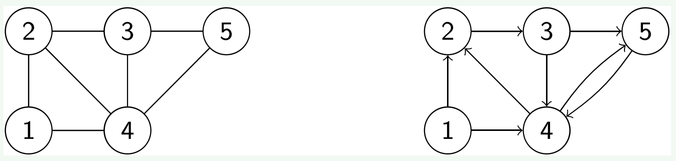
Some graph properties are:

- 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 other words, in a graph G, two edges are incident **if they share a common vertex**. For example, edge $E_1 = (v_1, v_2)$ and edge (v_1, v_3) are incident as they share the same vertex v_1.
- The degree concept depends on the type of graph:
  - Undirected graph: the **degree** of a node is the **number of incident edges**.
  - Directed graph: the **in-degree** and **out-degree** of a node is the **number of arcs that have it as succesor** and **predecessor**.

Example 2: adjacent, incident, degree, in-degree and out-degree

Given the graphs:



- Undirected graph:
  - Nodes 1 and 2 are **adjacent** (unlike nodes 1 and 3).
  - Edge $\{1, 2\}$ is **incident** in nodes 1 and 2.
  - Node 1 has **degree** 2, node 4 has **degree** 4.
- Directed graph: node 1 has **in-degree** 0, and **out-degree** 2.

Other useful features include:

- A **(directed) path from $i \in N$ to $j \in N$** is a sequence of (arcs) edges:

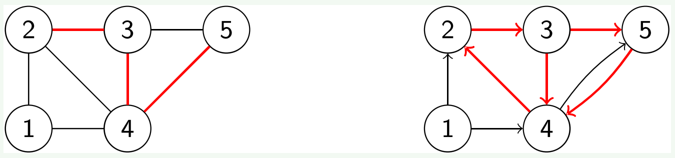
$$p = \langle \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\} \rangle$$

  Connecting nodes v_1 and v_k, with $\{v_i, v_{i+1}\} \in E$, for $i = 0, \ldots, k-1$.
- A generic **node** u and v are **connected** if there is a path connecting them.
- A **graph** (N, E) is **connected** if two generic nodes u, v are connected, $\forall u, v \in N$. Recall that in generic graph notation, the variable N represents a set of nodes or vertices and E represents a set of edges or arcs connecting them in pairs.

- A **graph** (N, E) is **strongly connected** if two generic nodes u, v are connected by a directed path, $\forall u, v \in N$ (for any node in the set of nodes or vertices of the graph).

Example 3: directed path, connected nodes, connected graph, strongly connected

Given the graphs:



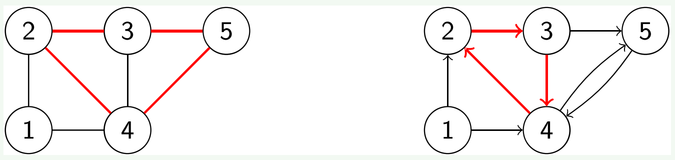
- Undirected graph:
  - $\langle \{2,3\}, \{3,4\}, \{4,5\} \rangle$ is a **path** from node 2 to node 5.
  - **Nodes** 2 and 5 are **connected**.
  - It is a **connected graph**.
- Directed graph:
  - $\langle \{3,5\}, \{5,4\}, \{4,2\}, \{2,3\}, \{3,4\} \rangle$ is a **directed path** from node 3 to node 4.
  - It is not a **strongly connected graph** because the node 1 cannot be the destination of none path. In other words, doesn't exist a directed path from node u to node 1 (where u is a generic node, $\forall u \in N \setminus \{1\}$).

Finally, there are other interesting properties and notations about graphs and edges:

- A **cycle (circuit)** is a directed path with $v_1 = v_k$ (source and destination are the same).
- A **graph** is **bipartite** if there is a partition $N = N_1 \cup N_2$ $(N_1 \cap N_2 = \emptyset)$ such that no edge connects nodes in the same subset.
- A **graph** is **complete** if $E = \{\{v_j, v_j\} \,:\, v_i, v_j \in N \,\wedge\, i \leq j\}$.
- Given a directed graph $G = (N, A)$ and $S \subset N$, the **outgoing cut** induced by S is:
$$\delta^+(S) = \{(u,v) \in A \,:\, u \in S \wedge : v \in N \subseteq S\}$$
The **incoming cut** induced by S is:
$$\delta^-(S) = \{(u,v) \in A \,:\, v \in S \wedge : u \in N \subseteq S\}$$

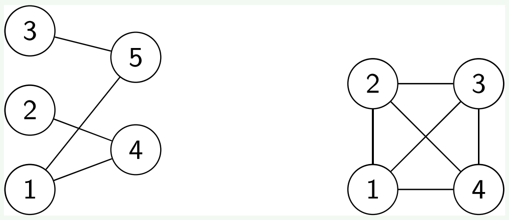
Example 4: cycle/circuit in graph, bipartite graph, complete graph, out/incoming cut

An example of cycle in graph:



- Undirected graph: $\langle \{2,3\}, \{3,5\}, \{5,4\}, \{4,2\} \rangle$ is a cycle.
- Directed graph: $\langle (2,3), (3,4), (4,2) \rangle$ is a circuit.

An example of bipartite/complete graph:

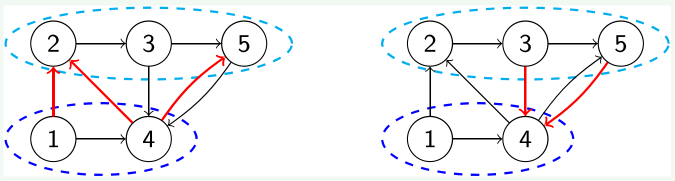


- To the left a **bipartite graph**, because:

$$N_1 = \{1,2,3\} \qquad N_2 = \{4,5\}$$

- And to the right a **complete graph**.

Finally, an example of out/incoming cut:



- Left graph:

$$\begin{aligned} \delta^+ (\{1,4\}) &= (\{1,2\}, \{4,2\}, \{4,5\}) \\ S &= \{1,4\} \\ N \setminus S &= \{2,3,5\} \end{aligned}$$

- Right graph:

$$\begin{aligned} \delta^- (\{1,4\}) &= (\{3,4\}, \{5,4\}) \\ S &= \{1,4\} \\ N \setminus S &= \{2,3,5\} \end{aligned}$$

2.1.2 Graphical representation

Such a matrix can easily be represented as a graph. This guarantees that it can be stored efficiently in a computer. But to understand how to do this in general, it's important to understand some other important properties:

- For any graph G with n nodes, the **number of edges** satisfies:
  - $m \leq \dfrac{n(n-1)}{2}$ if G is undirected.
  - $m \leq n(n-1)$ if G is directed.
- A graph is **dense** if $m \approx n^2$ and **sparse** if $m \ll n^2$. Where m is the number of arcs and n the number of nodes.
- For dense directed graphs, exist an **adjacency matrix** $A_{n\times n}$:

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

To build the adjacency matrix it is necessary to create a **list of successors** for each node. In other words, for **each node** we need to write the **outgoing edges** and write the matrix.

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{aligned} S(1) &= \{2,4\} \\ S(2) &= \{3\} \\ S(3) &= \{4,5\} \\ S(4) &= \{2,5\} \\ S(5) &= \{4\} \end{aligned}$$

Each row represents a node, and we set the value 1 if the column index is a node that has the arc of the row node as its incoming edge. So row one (node one) has the value one in column two (node two) and column four (node four).

2.1.3 Graph reachability problem

In general the **graph reachability problem** can be formulated as follows.

Definition 3: graph reachability problem

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

Where N is the set of nodes and A is the set of edges.

The problem takes:

- As **input** a ***graph*** $G = (N, A)$ described by the successor lists and node $s \in N$.
- As **output** produces a ***subset*** $M \subseteq N$ ***of nodes*** of the graph G reachable from s.

Our goal is to devise an efficient algorithm that allows us to find all nodes reachable from s.

2.1.3.1 Description and algorithm

Definition 4: Breadth-First Search

Breadth-First Search (BFS) is an **algorithm for searching a tree data structure for a node that satisfies a given property**. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

```
1  Q ← {s};
2  M ← ∅;
3  while Q ≠ ∅ do:
4      u ← node in Q;
5      Q ← Q \ {u};
6      // label u
7      M ← M ∪ {u} ;
8      for (u, v) ∈ δ+ (u) do:
9          if v ∉ M and v ∉ Q:
10             Q ← Q ∪ {v}
```

Algorithm 1: Graph reachability problem: Breadth-First Search $O(|N| + |A|)$

Rows 1-2. Declare a queue `Q` containing the nodes reachable from the source `s` and **not yet processed**. It is managed as a FIFO (First-In First-Out) queue. By definition, we add the `s` node at the beginning because it is our entry point.

Then we declare the set `M`. It represents the subset of nodes of the graph that are reachable from the source `s`. Obviously, it is empty at the beginning of the algorithm.

Row 3. The BFS algorithm continues to process the nodes until the queue is empty. As long as there is an element, it continues.

Rows 4-5. Take a node from the queue `Q` and assign it to the variable `u`. Also remove the element `u` from the set `Q`. In other words, perform a difference operation between the sets `Q` and the set composed only of the element `u` (`Q` $\setminus$ {`u`}).

For example, in Python we can get the same result using the `popleft` method of the `deque` data structure.

Row 7. Using the union between sets, add the visited node `u` to the subset `M` of reachable nodes. This operation is also called "labeling" because you are *labeling* a node as *visited*.

Row 8. Iterate each tuple (node `u` just popped from the queue, node `v` adjacent to node `u`) in the outgoing cut set of node `u`.

Rows 9-10. If the adjacent node `v` is not in the reachable set `M` and it is not in the queue (so it is not waiting to be evaluated), add the adjacent node `v` to `Q` using the union set operation.

As we said, the algorithm continues until the queue is not empty. Note that the queue is updated each time a neighboring node is found that is not already in the solution set (`M`).

2.1.3.2 Complexity of algorithm

BFS Algorithm - Time Complexity

The BFS time complexity[1] can be expressed as $O(|N| + |A|)$, since **every node and every edge will be explored in the <u>worst case</u>**.

- $|N|$ is the number of **nodes**;
- $|A|$ is the number of **edges** in the graph.

Note that $O(|A|)$ may vary between $O(1)$ and $O\left(N^2\right)$, depending on how sparse the input graph is. For example, for **dense graphs**, exactly $O\left(N^2\right)$.

BFS Algorithm - Space Complexity

When the number of nodes (or vertices) in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity[2] can be expressed as $O(|N|)$, where $|N|$ is the number of vertices. This is in addition to the space required for the graph itself, which may vary depending on the graph representation used by an implementation of the algorithm.

In other words, the algorithm needs:

- The **space to store** the set N, i.e. the **set of all nodes** in the graph.
- The **space to store the graph itself** depends on the implementation used.

[1]In theoretical computer science, the time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to be related by a constant factor. (source)

[2]The space complexity of an algorithm or a data structure is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely. This includes the memory space used by its inputs, called input space, and any other (auxiliary) memory it uses during execution, which is called auxiliary space. (source)

2.2 Trees

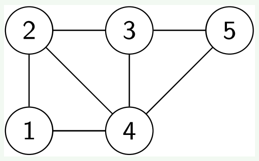
2.2.1 Definitions and characteristics

Before introducing what a tree is, it is necessary to understand what a subgraph is. Mathematically speaking, $G' = (N', E')$ is a **subgraph** of $G = (N, E)$ if $N' \subseteq N$ and $E' \subseteq E$.

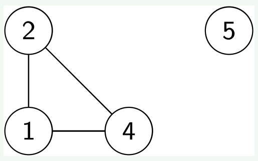
A **tree of the graph** G is a connected and acyclic subgraph of G and it is represented as $G_T = (N', T)$. If the tree contains exactly every node in the graph G, it is called the **spanning tree** of G and is represented as $G_T = (N', T)$. Finally, we call the **nodes of degree 1** in a tree as **leaves**.

Example 5: subgraph, tree and spanning tree

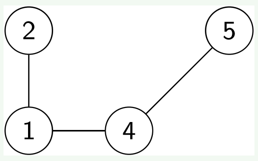
Given a graph G:



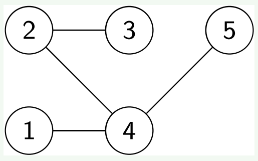
- The following figure is a **subgraph** of G, but not a tree, because there is a cycle $(1, 2, 4)$.



- The following figure is a **subgraph** of G, and it is a **tree** because there are no cycles and the graph is connected.



- The following figure is a **spanning tree** of G because it contains all the nodes in G, and it is a tree because it is connected and acyclic.



2.2.2 Properties

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

   Inductive proof. Base case: the claim holds for $n=1$ (tree with 1 node and 0 edges).

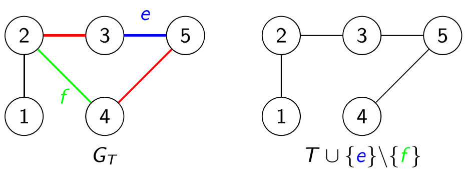
   Inductive step: show that, if this is true for trees with n nodes, then it is also true for those with $n+1$ nodes.

   Let T_1 be a tree with $n+1$ nodes and recall that any tree with $n \geq 2$ nodes has at least 2 leaves (two nodes of degree 1, the number of incident edges). By deleting one 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. QED

2. **Every pair of nodes in a tree is connected by a unique path**. The proof is not necessary, because otherwise there would be a cycle (and this is against the definition of a tree).

3. **By adding a new edge to a tree, we can create a unique cycle**.

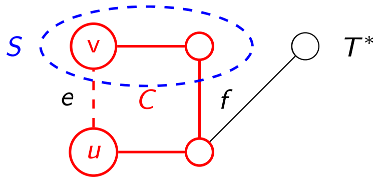
4. Let $G_T=(N,T)$ be a spanning tree of $G=(N,E)$. Consider an edge $e \notin T$ and the unique cycle C of $T \cup \{e\}$ (as in property 3). For each edge $f \in C \setminus \{e\}$, the subgraph $T \cup \{e\} \setminus \{f\}$ is also a spanning tree of G.



5. Let F be a partial tree (spanning nodes in $S \subseteq N$) 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 (is better explained in the 2.2.3.1 paragraph, page 29).

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



QED

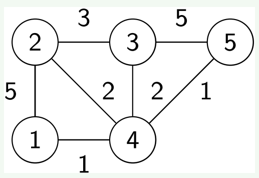
2.2.3 Optimal cost spanning trees

Spanning trees are very common because they are used in a wide range of applications such as network design, IP network protocols, enterprise storage, etc.

Example 6: introduction to finding the best and optimal cost solution

Design a communication network so as to connect n cities at **minimum total cost**.
The model is made up as follows:

- Graph $G = (N, E)$ with $n = |N|$, $m = |E|$
- Cost function $c : E \to c_e \in \mathbb{R}$, with $e = \{u, v\} \in E$.

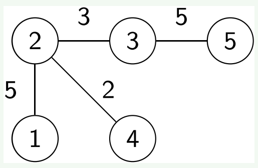


The required properties are:

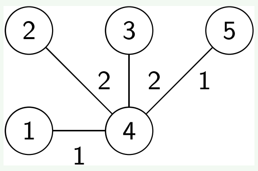
- Each pair of cities must communicate, then the connected subgraph containing all the nodes.
- The minimum total cost, then the subgraph must have no cycles.

We give two solutions, where the second is better because it is more feasible and optimal:

1. $c(T_1) = 15$



2. $c(T_2) = 6$



In general, we can formalize the **optimal cost** as follows. Given an undirected graph $G = (N, E)$ and a cost function, **find a spanning tree** $G_T = (N, T)$ **of minimum total cost**:

$$\min_{T \in X} \sum_{e \in T} c_e \qquad \text{where } X \text{ is the set of all spanning trees of } G \tag{1}$$

Doesn't exists only one spanning tree, they are multiples. But the number of spanning trees available in a complete graph is defined by a theorem.

Theorem 1 (Cayley, 1889)**.** *A complete graph with n nodes ($n \geq 1$) has n^{n-2} spanning trees.*

In general, the problem of finding a spanning tree with minimum total cost is also called **minimum spanning tree (MST) problem**. It plays an important role in many networking applications, such as routing and networking. [3]

2.2.3.1 Prim's algorithm

Definition 5: Prim's

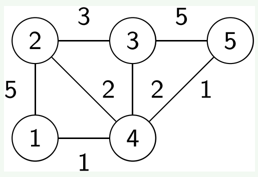
Prim's algorithm is a **greedy algorithm that finds a minimum spanning tree for a weighted undirected graph**. This means it **finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized**. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex. Finally, **Prim's algorithm is exact** (it provides an optimal solution for every instance).

A **greedy algorithm** constructs a **feasible solution iteratively by making a "locally optimal" choice at each step, without reconsidering previous choices**. In Prim's algorithm, at each step a minimum-cost edge is selected from those in the cut $\delta(S)$ induced by the current set of nodes S. Unfortunately, for most discrete optimization problems greedy-type algorithms yield a feasible solution with no guarantee of optimality.

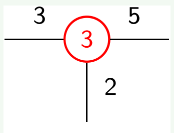
According to the definition of a greedy algorithm, the main idea of Prim is to build a spanning tree iteratively. It **starts with an initial tree** (S, T) with $S = \{u\}$ ($u \in N$, so it can be any node in the set N) and $T = \emptyset$. At **each step**, **add** to the current sub-tree (S, T) **a minimum cost edge** among those that connect a node in S to a node in $N \setminus S$.

Example 7: Prim's algorithm

Suppose we have the following graph:



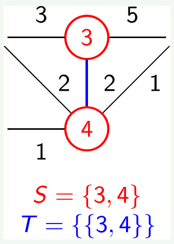
1. We start from an arbitrarily node u that it is in the set N, for example 3:



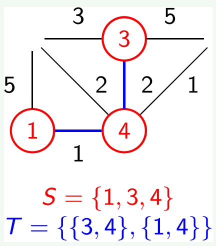
$S = \{3\}$
$T = \emptyset$

2. As we said, at each step we add to the current subtree (S, T) a minimum cost edge among those that connect a node in S to a

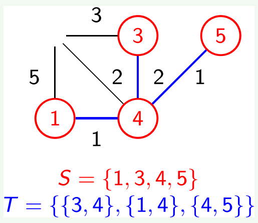
node in $N \setminus S$. So at this step we choose *node 4* because the edge starting from *node 3* is the less weighted edge.



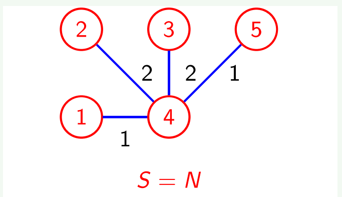
3. At this point we continue with the same logic. The edge, starting from *node 4*, with less weighted edge is *node 1*. Note that at parity of the weighted edge, it doesn't matter which one we choose. Maybe the decision will lead to a different subgraph, but Prim's algorithm is greedy by definition, then it doesn't think about these problems; it assumes that it is a locally optimal choice.



4. Since we are lucky, the previous doubt is useless (with parity of weighted vertices, which should we choose?). Because *node 1* exposes an edge with a weight of 5, but *node 4* has a weighted edge of only 1, the choice is clear.



5. Finally, we choose the edge with a weight of 2 (to the *node 2*) because is the lowest. At this step, the algorithm is finished because there are no more vertices ($S = N$). The cost function returns the value 6 (sum of each weighted edge selected).



$S = N$
$T = \{\{3,4\},\{1,4\},\{4,5\},\{2,4\}\}$

From the previous example it should be clear that at each step the Prim's algorithm creates the need to solve a minimum search problem.

The Prim's algorithm take:

- Input: a connected graph $G = (N, E)$ with edge costs.
- Output: a subset $T \subseteq N$ of edges of G such that $G_T = (N, T)$ is a minimum cost spanning tree of G.

```
S ← {u};
T ← ∅;
while $|T| < n - 1$ do:
    $\{u\} \leftarrow$ edge in $\delta(S)$ of minimum cost; // with $u \in S$ and $v \in N \setminus S$
    $S \leftarrow S \cup \{v\}$;
    $T \leftarrow T \cup \{\{u, v\}\}$
```

Algorithm 2: Minimum spanning tree (MST) problem: Prim's $O(nm)$

Rows 1-2. Declare the general sets S and T. The first is filled with the starting node u and the second is empty, because the core of the algorithm has not yet started.

Row 3. Continue building the spanning tree until the length of the set of transitions T is not equal to the number of nodes minus one.

Row 4. Find the edge with the lowest weight from the set $\delta(S)$. Then choose one of the edges with the lowest weight and the corresponding target node. Obviously the target node v cannot be already evaluated ($v \in N \setminus S$) and the source node u must be in the set of nodes already evaluated ($u \in S$).

Rows 5-6. Therefore, the most complex part of the algorithm (minimum search) is to store the target vertex with the lowest edge weight and add the tuple (source node, target node) to the set T.

The complexity of the algorithm is pretty much guessed. Or in the better case neither worst case, we need to evaluate each node. Then the main difference is made by the weighted edge search. If each edge has to be analyzed at each iteration, the **complexity** order is given by $O(nm)$.

2.2.3.2 Implementation of Prim's algorithm in $O\left(n^2\right)$

Prim's algorithm is based on graph traversals (visiting every vertex in the graph), which are inherently difficult to parallelize. It also has an irregular memory access pattern. In CPUs, this limits the use of the cache and leads to an overall performance penalty. The prim's algorithm is highly dependent on the organization of memory storage and memory access patterns. [3]

The following data structure we propose guarantees a complexity of $O\left(n^2\right)$.

- k is the number of edges selected so far;
- S is the subset $S \subseteq N$ of nodes incident to the selected edges (same as explained in the 2.2.3.1 section, page 29);
- T is the subset $T \subseteq E$ of selected edges (same as explained in the 2.2.3.1 section, page 29);
- C_j is a vector which has a value equal to:

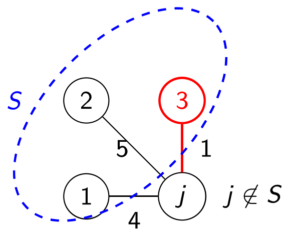
$$C_j = \begin{cases} \min\left\{c_{ij} \ : \ i \in S\right\} & j \notin S \\ \infty & \text{otherwise} \end{cases}$$

  At the beginning of the algorithm it is clearly composed of infinite values if the edge i to j doesn't exist, otherwise the weight of the edge. In the core of the algorithm, each position is updated with the minimum weighted edge value.

- closest_j is a vector which has a value equal to:

$$\text{closest}_j = \begin{cases} \operatorname{argmin}\left\{c_{ij} \ : \ i \in S\right\} & j \notin S \\ \text{predecessor of } j \text{ in the minimum spanning tree} & j \in S \end{cases}$$

  The node closest to the edge has less weight, otherwise if we look at the node j in the set S, the predecessor of that node in the minimum spanning tree. A trivial example:

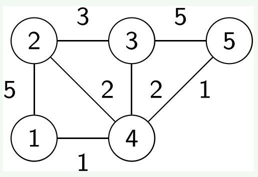


  With $\text{closest}_j = 3$ and $c_{\text{closest}_j}, j = 1$.

Let's take an example to clear up any doubts.

Example 8: Prim's algorithm $O(n^2)$

Suppose we have the following graph:

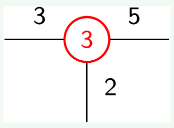


1. At the beginning we choose an arbitrary node u that is in the set N, for example 3. We set the set of transitions T to the empty set, the set of evaluated nodes S to the selected node u, in our case the value 3, and finally the two vectors. The vector C_j is filled of infinite values if the edge u to j doesn't exist, otherwise the weight of the edge, and the vector closest_j is filled with the selected starting node, in our case 3.

    - $T = \emptyset$
    - $S = \{u\} = \{3\}$
    - $C_j = [+\infty, 3, -, 2, 5]$
    - $\text{closest}_j = [3, 3, -, 3, 3]$

    Some observations:

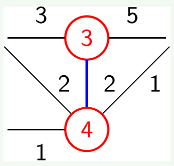
    - The symbol $-$ is the don't care term used in digital logic.
    - The position of each value respects the j-index. For example, the node 3 in the vectors C_j and closest_j is placed at position number 3. The value is don't care $(-)$ because it is the starting point. The other values depend on the graph. Node 1 has no direct edge to node 3, so it has an infty value; node 2 has a direct edge with weight 3; node 3 doesn't care because it's the starting point; and so on.

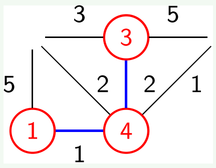
2. The minimum value in C_j is 2 and since it is in the 4th position, node 4 is selected.

    - $T = \{\{3, 4\}\}$
    - $S = \{3, 4\}$
    - $C_j = [\mathbf{1}, \mathbf{2}, -, 2, \mathbf{1}]$
    - $\text{closest}_j = [\mathbf{4}, \mathbf{4}, -, 3, \mathbf{4}]$

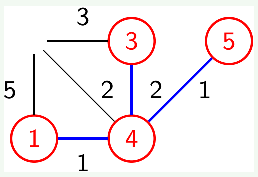
    The values updated are the first, second and fifth positions, as nodes three and four are in the S set.



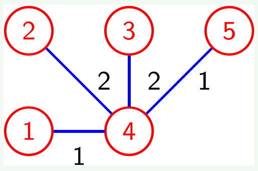
3. The minimum value in C_j is the first position. But the fifth position is also the minimum. We choose the first value because the vector is analyzed sequentially (first to last).
   - $T = \{\{3,4\}, \{1,4\}\}$
   - $S = \{1,3,4\}$
   - $C_j = [1, \mathbf{2}, -, 2, \mathbf{1}]$
   - $\text{closest}_j = [4, \mathbf{4}, -, 3, \mathbf{4}]$



4. The minimum value in C_j is quite trivial, between 2 and 1. We choose the value 1 and the node 4 is the closest.
   - $T = \{\{3,4\}, \{1,4\}, \{4,5\}\}$
   - $S = \{1,3,4,5\}$
   - $C_j = [1, \mathbf{2}, -, 2, 1]$
   - $\text{closest}_j = [4, \mathbf{4}, -, 3, 4]$



5. Finally, node 2 with edge weight 2 is selected.
   - $T = \{\{3,4\}, \{1,4\}, \{4,5\}, \{2,4\}\}$
   - $S = \{1,2,3,4,5\} = N$
   - $C_j = [1, 2, -, 2, 1]$
   - $\text{closest}_j = [4, 4, -, 3, 4]$



Ok, but how do we get the spanning tree from the nearest vector?

The minimum spanning tree found by Prim's algorithm consists of the $n-1$ edges:

$$\{\text{closest}_j, j\} \qquad \text{with } j = 1, 2, \dots, n$$

For example, from the previous example, let the *closest* vector:

$$\text{closest}_j = [4, 4, -, 3, 4]$$

A minimum cost spanning tree consists of the edges:

$$\{4, 1\}, \{4, 2\}, \cancel{\{-, -\}}, \{3, 4\}, \{4, 4\}$$

```
 1  S ← {u};
 2  T ← ∅;
 3  for $j \in N \setminus S$ do:
 4      $C_j \leftarrow c_{uj}$; // or $+\infty$ if $\{u,j\} \notin E$
 5      closest$_j \leftarrow u$;
 6  for $k = 1, \dots, n-1$ do:
 7      // select min edge in $\delta(S)$
 8      min ← $+\infty$;
 9      for $j = 1, \dots, n$ do:
10          if $j \notin S$ and $C_j <$ min:
11              min ← $C_j$;
12              v ← j;
13      // extend S and T
14      S ← S $\cup \{v\}$;
15      T ← T $\cup \{\{\text{closest}_v, v\}\}$;
16      // update $C_j$ and closest$_j$, $\forall j \notin S$
17      for $j = 1, \dots, n$ do:
18          if $j \notin S$ and $c_{vj} < C_j$:
19              $C_j \leftarrow c_{vj}$;
20              closest$_j \leftarrow v$;
```

Algorithm 3: Minimum spanning tree (MST) problem: Prim's $O\left(n^2\right)$

Rows 1-2. Declare the general sets S and T. The first is filled with the starting node u and the second is empty, because the core of the algorithm has not yet started.

Rows 3-5. The first for statement is used to initialize the two vectors C_j and closest_j. It inserts the edge weight into C_j if there is a direct edge from u to j, otherwise infinity is used. Meanwhile, the *closest* vector consists only of the starting node u at the beginning of the algorithm.

Row 6. The second for statement is the core of the algorithm. Here the index goes from one to the number of nodes minus one.

Rows 8-12. This piece of code is used to select the minimum edge available in the C_j vector. It starts by setting the `min` variable to infinity, to ensure that a value is selected. Therefore, the for statement iterates over each node; at each iteration, if the selected node is not in the S set (so it has not already been evaluated) and the value at the corresponding position in the vector C_j is less than minimum, then assign to the minimum the value of the vector C_j at position j and to v the index j.

Rows 14-15. Now it updates the variable S with the node v and the transitions set with the tuple (value at the corresponding position in the vector closest_v, node v).

Rows 17-20. There is another for statement similar to the previous one, because here it needs to update the vectors C_j and closest_j with the new values. The for iterates over every node of the graph. So at each iteration it checks that the selected vertex is not in the S-set (not already evaluated) and that the weight of the edge from vertex v to j (if it exists, otherwise infinity) is less than the value C_j. If the double condition is true, it updates the two vectors at position j.

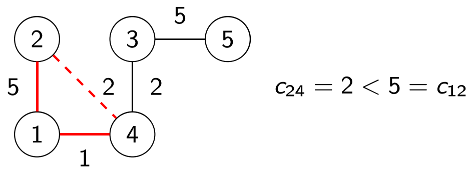
The overall complexity is given by:

- The number of iterations of the first for at row 3, which is: $(n-1)$
- Plus the number of iterations of the second for at row 6, which is: $(n-1)$
- Times the number of iterations of the third and fourth for at lines 9 and 17, which is: $(n-1+n-1)$

The result is $O\left(n^2\right)$. For sparse graphs, a more sophisticated data structure leads to an $O\left(m \log n\right)$ complexity.

2.2.3.3 Optimality condition

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 $\exists f \in C \setminus \{e\}$ such that $c_e < c_f$.



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

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

Theorem 2 (Tree optimality condition)**.** *A tree T is of minimum total cost if and only if no cost-decreasing edge exists.*

Proof. $\Rightarrow$ If a cost-decreasing edge exists, then T is not of minimum total cost.

$\Leftarrow$ 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. QED

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

3 Laboratory

3.1 Introduction

For this course of Foundations of Operations Research we will use the `mip` package (official website, installation guide) for modelling optimization problems in Python. The mip package is a collection of Python tools for the modeling and solution of Mixed-Integer Linear programs (MIPs).

3.1.1 Diet problem

A canteen has to plan the composition of the meals that it provides. A meal can be composed of the types of food indicated in the following table. Costs, in Euro per hg, and availabilities, in hg, are also indicated.

Food	**Cost**	**Availability**
Bread	0.1	4
Milk	0.5	3
Eggs	0.12	1
Meat	0.9	2
Cake	1.3	2

A meal must contain at least the following amount of each nutrient:

Nutrient	**Minimum quantity**
Calories	600 cal
Proteins	50 g
Calcium	0.7 g

Each hg of each type of food contains to following amount of nutrients:

Food	**Calories**	**Proteins**	**Calcium**
Bread	30 cal	5 g	0.02 g
Milk	50 cal	15 g	0.15 g
Eggs	150 cal	30 g	0.05 g
Meat	180 cal	90 g	0.08 g
Cake	400 cal	70 g	0.01 g

Give a linear programming formulation for the problem of finding a diet (a meal) of minimum total cost which satisfies the minimum nutrient requirements.

Diet problem formulation

- Sets
  - I: food types
  - J: nutrients
- Parameters
  - c_i: unit cost of food type $i \in I$
  - q_i: available quantity of food type $i \in I$
  - b_j: minimum quantity of nutrient $j \in J$ required
  - a_{ij}: quantity of nutrient $j \in J$ per unit of food of type $i \in I$

- Variables
  - x_i: quantity of food of type $i \in I$ included in the diet
- Model

$$\begin{array}{llll} \min & \sum_{i\in I} c_i x_i & & \text{(cost)} \\ \text{s.t.} & \sum_{i\in I} a_{ij} x_{ij} \geq b_j & j \in J & \text{(min nutrients)} \\ & x_i \leq q_i & i \in I & \text{(availability)} \\ & x_i \geq 0 & i \in I & \text{(nonnegativity)} \end{array}$$

Diet problem implementation

1. Write the dataset specified by input:

```python
# We need to import the mip package (useful later)
import mip

# Food
I = {'Bread', 'Milk', 'Eggs', 'Meat', 'Cake'}

# Nutrients
J = {'Calories', 'Proteins', 'Calcium'}

# Cost in Euro per hg of food
c = {
    'Bread': 0.1,
    'Milk': 0.5,
    'Eggs': 0.12,
    'Meat': 0.9,
    'Cake': 1.3
}

# Availability per hg of food
q = {
    'Bread': 4,
    'Milk': 3,
    'Eggs': 1,
    'Meat': 2,
    'Cake': 2
}

# Minimum nutrients
b = {
    'Calories': 600,
    'Proteins': 50,
    'Calcium': 0.7
}

# Nutrients per hf of food
a = {
    ('Bread', 'Calories'): 30,
    ('Milk', 'Calories'): 50,
    ('Eggs', 'Calories'): 150,
    ('Meat', 'Calories'): 180,
    ('Cake', 'Calories'): 400,
    ('Bread', 'Proteins'): 5,
    ('Milk', 'Proteins'): 15,
    ('Eggs', 'Proteins'): 30,
    ('Meat', 'Proteins'): 90,
    ('Cake', 'Proteins'): 70,
    ('Bread', 'Calcium'): 0.02,
    ('Milk', 'Calcium'): 0.15,
    ('Eggs', 'Calcium'): 0.05,
    ('Meat', 'Calcium'): 0.08,
    ('Cake', 'Calcium'): 0.01
}
```

2. Now we create an empty model and add the variables:

```
# Define a model
model = mip.Model()

# Define variables
x = [model.add_var(name = i, lb = 0) for i in I]

# Define the objective function
model.objective = mip.minimize(
    mip.xsum(x[i] * c[food] for i, food in enumerate(I))
)

# CONSTRAINTS
# Availability constraint
for i, food in enumerate(I):
    model.add_constr(x[i] <= q[food])

# Minimum nutrients
for j in J:
    model.add_constr(
        mip.xsum(
            x[i] * a[food, j] for i, food in enumerate(I)
        ) >= b[j]
    )
```

3. The model is complete. Let us solve it and print the optimal solution:

```
# Optimizing command
print('Optimizing: ' + model.optimize())
# Optimizing: OptimizationStatus.OPTIMAL

# Optimal objective function value
print('Optimal objective function value: {}'.format(
    model.objective.x
))
# Optimal objective function value: 3.37

# Printing the variables values
for i in model.vars:
    print(i.name, i.x)
# Meat 1.5000000000000002
# Bread 3.9999999999999996
# Cake 0.0
# Milk 3.0
# Eggs 1.0
```

3.1.2 Oil blending problem

A refinery has to blend 4 types of oil to obtain 3 types of gasoline. The following table reports the available quantity of oil of each type (in barrels) and the respective unit cost (Euro per barrel):

Oil type	**Cost**	**Availability**
1	9	5000
2	7	2400
3	12	4000
4	6	1500

Blending constraints are to be taken into account, since each type of gasoline must contain at least a specific, predefined, quantity of each type of oil, as indicated in the next table. The unit revenue of each type of gasoline (Euro per barrel) is also indicated:

Gasoline type	**Requirements**	**Revenue**
A	$\geq$ 20% of type 2	12
A	$\leq$ 30% of type 3	12
B	$\geq$ 40% of type 3	18
C	$\leq$ 50% of type 2	10

Oil blending problem formulation

- Sets
  - I: set of oil types
  - J: set of gasoline types
- Parameters
  - c_i: unit cost for oil of type $i \in I$
  - b_i: availability of oil of type $i \in I$
  - r_j: price of gasoline of type $i \in I$
  - b_i: minimum quantity of nutrient $i \in I$ required
  - $q_{ij}^{\max}$: maximum quantity (percentage) of oil of type $i \in I$ for gasoline of type $j \in J$
  - $q_{ij}^{\min}$: minimum quantity (percentage) of oil of type $i \in I$ for gasoline of type $j \in J$

- Variables
  - x_{ij}: units of oil of type $i \in I$ used for gasoline of type $j \in J$
  - y_j: units of gasoline of type $j \in J$ that are produced
- Model

$$
\begin{array}{lllr}
\max & \sum_{j\in J} r_j y_j - \sum_{i\in I, j\in J} c_j x_{ij} & & \text{(revenue)} \\
\text{s. t.} & \sum_{j\in J} x_{ij} \le b_i & i \in I & \text{(availability)} \\
& \sum_{i\in I} x_{ij} = y_j & j \in J & \text{(conservation)} \\
& x_{ij} \le q_{ij}^{\max} y_j & i \in I, j \in J & \text{(maximum qty)} \\
& x_{ij} \ge q_{ij}^{\min} y_j & i \in I, j \in J & \text{(minimum qty)} \\
& x_{ij}, y_j \ge 0 & i \in I, j \in J & \text{(nonnegativity)}
\end{array}
$$

Oil blending problem implementation

1. Write the dataset specified by input:

```
# Set of oil types
I = range(4)

# Set of gasoline types
J = {'A', 'B', 'C'}

# Unit cost for oil of type i
c = {0:9, 1:7, 2:12, 3:6}

# Availability of oil type i
b = {0:5000, 1:2400, 2:4000, 3:1500}

# Price of gasoline of type j
r = {'A':12, 'B':18, 'C':10}

# Maximum quantity (percentage) of oil
q_max = {}
for i in I:
    for j in J:
        q_max[(str(i), j)] = 1
q_max[('2', 'A')] = 0.3
q_max[('1', 'C')] = 0.5

# Minimum quantity (percentage) of oil
q_min = {}
for i in I:
    for j in J:
        q_min[(str(i), j)] = 0
q_min[('1', 'A')] = 0.2
q_min[('2', 'B')] = 0.4
```

2. Now we create an empty model and add the variables:

```
# Define a model
model2 = mip.Model()

# Define variables
x = {
    (str(i), j): model2.add_var(name=str(i)+','+j, lb=0)
    for i in I for j in J
}
y = {
    j: model2.add_var(name=j, lb=0) for j in J
}
```

3. Let us write the objective function: in the general case, it can be written as a sum over the set of models.

```
# Define the objective function
model2.objective = mip.maximize(
    mip.xsum(y[j]*r[j] for j in J) -
    mip.xsum(c[i]*x[str(i), j] for i in I for j in J)
)
```

4. The constraints can be generated in loops:

```
# CONSTRAINTS
# Availability constraint
for i in I:
    model2.add_constr(mip.xsum(x[str(i), j] for j in J) <= b[i
    ])

# Conservation constraint
for j in J:
    model2.add_constr(mip.xsum(x[str(i), j] for i in I) == y[j
    ])
# Maximum quantity
for i in I:
    for j in J:
        model2.add_constr(x[str(i), j] <= q_max[str(i),j]*y[j
    ])
# Minimum quantity
for i in I:
    for j in J:
        model2.add_constr(x[str(i), j] >= q_min[str(i),j]*y[j
    ])
```

5. The model is complete. Let's solve it and print the optimal solution.

```
# Optimizing command
print(model2.optimize())
# OptimizationStatus.OPTIMAL

# Optimal objective function value
print(model2.objective.x)
# 96000.0

# Printing the variables values
for i in model2.vars:
    print(i.name, i.x)
# 0,C   0.0
# 0,A   500.0
# 0,B   4500.0
# 1,C   0.0
# 1,A   2400.0
# 1,B   0.0
# 2,C   0.0
# 2,A   0.0
# 2,B   4000.0
# 3,C   0.0
# 3,A   0.0
# 3,B   1500.0
# C     0.0
# A     2900.0
# B     10000.0
```

References

[1] A. Abraham, L.C. Jain, and R. Goldberg. *Evolutionary Multiobjective Optimization: Theoretical Advances and Applications.* Advanced Information and Knowledge Processing. Springer, 2005.

[2] Braz Pascoal Marta Margarida. Foundations of Operations Research. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[3] Artur Mariano, Dongwook Lee, Andreas Gerstlauer, and Derek Chiou. Hardware and software implementations of prim's algorithm for efficient minimum spanning tree computation. In Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro C. Zanella, and Franz J. Rammig, editors, *Embedded Systems: Design, Analysis and Verification*, pages 151–158, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Wikipedia. Operations research - Wikipedia. `https://en.wikipedia.org/wiki/Operations_research`. [Accessed 08-09-2024].

Index