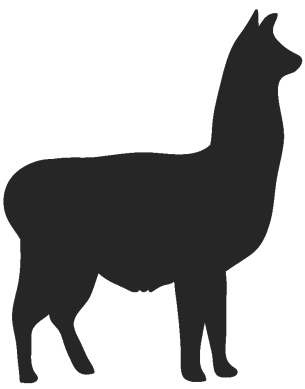
Foundations of Operations Research

Lorenzo Rossi and everyone who kindly helped!

2022/2023

Last update: 2022-11-08



no alpaca has been harmed while writing these notes

Contents

1 Introduction

1.1 Algorithm

An algorithm for a problem is a sequence of instructions that allows to solve any of its instances. The execution time of an algorithm depends on various factors, most notably the instance and the computer.

Properties

- An algorithm is **exact** if it provides an optimal solution for every instance.
  - otherwise is **heuristic**
- A **greedy algorithm** constructs a feasible solution iteratively, by making at each step a *locally optimal* choice, without reconsidering previous choices
  - for most *discrete optimization problems*, greedy type algorithms yield a feasible solution with no guarantee of optimality

1.2 Dynamic Programming

Proposed by *Richard Bellman* in 1950, **dynamic programming** *(or DP)* is a method for solving optimization problems, composed of a sequence of decisions, by solving a set of recursive equations.
DP is applicable to any sequential decision problem, for which the optimality property is satisfied; as such, it has a wide range of applications, including scheduling, transportation, and assignment problems.

1.3 Complexity of algorithms

In order analyze an algorithm, it's necessary to consider its complexity as a function of the size of the instance *(the size of the input)*, independently of the computer; the complexity is defined as the number of elementary operations by assuming that each elementary operation takes a constant time.
Since it's hard to determine the exact number of elementary operations, an additional assumption is made: only the asymptotic number of elementary operations in the worst case (for the worst instances) is considered.
The complexity evaluation is then performed by looking for the function $f(n)$ that best approximates the upper bound on the number of elementary operations n for the worst instances.

1.3.1 Big-O notation

A function f if order of g, written $f(n) = \mathcal{O}(g(n))$ if exists a constant $c > 0$ and a constant $n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
An illustration of the big-O notation is shown in Figure 1.

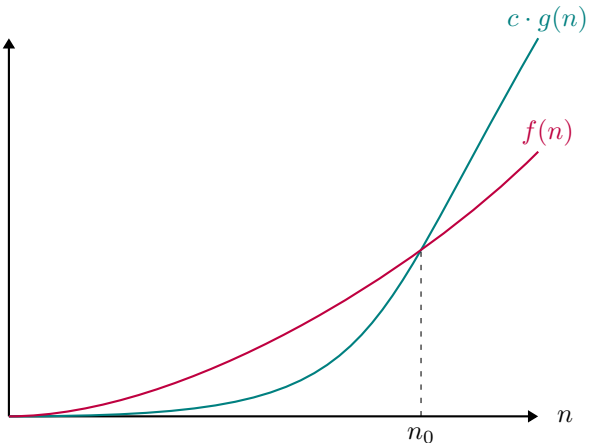


Figure 1: Big-O notation

n	n^2	2^n
1	$1\,\mu s$	$1\,\mu s$
10	$100\,\mu s$	$1.024\,ms$
20	$400\,\mu s$	$\approx 1.04\,s$
30	$900\,\mu s$	$\approx 18\,m$
40	$1.6\,ms$	$\approx 13\,d$
50	$2.5\,ms$	$\approx 36\,y$
60	$3.6\,ms$	$\approx 36535\,y$

Table 1: Complexity classes

1.4 Complexity classes

Two classes of algorithms are considered, according to their worst case order of complexity:

- **Polynomial**: $\mathcal{O}\left(n^d\right)$ for a constant $d > 0, d \in \mathbb{R}$
- **Exponential**: $\mathcal{O}\left(d^n\right)$ for a constant $d > 0, d \in \mathbb{R}$

Algorithms with a hight order Polynomial complexity are not considered efficient.
A comparison of the two classes, assuming that $1\,\mu s$ is needed for each elementary operation, is shown in Table 1.

2 Graph and Network Optimization

Many **decision making problems** can be formulated in terms of graphs and networks, such as:

- **transportation** and **distribution** problems
- **network design** problems
- **location** problem
- timetable **scheduling**
- ...

2.1 Graphs

A **graph** is a pair $G = (N, E)$ with:

- N a set of **nodes** or **vertices**
- $E \subseteq N \times N$ a set of **edges** or arcs connecting them pairwise
  - A **directed** graph is a graph in which set set E is composed by ordered pairs of distinct nodes
    $\rightarrow$ an edge connecting nodes i and j is represented by $\{i, j\}$
    $\rightarrow$ the flow is permitted only in the direction of the arrow *(from i to j)*
    $\rightarrow$ i is the head of the arc and j is the tail
  - A **undirected** graph is a graph in which set set E is composed by unordered pairs of distinct nodes
    $\rightarrow$ an edge connecting nodes i and j is represented by (i, j)
    $\rightarrow$ since all edges are undirected, (i, j) is equivalent to (j, i)

Properties

- Two **nodes** i and j are **adjacent** if they are connected by an edge (i, j) or $\{i, j\}$
- An **edge** e is **incident** in a node v if v is an **endpoint** of e
  - **undirected** graphs: the degree of a node is the number of incident edges
  - **directed** graphs: the in-degree *(out-degree)* of a node is the number of arcs that have it as successor *(predecessor)*
- A **path** from $i \in N$ to $j \in N$ is a sequence of edges in a undirected graph

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

  connecting nodes $v_1, \ldots, v_k$, with $\{v_i, v_{i+1} \in E\}$ for $i = 1, \ldots, k-1$
- A **directed path** $i \in N$ to $j \in N$ is a sequence of arcs in a directed graph

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

  connecting nodes, with $(v_i, v_{i+1} \in E)$ for $i = 1, \ldots, k-1$
- A **cycle** is a path

  $$\langle v_1, v_2, \ldots, v_1 \rangle$$

  where the first and last nodes are the same
  - A **directed cycle** is path

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

    where the first and the last nodes are the same
- Nodes u and v are **connected** if exists a path connecting them
- A graph (N, E) is **connected** if u, v are connecting $\forall\, u, v \in N$
  - A graph (N, E) is **strongly connected** if u, v are connected by a **directed** path $\forall\, u, v \in N$
- A graph is **bipartite** if there is a partition $N = N_1 \cup N_2$, $N_1 \cap N_2 = \emptyset$ such that $\forall\, (u, v) \in E, u \in N_1$ and $v \in N_2$
- A graph is **complete** if $E = \{\{v_i, v_j\} \mid v_i, v_j \in N \wedge i \leq j\}$

- Given a directed graph $G = (N, A)$ and $S \subseteq N$, the **outgoing cut** induced by S is the set of arcs:

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

  the **incoming cut** induced by S is the set of arcs:

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

- A graph with n **nodes** has at most $m = \dfrac{n(n-1)}{2}$ **edges**
- A **directed** graph with n **nodes** has at most $m = n(n-1)$ **arcs**
  - a graph is **dense** if $m \approx n^2$
  - a graph is **sparse** if $m \ll n$
- The adjacency list $A(i)$ of a node i is the set of arcs emanating from that node

$$A(i) = \{j \in N \mid (i, j) \in A\}$$

Some examples are shown in Figure 2.

2.1.1 Graphs representation

Graphs are represented by:

- Adjacency **matrix A** of size $n \times n$ if the graph is **dense**

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

- Adjacency **list A** of size n if the graph is **sparse**

The same representation can be used for both directed and undirected graphs; the adjacency matrix for an undirected graph is **symmetric**.

An example of a graph representation is shown in Figure 3.

2.1.2 Graph reachability problem

Problem 2.1 (Graph reachability)**.** Given a directed graph $G = (N, A)$ and a node $s \in N$, the **graph reachability problem** consists in finding all nodes reachable from s.

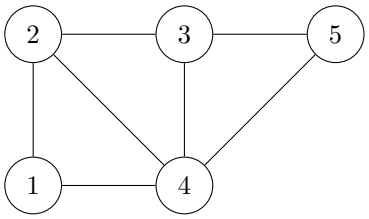
Goal

→ *Input*: graph $G = (N, A)$, described via successor lists, and a node $s \in N$
→ *Output*: subset $M \subseteq N$ of nodes of G reachable from s

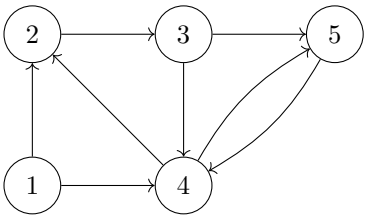
The goal is reached by an *efficient* algorithm to solve the problem, with the following properties:

- a **queue** Q of nodes not yet processed is kept by the algorithm
- the queue uses a `FIFO` policy
- the nodes exploration is performed in a **breadth-first** manner

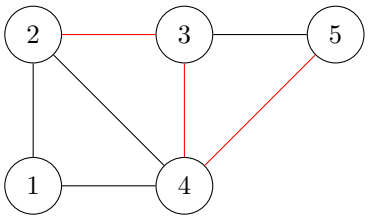
Algorithm The algorithm pseudocode is shown in Code 1.
The algorithm stops when $\delta^+(M) = \emptyset$ *(when the outgoing cut of the set of nodes M is empty)*; $\delta^-(M)$ is the set of arcs with head node in M and tail in $N \setminus M$.



(a) Undirected graph

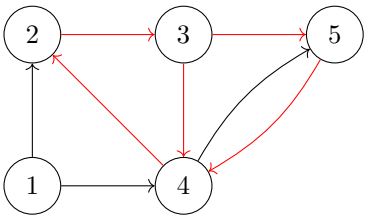


(b) Directed graph



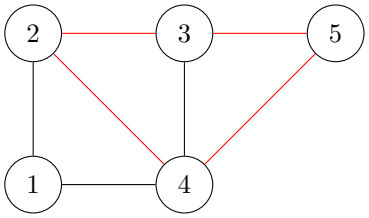
Connected graph, nodes 2 and 5 are connected

(c) $\langle\{2,3\},\{3,4\},\{4,5\}\rangle$ is a path



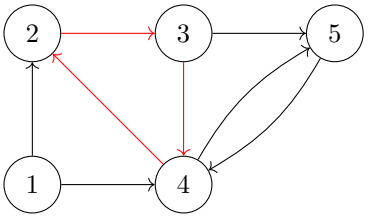
Not strongly connected graph

(d) $\langle\{3,5\},\{5,4\},\{4,2\},\{2,3\},\{3,4\}\rangle$ is a directed path



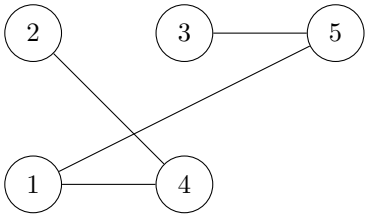
Cycle

(e) $\langle\{2,3\},\{3,5\},\{5,4\},\{4,2\}\rangle$ is a cycle



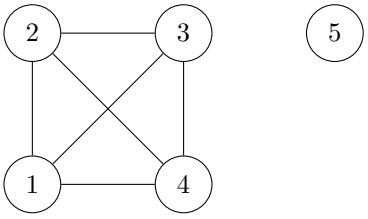
Circuit

(f) $\langle(2,3),(3,4),(4,2)$ is a circuit



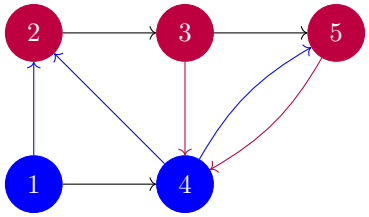
Bipartite graph

(g) $N_1 = \{1,2,3\}, N_2 = \{4,5\}$



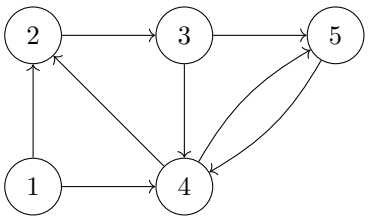
Complete graph

(h) $N = \{1,2,3,4\}$



(i) incoming (δ^+) and outgoing (δ^-) cuts of two sets of nodes *(purple and blue)*

Figure 2: Examples of graphs



(a) Graph

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

(b) Adjacency matrix

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

(c) Adjacency list

Figure 3: Graph representation

```
Q := {s}
M := {}
while Q is not empty do
  u := node ∈ Q
  Q := Q \ {u}
  M := M ∪ {u}
  for (u, v) ∈ δ+(u) do
    if v ∉ M and v ∉ Q then
      Q := Q ∪ {v}
    end
  end
end
```

Code 1: Graph reachability

2.1.2.1 Complexity analysis

At each iteration of the **while** loop:

1. A node u is **removed** from the queue Q and **added** to the set M
2. For all nodes v directly reachable from u and not already in M or Q, v is added to Q

Since each node u is inserted in Q at most once and each arch (u, v) is considered at most once, the overall complexity is:

$$\mathcal{O}(n + m) \quad n = |N|,\ m = |A|$$

For dense graphs, this value converges to $\mathcal{O}\left(n^2\right)$.

2.2 Subgraphs and Trees

Let $G = (N, E)$ be a graph. Then:

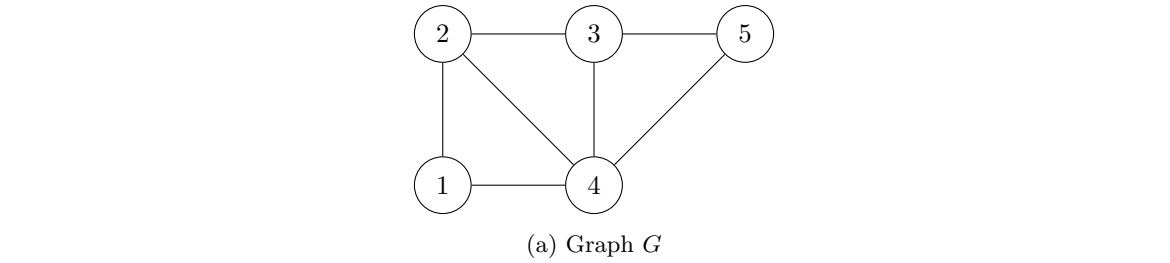
- $G' = (N', E')$ is a **subgraph** of G if $N' \subseteq N$ and $E' \subseteq E$
- A **tree** $G_T = (N', T)$ of G is a connected, acyclic, subgraph of G
- $G_T = (N', T)$ is a **spanning tree** of G if it contains all the nodes *($N' = N$)*
- The **leaves** of a tree are the nodes with degree 1

A representation of these concepts is shown in Figure 4.

2.3 Properties of trees

Property 2.1 (Number of edges)**.** Every tree with n nodes has $n - 1$ edges.

Proof.



(a) Graph G

(b) Subgraph G' of G

(c) Subgraph G'' of G and a tree

(d) Subgraph G'' of G and a tree

Figure 4: Subgraphs and trees

- **Base case**: the claim holds for $n = 1$ *(a tree with a single node has no edges)*
- **Inductive steps**: show that the claim is valid for for any tree with $n + 1$ nodes
  - let T_1 be a tree with $n + 1$ and recall with any tree with $n \geq 2$ nodes has at least 2 leaves
  - by deleting one of the leaves and its incident edge, a tree T_2 with n nodes is obtained
  - by induction hypothesis, T_2 has $n - 1$ edges; therefore, T_1 has $n - 1 + 1 = n$ edges

□

Property 2.2. Any pair of nodes in a tree is connected via a unique path. Otherwise, the tree would contain a cycle.

Property 2.3. By adding a new edge to a tree, a new unique cycle is created. This cycle consists of the path created in Property 2.2 and the new edge.

Property 2.4. Let $G_T = (N, T)$ be a spanning tree of $G = (N, E)$. Consider an edge $e \notin T$ and the unique cycle C of $T \cup \{e\}$. For each edge $f \in C \setminus \{e\}$, the subgraph $T \cup \{e\} \setminus \{f\}$ is a spanning tree of G.

(a) Graph G_T, edge f is red, edge e is blue

(b) $T \cup \{e\} \setminus \{f\}$

Figure 5: Exchange property

Property 2.5. Let F be a partial tree *(spanning nodes in $S \subseteq N$)* contained in a optimal spanning tree of $G = (N, E)$. Consider $e = \{u, v\} \in \delta(S)$ of minimum cost, then there exists a minimum cost spanning tree of G containing e.

Proof. By contradiction, assume $T^* \subseteq E$ is a minimum cost spanning tree with $F \subseteq T^*$ and $e \notin T^*$.
Adding an 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 a minimum cost spanning tree of G as it has the same cost as T^*
- If $c_e < c_f$, then $c\left(T^* \cup \{e\} \setminus \{f\}\right) < c\left(T^*\right)$, hence T^* is not optimal

□

2.4 Optimal cost spanning tree

Spanning trees have a number of applications, including:

- **network** design
- **IP network** protocols
- **compact memory** storage

Model an undirected graph $G = (N, E)$, $n = |N|$, $m = |E|$ and a cost function $c : E \to \mathbb{R}$, that assigns a cost to each edge, with $e = \{u, v\} \in E$.

Required properties

1. Each **pair of nodes** must be in a **path**
   $\Rightarrow$ the output must be a **connected subgraph** containing all the nodes N of G
2. The **subgraph** must have **no cycles**
   $\Rightarrow$ the output must be a **tree**

Problem 2.2 (Problem definition)**.** Given an undirected graph $G = (N, E)$ and a cost function $c : E \to \mathbb{R}$, find a spanning tree $G_T(N, T)$ of G of minimum, total cost.
The objective is finding:

$$\min_{T \in X} \sum_{e \in T} c_e \qquad X = \text{ set of all spanning trees of } G$$

Theorem 2.1. *A complete graph with n nodes ($n \geq 1$) has n^{n-2} spanning trees.*

Property 2.6. Every spanning tree of a connected n-node graph has $n - 1$ edges.

2.4.1 Prim's algorithm

Idea: iteratively build a spanning tree.

Method

1. Start from initial tree (S, T) with $S = \{u\}, S \subseteq N$ and $T = \emptyset$
2. At each step, add to the current partial tree (S, T) an edge of minimum cost among those which connect a node in S to a node in $N \setminus S$

Goal

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

Complexity if all edges are scanned at each iteration, the complexity order is $\mathcal{O}(nm)$

Algorithm the pseudocode of the algorithm is shown in Code 2. 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.

2.4.1.1 Correcteness of Prim's algorithm

Proposition 2.1. 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 $\delta(S)$. Each selected edge is part of the optimal solution as it belongs to a minimum spanning tree.
The optimality condition allows to verify whether a spanning tree T is optimal or not; it suffices to check that each $e \in E \setminus T$ is not a cost decreasing edge.

```
S := {u}
T := {}
while |T| < n - 1 do
  {u, v} := edge ∈ δ(S) with minimum cost // u ∈ S, v ∈ N \ S
  S := S ∪ {v}
  T := T ∪ {{u, v}}
end
```

Code 2: Prim's algorithm

2.4.1.2 Implementation in quadratic time

The Prim's algorithm can be implemented in quadratic time, i.e. $\mathcal{O}\left(n^2\right)$.
Data structure

- k number of edges selected so far
- Subset $S \subseteq N$ of nodes incident to the selected edges
- Subset $T \subseteq E$ of selected edges
- $C_j = \begin{cases} \min\{c_{ij} \mid i \in S\} & j \notin S \\ +\infty & \text{otherwise} \end{cases}$
- $closest_j = \begin{cases} \arg\min\{c_{ij} \mid i \in S\} & j \notin S \\ \text{predecessor of } j \text{ in the minimum spanning tree} & j \in S \end{cases}$

An example of a step is shown in Figure 6.

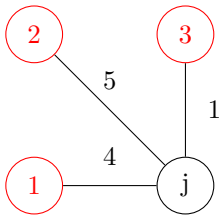


Figure 6: Data structure
nodes $1, 2, 3 \in S$, node $j \notin S$
$\text{closest}_j = 3 \quad c_{\text{closest}_j j} = 1$

The spanning tree is built is built by selecting the node j with minimum cost C_j and adding the edge $\{j, closest_j\}$ to the spanning tree.

The code for this algorithm is shown in Code 3.

The complexity of this algorithm is $\mathcal{O}\left(n^2\right)$. For sparse graphs, where $m \ll \frac{n(n-1)}{2}$, a more efficient implementation ($\mathcal{O}\left(m \log\left(n\right)\right)$) *(using priority queues)* is possible.

2.4.2 Kruskal's algorithm

Kruskal's algorithm has the same Goal of Prim's algorithm but with a different approach; like the latter, is greedy.

```
T := {}
S := {u}
// initialization
for j ∉ N \ S do
  if {u, j} ∈ E then
    C_j := c_{u, j}
  else
    C_j := +\infty
  end
  closest_j := u
end
for k := 1 to n - 1 do
  min := +∞ // selection of min cost edge
  for j := 1, ..., n do
    if j ∉ S and C_j < min then
      min := C_j
      v := j
    end
  end
  S := S ∪ {v} // extend S
  T := T ∪ {{v, closest_v}} // extend T
  for j := 1 to n do
    if j ∉ S and c_vj < C_j then
      C_j := c_vj
      closest_j := v
    end
  end
end
```

Code 3: Prim's algorithm in quadratic time

```
F := {}
S := {e_1, ..., e_m}, e ∈ E, sorted by cost in non-decreasing order
i := 1
while |F| < n - 1 do
  if the two endpoints of e_i are in different trees then
    F := F ∪ {e_i}
    merge the two trees containing the two endpoints
  end
  i := i + 1
end15
```

Code 4: Kruskal's algorithm

Method

1. create a **forest** F where each vertex is a separate tree
2. create a **set** S containing all the edges in the graph
3. while S is **non empty** and F is not spanning:
   (a) **remove an edge** with minimum cost from S
   (b) if the removed edge connects two different trees then **add it to the forest** F, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

Complexity the complexity of the algorithm is $\mathcal{O}(E \log)(E)$ with E the number of edges in the graph.

Algorithm the Kruskal's algorithm is implemented in Code 4.

2.4.3 Optimality condition of a spanning tree

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

Theorem 2.2. *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 of graph G, found via by Prim's algorithm
- it can be verified that T^* can be iteratively *(changing one edge at a time)* transformed into T without changing the total cost
- thus, T is also optimal

□

2.5 Optimal paths

Optimal *(shortest, longest, ...)* paths have a wide range of applications, including:

- **Google Maps**, **GPS** navigators
- Planning and management of **transportation**, **electrical**, and **telecommunication networks**
- **Problem** planning

Problem 2.3 (Problem definition)**.** Given a directed graph $G = (N, A)$ with a cost $c_{ij} \in \mathbb{R}$ associated to each arc $(i, j) \in A$, and two nodes s and t, determine a minimum cost *(shortest)* path from s to t.

- Each **value** $c_{i,j}$ represents the **cost** *(or length, travel time, ...)* of arc $(i, j) \in A$
- Node s is the **origin** *(or source)*, node t is the **destination** *(or sink)*

Property 2.7. A path $\langle (i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k) \rangle$ is **simple** if no node is visited more than once

Property 2.8. If $c_{ij} \geq 0$ for all $(i, j) \in A$, there is at least one shortest path that is simple.

2.5.1 Dijkstra's algorithm

Idea: consider the nodes in increasing order of length *(cost)* of the shortest path from s to any one of the other nodes.

Method

- To each **node** $j \in N$, a **label** L_j is associated
  $\Rightarrow$ at the end of the algorithm, this label will be the cost of the minimum cost path from s to j
- Another label $predecessor_j$ is associated to each node $j \in N$
  $\Rightarrow$ at the end of the algorithm, this label will be the node that precedes j on the minimum cost path from s to j
- Make a **greedy** choice with respect to the paths from s to j
- A set of **shortest paths** from s to any node $j \notin s$ can be retrieved backwards from t to s iterating over the predecessors

Goal

$\rightarrow$ *Input*: graph $G = (N, A)$, cost $c_{ij} \geq 0 \, \forall \, i, j$, origin $s \in N$
$\rightarrow$ *Output*: shortest path from s to all other nodes in G

Data structure

- $S \subseteq N$: subset of nodes whose labels are permanent
- $X \subseteq N$: subset of nodes with temporary labels
- $L_j = \begin{cases} \text{cost of a shortest path from } s \text{ to } j & j \in S \\ \min\{L_i + c_{ij} \mid (i, j) \in \delta^+(S) & j \notin S\} \end{cases}$
  $\rightarrow$ given a directed graph G and the current subset of nodes $S \subset N$, consider the outgoing cut $\delta^+(S)$ and select $(u, v) \in \delta^+(S)$ such that: $L_u + c_{uv} = \min\{L_i + c_{ij} \mid (i, j) \in \delta^+(S)\}$
  $\rightarrow$ thus: $L_u + c_{uv} \leq L_i + c_{ij}, \forall \, (i, j) \in \delta^+(S)$
- $\text{predecessor}_j = \begin{cases} \text{predecessor of } j \text{ in the shortest path from } s \text{ to } j & j \in S \\ u \text{ such that } L_u + c_{uj} = \min\{L_i + c_{ij} \mid i \in S\} & j \notin S \end{cases}$

Complexity the complexity of the algorithm depends on the how the arc (u, v) is selected among those of the current cut $\delta^+(u)$.

- If all m arcs are scanned, the overall complexity would be $\mathcal{O}(nm)$, hence $\mathcal{O}(n^3)$
- If all labels L_j are determined by appropriate updates (as in Prim's algorithm), only a single arc of $\delta^+(j)$ is scanned, hence the complexity is $\mathcal{O}(n^2)$

Notes

- A set of shortest paths from s to all the nodes $j \in N$ can be retrieved backwards from t to s iterating over the predecessors
- The union of a set of shortest paths from node s to all the other nodes of G is an arborescence rooted at s
- Dijkstra's algorithm does not work when there are arcs with negative cost: if G contains a circuit of negative cost, the shortest path problem may not be well defined

The code for this algorithm is shown in Code 5.

```
S := {}
X := {s}
for u ∈ N do
  L_u := ∞
end
L_s := 0
while |S| < |N| do
  u := argmin{L_i | i ∈ X}
  X := X \ {u}
  S := S ∪ {u}
  for (u, v) ∈ δ+(u) do
    if L_v > L_u + c_uv then
      L_v := L_u + c_uv
      predecessor_v := u
      X := X ∪ {v}
    end
  end
end
```

Code 5: Dijkstra's algorithm

2.5.1.1 Correcteness of Dijkstra's algorithm

Proposition 2.2. Dijkstra's algorithm is correct.

Proof.

1. A the k-th step:
   - $S = \{s, i_1, \dots, i_{k-1}\}$
   - $\begin{cases} \text{cost of a minimum cost path from } s \text{ to } j & j \in S \\ \text{cost of a minimum cost path with all intermediate nodes in } S & j \notin S \end{cases}$
2. By induction on the number k of steps:
   - base case: for $k = 1$ the statement holds, since
     $$S = \{s\}, \quad L_s = 0, \quad L_j = +\infty, \quad \forall\, j \notin S$$
   - inductive step: assume that the statement holds for $k + 1$
     - let $u \notin S$ be the node that is inserted in S and ϕ the path from s to u such that:
       $$L_v + c_{vu} \leq L_i + c_{iu}, \quad \forall\, (i, v) \in \delta^+(S)$$
     - every path π from s to u has $c(\pi) \geq c(\phi)$, as there exists $i \in S$ and $j \notin S$ such that:
       $$\pi = \pi_1 \cup \{(i, j)\} \cup \pi_2$$
       where (i, j) is the first arc in $\pi \cap \delta^+(S)$
     - it holds that
       $$c(\pi) = c(\pi_1) + c_{ij} + c(\pi_2) \geq L_i + c_{ij}$$
       because $c_{ij} \geq 0 \Rightarrow c(\pi_2) \geq 0$ and by the choice of (v, u), $c(\pi_1) \geq L_i$
     - finally, by induction assumption:
       $$L_i + c_{ij} \geq L_v + c_{vu} = c(\phi)$$
     - a visualization of this step of the proof is shown in Figure 7

□

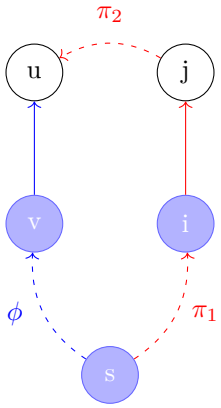


Figure 7: Proof of the induction step; nodes s, v, i are in cut S

2.5.1.2 Example of Dijkstra's algorithm

An example of Dijkstra's algorithm is shown in Figure 8.

2.5.2 Floyd-Warshall's algorithm

Goal

→ *Input*: a directed graph $G = (N, A)$ with an $n \times n$ cost matrix $C = [c_{ij}]$
→ *Output*: for each pair of nodes $i, j \in N$, the cost c_{ij} of the shortest path from i to j

Data structure

- Two $n \times n$ matrices D, P whose elements correspond, at the end of the algorithm, to:
  - d_{ij} the cost of the shortest path from i to j
  - p_{ij} the predecessor of j on the shortest path from i to j

Method

1. **Initialization** of D and P:

$$p_{ij} = i \quad \forall i$$

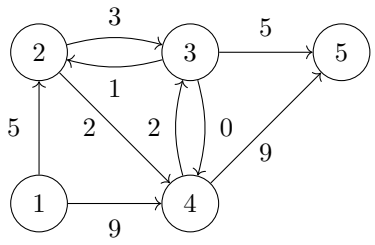
$$d_{ij} = \begin{cases} 0 & i = j \\ c_{ij} & i \neq j \wedge (i,j) \in A \\ +\infty & \text{otherwise} \end{cases}$$

2. **Triangular** operation: for each pair of nodes i, j, where $i \neq u, j \neq u$, check whether the path from i to j is shorter by going through u (i.e. $d_{iu} + d_{uj} < d_{ij}$)

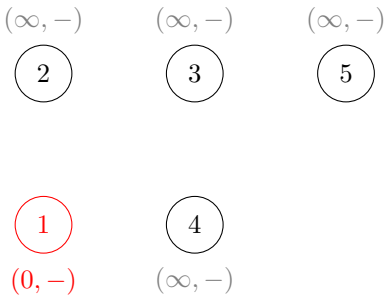
Complexity

- Since in the worst case the triangular operation is executed for all nodes u ad for each pair of nodes i, j, the complexity is $\mathcal{O}\left(n^3\right)$

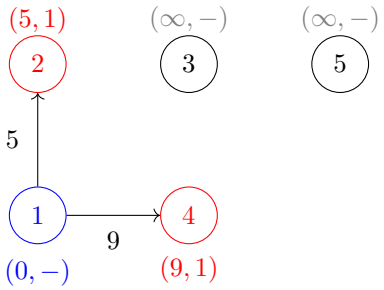
The code for this algorithm is shown in Code 6.



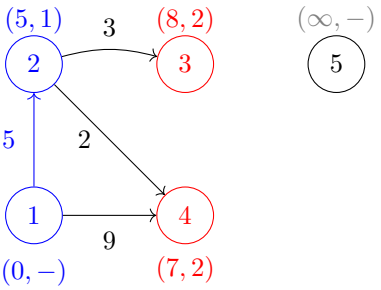
(a) Sample graph, with the cost of each arc



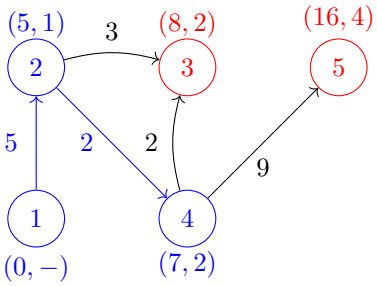
(b) step 1 of Dijkstra's algorithm



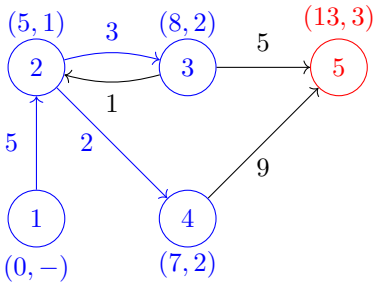
(c) step 2 of Dijkstra's algorithm



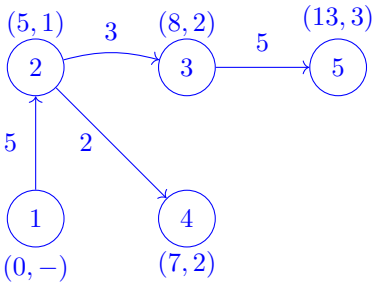
(d) step 3 of Dijkstra's algorithm



(e) step 4 of Dijkstra's algorithm



(f) step 5 of Dijkstra's algorithm



(g) step 6 of Dijkstra's algorithm

Figure 8: Example of Dijkstra's algorithm

```
for j := 1 to n do
  p_id := i
  if i = j then
    d_ij := 0
  else if (i, j) in A then
    d_ij := c_ij
  else
    d_ij := +∞
  end
end
for u ∈ N do
  for i ∈ N \{ u } do
    for j ∈ N \{ u }
      if d_iu + d_uj < d_ij then
        p_ij := p_uj
        d_ij := d_iu + d_uj
      end
    end
  for i ∈ N do
    if d_ij < 0 then
      error "negative cycle"
    end
  end
end
```

Code 6: Floyd-Warshall's algorithm

2.5.2.1 Correctness of Floyd-Warshall's algorithm

Proposition 2.3. Floyd-Warshall's algorithm is correct.

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, j)* corresponds to the cost of a shortest path from i to j with at most u intermediate nodes ($\{1, \dots, u\}$) □

2.6 Optimal paths in directed, acyclic graphs

A directed graph $G = (N, A)$ is **acyclic** if it does not contain any circuit. A directed acyclic graph G is referred to as a *DAG*.

Property 2.9 (Topological ordering)**.** The nodes of any *DAG* G can be ordered topologically, i.e. indexed so that for each arc $(i, j) \in A$ the index of i is less than the index of j *($i \leq j$).*

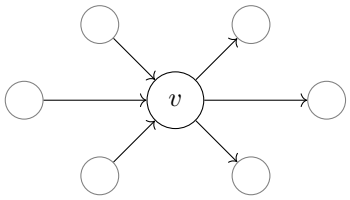
The topological order can be exploited by dynamic programming algorithms to compute efficiently the shortest paths in a *DAG*.

Problem 2.4 (Problem description)**.** Given a *DAG* $G = (N, A)$ with a cost $c_{ij} \in \mathbb{R}$ and nodes s, t, determine the shortest *(or longest)* path from s to t.

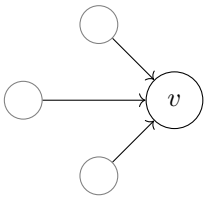
2.6.1 Topological ordering method

The method requires $G = (N, A)$ to be a *DAG* represented via the list of predecessors $\delta^-(v)$ and the list of successors $\delta^+(v)$ of each node $v \in N$. Then, it works as follows:

1. **Assign** the smallest positive integer not yet assigned to a node $v \in N$ with $\delta^-(v) = \emptyset$
   → such node always exists because G does not contain circuits
2. **Delete** the node v with all its incident arcs
3. **Go** to step (1) until all nodes have been assigned a number



(a) Topological ordering



(b) Node v with $\delta^-(v) = \emptyset$, as in step (1) of the algorithm

Figure 9: Topological ordering method

This algorithm has complexity $\mathcal{O}(|A|)$, because each node is assigned a number only once. Furthermore, all arcs incident to a node are deleted only once.

2.6.2 Dynamic programming for shortest path in *DAGs*

Any **shortest path** from 1 to t, called π_t, with at least 2 arcs can be subdivided into two parts:

- π_i, the shortest **subpath** from s to i
- (i, t), the **remaining part**

This decomposition is called the **optimality principle of shortest paths in *DAGs***. An illustration of this decomposition is shown in Figure 10.

The strategy to find the shortest path is:

Figure 10: Shortest path from 1 to t

```
sort the nodes of G topologically
L_1 := 0
for j := 2 to n do
  L_j := min{L_i + c_{ij} | (i, j) ∈ δ⁻(j) ∧ i < j}
  pred_j := v such that (v, j) = argmin{L_i + c_{ij} | (i, j) ∈ δ⁻(j) ∧ i < j}
end
```

Code 7: Shortest path in *DAG*

1. For each node $i = 1, \ldots, t$ let L_i be the cost of a shortest path from 1 to i
   $\rightarrow$ $L_t = \min\limits_{(i,t)\in\delta^-(t)} \{L_i + c_{it}\}$
   $\rightarrow$ the minimum is taken over all possible predecessors i of t
2. If G is topologically ordered *DAG*, then the only possible predecessors of t in a shortest path π_t from 1 to t are those with index $i < t$
   $\rightarrow$ $L_t = \min\limits_{i<t} \{L_i + c_{it}\}$
   $\rightarrow$ in a graph with circuits, any node i can be a predecessor of t if $i \neq t$

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

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

which can be solved in reversed order

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

Algorithm finally, the algorithm is shown in pseudocode in Code 7.

Complexity of the algorithm is $\mathcal{O}(|A|)$:

- Topological ordering of the nodes: $\mathcal{O}(m)$ with $m = |A|$ *(number of arcs)*
- Each node/arc is processed only once: $\mathcal{O}(n + m)$

In order to find the longest path, the algorithm can be adapted as follows:

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

2.6.2.1 Optimality of the algorithm

The Dynamic Programming algorithm for finding shortest or longest paths in *DAGs* is exact. This is due to the optimality principle, already explored in the previous section.

2.7 Project planning

Problem 2.5 (Problem definition)**.** A project consists of a set of m activities with their (estimated) duration: activity A_i has duration $d_i \geq 0, i = 1, \ldots, m$. Some pair of activities allow a precedent constraint: $A_i \propto A_j$ indicated that A_i must be performed before A_j.
A project can be represented by a directed graph $G = (N, A)$ where:

- each **arc** corresponds to an **activity**
- the **arc length** represent the **duration** of the corresponding activity

In order to account for precedence constraints, the arcs must be positioned such that for activities $A_i \propto A_j$ there exists a directed path where the arc associated to A_i precedes the arc associated to A_j. Such notation is shown in Figure 11.

Figure 11: Precedence relation in project planning

Therefore, a nove v marks an event corresponding to the end fo all the activities $(i, v) \in \delta^-(v)$ and the *(possible)* start of all the activities $(v, j) \in \delta^+(v)$.

Property 2.10. The directed graph G representing a project is acyclic *(is a DAG)*.

Proof. by contradiction, if $A_{i1} \propto A_{12} \propto \ldots \propto A_{jk} \propto A_{kj}$ there would be a logical inconsistency. □

2.7.1 Optimal paths

A graph G can be simplified by contracting some arcs, but it's important to not introduce unwanted precedence constraints. Artificial nodes or artificial arcs are introduced 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 the **same origin and destination**

Problem 2.6 (Problem description)**.** Given a project *(set of activities with duration and precedence constraints)*, schedule the activities in order to minimize the overall project duration *(the time needed to complete all the activities)*.

Property 2.11. The minimum overall project duration is the length of a longest path from s to t in the graph G.

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

2.7.1.1 Critical path method - *CPM*

The critical path method *(CPM)* determines:

- A **schedule** *(a plan for executing the activities specifying the order and the assigned time)* that minimizes the overall project duration
- The **slack** of each activity *(the amount of time by which its execution can be delayed without affecting the overall minimum project duration)*

```
sort the nodes topologically
T_min_i := 0
for j = 2 to n do
  T_min_j := max{T_min_i + d_ij | (i, j) ∈ δ⁻(j)}
end
T_max_n := T_min_n // minimum project duration
for i = n-1 to 1 do
  T_max_i := min{T_max_j - d_ij | (i, j) ∈ δ⁺(i)}
end
```

Code 8: Critical path method

Initialization construct the graph G representing the project.

Method

1. Find a topological **order of the nodes**
2. 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

   $\rightarrow$ T_{min_h} corresponds to the minimum project duration
3. 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}
4. For each activity $(i, j) \in A$ find the **slack**

   $\rightarrow$ the slack is calculated as $\sigma ij = T_{max_j} - T_{min_i} - d_{ij}$

Goal

$\rightarrow$ *Input:* graph $G = (N, A)$ with $n = |N|$ and the duration d_{ij} associated to each $(i, j) \in A$
$\rightarrow$ *Output*: $(T_{min_i}, T_{max_i}), i = 1, \ldots, n$

Algorithm finally, the algorithm is shown in pseudocode in Code 8.

Complexity the overall complexity is $\mathcal{O}(n + m) \approx \mathcal{O}(m)$, due to the sum of:

- complexity of the topological sort - $\mathcal{O}(n + m)$
- complexity of the first loop - $\mathcal{O}(n + m)$
- complexity of the second loop - $\mathcal{O}(n + m)$

2.7.1.2 Critical paths

An activity (i, j) with zero slack $\sigma_{ij} = T_{max_j} = T_{min_i} = d_{ij} = 0$ is called **critical**.
A critical path is a path in a $s - t$ composed uniquely by critical activities. At least one always exists.

2.7.1.3 Gantt charts

A **Gantt chart** is a graphical representation of a project schedule. It was introduced in *1896* by *Henry Gantt*, an American mechanical engineer and management consultant.
There are two types of Gantt charts:

- Gantt chart at **earliest** - each activity (i, j) starts at T_{min_i} and ends at $T_{min_i} + d_{ij}$
- Gantt chart at **latest** - each activity (i, j) starts at T_{max_i} and ends at $T_{max_i} + d_{ij}$

2.8 Network flows

Network flows problems involve the distribution of a given *product (such as water, gas, data)* from a set of *sources* to a set of *users* so as to optimize a given objective function *(e.g. minimize the total cost of the distribution)*.
It has many indirect applications, such as:

- **Telecommunication**
- **Transportation**
- **Logistics**

- A **network** is a directed and connected graph $G = (V, A)$ with a source $s \in V$ and a sing $t \in V$, with $s \neq t$, and a capacity $k_{ij} \geq 0$ for each arc $(i, j) \in A$.
- A **feasible flow** x from s to t is a vector $x \in \mathbb{R}^m$ with a component x_{ij} for each arc $(i, j) \in A$ satisfying the capacity constraint

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

  and the flow balance constraint at each intermediate node $u \in \{V \setminus \{s, t\}\}$:

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

- The value of **flow** x is

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

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

A problem related to flow is defined as follows *(Definition 2.1)*.

Definition 2.1 (flow Problem definition)**.** Given a network $G = (V, A)$ with an integer capacity k_{ij} for each arc $(i, j) \in A$, find a feasible flow x from s to t with maximum value.
If multiple sources or sink are present while only one product is considered, dummy notes s^* and t^* can be added *(as shown in Figure 12)*.

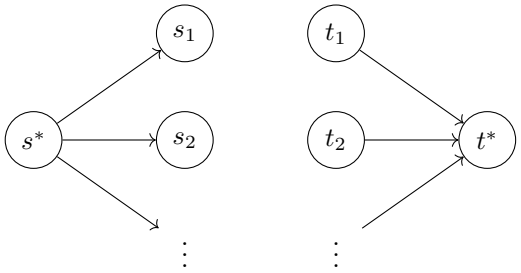


Figure 12: Multiple sources and sinks. $\delta^-(s^*) = \delta^+(t^*) = \emptyset$, k_{s^*i} = availability limit, $k_{jt^*} = \infty$

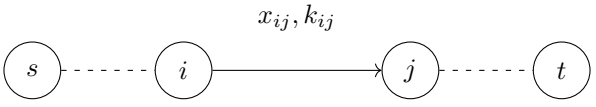


Figure 13: Graph representation of a flow

2.8.1 Linear programming model

The linear programming model *(refer to Section 3 for a in dept explanation of linear programming)* of the network flow is defined as:

$$\begin{aligned}
&\max \quad \phi \\
&\text{s.t.} \quad \sum_{(u,j)\in\delta^+(u)} x_{uj} + \sum_{(i,u)\in\delta^-(u)} x_{iu} \quad = \begin{cases} \phi & \text{if } u = s \\ -\phi & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\
&\qquad 0 \leq x_{ij} \leq k_{ij} \quad \forall\ (i,j) \in A \\
&\qquad \phi \in \mathbb{R}, x_{ij} \in \mathbb{R} \quad \forall\ (i,j) \in A
\end{aligned}$$

where ϕ denotes the value of the **feasible flow** x; ϕ is also the amount of product extracted from the source s.

- A **cut** separating s from t is $\delta(S)$ of $G,\ s \in S,\ t \in V \setminus S$
- There are 2^{n-2} cuts from s to t, with $n = |V|$
- The **capacity** of the cut $\delta(S)$ induced by S is:
$$k(S) = \sum_{(i,j)\in\delta^+(S)} k_{ij}$$
- Given a feasible flow x from s to t and a cut $\delta(S)$ separating them, the value of the **feasible flow through the cut** is
$$\phi(S) = \sum_{(i,j)\in\delta^+(S)} x_{ij} - \sum_{(i,j)\in\delta^-(S)} x_{ij}$$
with this notation, the value of the flow x is $\phi = \phi(\{s\})$

2.8.2 Properties of the flows

Property 2.12. Given a feasible flow x from s to t, for each cut separating $\delta(S)$ separating s from t:

$$\phi(S) = \phi(\{s\})$$

This Property is implied by the flow balance equations $\forall\, v\ \in V \setminus \{s,t\}$.

Property 2.13 (Weak duality)**.** For every feasible flow s from s to t and every cut $\delta(S),\ s \in V$ separating s from t:

$$\phi(s) \leq k(S) \quad \text{(value of the flow } \leq \text{ capacity of the cut)}$$

Proof. By definition of value of the flow through the cut $\delta(S$:

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

And, since $0 \leq x_{ij} \leq k_{ij}$ for all $(i,j) \in A$:

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

Finally, $\phi(S) \leq k(S)$. □

Consequence If $\phi(S) = k(S)$, for a subset $S \subseteq V,\ s \in S,\ t \notin S$, then x is a flow of maximum value and the cut $\delta(S)$ is of minimum capacity.
The property *(2.13)* for any feasible flow x and for any cut $\delta(S)$ separating s from t expresses a **weak duality relation** between the two problems of finding a maximum flow and a minimum cut.

- **Primal problem**: given $G = (V, A)$ with integer capacities on the arcs and $s, t \in V$, determine a feasible flow of maximum problem
- **Dual problem**: given $G = (V, A)$ with integer capacities on the arcs and $s, t \in V$, determine a cut *(separating s from t)* of minimum capacity separating s from t

2.8.3 Ford-Fulkerson algorithm

Idea start from a feasible flow x and try to iteratively increase its value ϕ by sending, at each iteration, an additional amount of product along a directed *(or undirected)* path from s to t with a strictly positive residual capacity.

- If (i,j) is **not saturated** *(i.e. $x_{ij} < k_{ij}$)*, then (i,j) is called a **residual arc** and x_{ij} can be increased
- If (i,j) is **not empty** *(i.e. $x_{ij} \geq 0$)*, then (i,j) is called a **reverse arc** and x_{ij} can be decreased while respecting the flow balance equations

An arc (i,j) in a $s-t$ cut, where $s \in S,\ t \in S'$, is called:

- **forward** if $s \in S, t \in S'$
- **backward** if $s \in S', t \in S$

The following Property *(2.14)* is the key to the Ford-Fulkerson algorithm.

Property 2.14 (augmenting path)**.** 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 and $x_{ij} > 0$ for any backward arc.

Then the algorithm works by sending δ additional units of product can be sent from s to t:

- $+\delta$ along forward arcs
- $-\delta$ along backward arcs

Given a feasible flow x for $G = (V, A)$, a **residual network** $\overline{G} = (V\overline{A})$ associated to x is defined as follows:

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

  $\rightarrow$ $\overline{k}_{ij}$ is called the **residual capacity** of (i,j)

At each iteration:

- Find an augmenting path P from s to t
- Send δ units of product along P
- Update the feasible flow x and the residual network $\overline{G}$

Goal

$\rightarrow$ *Input*: graph $G = (N.A)$ with capacity $k_{ij} > 0$ for any $(i,j) \in A,\ s,t \in N$
$\rightarrow$ *Output*: a feasible flow x from s to t of maximum value ϕ^*

The pseudocode is shown in Code 9.

```
x := 0
phi := 0
optimum := false
do
  build residual network G associated to x
  P := path from s to t in G
  if P is not defined then
    optimum := true
  else
    delta := min residual capacity of arcs in P
    phi := phi + delta
    for each arc (i, j) in P do
      if (i, j) is a forward arc then
        x_ij := x_ij + delta
```

```
      else
        x_ij := x_ij - delta
      end
    end
  end
until optimum = true
```

Code 9: Ford-Fulkerson algorithm

Complexity

- Since $\delta > 0$, the value ϕ increases at each iteration
- If $k_{ij} \in \mathbb{N} \forall\, i, j$, $x \in \mathbb{N} \overline{k}_{ij} \in \mathbb{N}$, and'' $\delta \geq 1$, then there are at most ϕ^* increases
- Since
$$phi^* \leq k(\{s\}) \leq mk_{max}, \quad m = |A|, \quad k_{max} = \max\{k_{ij} \mid (i,j) \in A\}$$
and each cycle is $\mathcal{O}(m)$, the overall complexity is $\mathcal{O}\left(mk_{max}^2\right)$.

Space complexity:

- The size of an instance I, written as $|I|$, is the number of bits needed to represent I
- Since $\lceil \log_2 i \rceil + 1$ bits are needed to store an integer i, $|I| = \mathcal{O}(m \log_2(k_{max}))$
- $\mathcal{O}\left(m^2 k_{max}\right)$ grows exponentially with $|I|$ because $k_{max} = 2^{\log_2 k_{max}}$???

2.8.3.1 Correcteness of the Ford-Fulkerson algorithm

Proposition 2.4. The Ford-Fulkerson algorithm is exact

Proof. A feasible flow x has a maximum value if and only if t is not reachable from s in the residual network $\overline{G}$ associated to x.

$\Rightarrow$ If exists an augmenting path, then x is not optimal
$\Leftarrow$ If t is not reachable from s in $\overline{G}$, then there is a cut of $\overline{G}$ such that $\delta^+_{\overline{G}}(S^*) = \emptyset$

By definition of $\overline{G}$, every arc $(i,j) \in \delta^+_{\overline{G}}(S^*)$ is saturated, while every arc $(i,j) \in \delta^-_{\overline{G}}(S^*)$ 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 Property *(2.13)*, $\phi(S^*) \leq k(S^*) \forall$ feasible, $\forall\, S \in V, s \in S, t \notin S$. Then, the flow x has maximum value and the cut induced by S^* has minimum capacity. □

2.8.3.2 Strong duality of the Ford-Fulkerson algorithm

The *Ford-Fulkerson* algorithm implies the following Theorem *(2.3)*.

Theorem 2.3 (Strong duality)**.** *The value of a feasible flow of maximum value is equal to the capacity of a cut of minimum capacity.*

Remarks

- If all the capacities $k_{ij} \in \mathbb{Z}^+$, the flow x of maximum value has all x_{ij} integer and an integer value ϕ^*
- *Ford-Fulkerson* algorithm is not greedy *(x_{ij} can be increased or decreased)*

2.8.3.3 Polynomial time algorithms for flow problems

More efficient algorithms exists, based on augmenting paths, pre flows and capacity scaling.

Idea Start from a feasible flow x of value ϕ and send, at each iteration, an additional amount of product in the residual network, respecting the residual capacities and the value ϕ, along cycles of negative cost.

2.8.4 Indirect applicaton - assignment problem

A common indirect application of the Ford-Fulkerson algorithm is the assignment (or matching) problem: Given m engineers, n tasks and for each engineer the list of tasks they can perform. Assign the task to each engineer such that:

- each engineer is assigned at most one task
- each task is assigned to at most one engineer
- the tasks assigned to each engineer are the ones they can perform

Graphical model bipartite graph of competences. An example is shown in Figure 14.

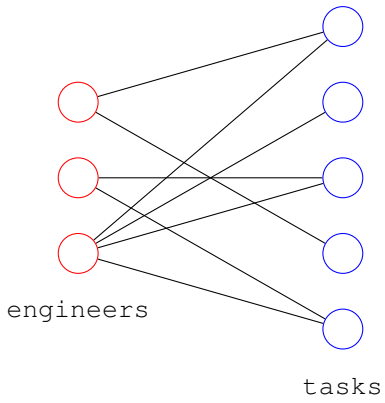


Figure 14: Example of assignment problem

Definition 2.2 (matching)**.** Given an undirected graph $G = (V, E)$, a *matching* is a subset $M \subseteq E$ such that:

- M is a set of edges
- each vertex of V is incident to at most one edge of M

Thanks to Definition *(2.2,)* the assignment problem can be solved by finding a maximum matching in the bipartite graph of competences; this equates to finding a feasible flow of maximum value from s to t in the bipartite graph of competences. An example of this problem is shown in Figure 15.
There is a correspondence between the feasible flows from source to sink of value ϕ and the matchings contained the edges of ϕ.

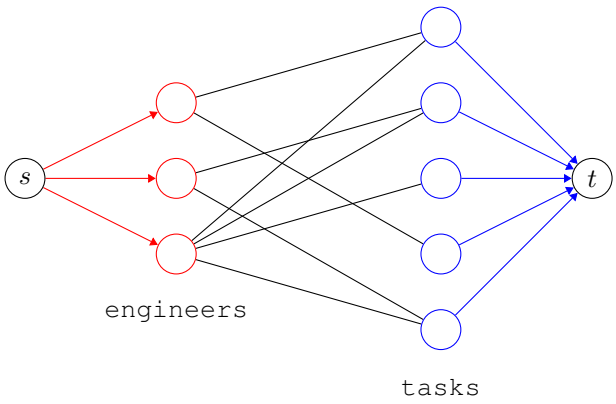


Figure 15: Example of assignment problem solved by flow

3 Linear Programming

3.1 Optimization problems

Optimization problems are problems that require to find the best solution among a set of possible solutions.

- An **instance** of an optimization problem is a pair (F, c) where:
  - F is the **domain** of feasible point
  - c is the **cost function**, a mapping $c : F \to \mathbb{R}$
  - the problem is finding an $f \in F$ such that $c(f) \leq c(y)\ \forall\, y \in F$
    - such point is called a **globally optimal** *(or just optimal)* solution to the given instance.
- An **optimization problem** is a set of I instances of a given optimization problem.

Definition 3.1. A **linear programming** *(or LP)* problem is an optimization problem such as

$$\begin{aligned} &\min\ f(x) \\ &\text{s.t. } x \in X \subseteq \mathbb{R}^n \to \mathbb{R} \end{aligned}$$

where:

- the **objective function** $f : X \to \mathbb{R}$ is **linear**
- the **feasible region** $X\,\{x \in \mathbb{R}^n \mid g_i(x)\, r_i\, 0 \wedge i \in \{1, \ldots, m\}\}$ with:

  $$r_i \in \{=, \geq, \leq\} \text{ and } g_i : \mathbb{R}^n \to \mathbb{R}$$

  are **linear** functions for $i \in \{1, \ldots, m\}$
- $x^* \in \mathbb{R}^n$ is an **optimal solution** of the *LP* 3.1 if $f(x^*) \leq f(x)\, \forall\, x \in X$

A wide variety of decision making problems can be formulated or approximated as *LP*, as they often involve the optimal allocation of a given set of limited resources to different activities.

Two forms of *LP* are commonly used:

- **General form** of a linear programming problem:

$$\begin{aligned} &\min\ z = c_1 x^1 + \cdot + c_n x_n \\ &\text{s.t. } a_{11} x^1 + \cdot + a_{1n} x_n\, (\leq, =, \geq)\, b_1 \\ &\quad \vdots \\ &\quad a_{m1} x^1 + \cdot + a_{mn} x_n\, (\leq, =, \geq)\, b_m \\ &\quad x^1, \ldots, x_n \geq 0 \end{aligned}$$

- **Matrix notation** of a linear programming problem:

$$\min z = \begin{bmatrix} c1 & c2 & \cdots & cn \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ \vdots \\ xn \end{bmatrix}$$

$$\text{s.t.} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x^1 \\ x^2 \\ \vdots \\ x_n \end{bmatrix} \{\leq, =, \geq\} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$\begin{bmatrix} x^1 \\ \vdots \\ x_n \end{bmatrix} \geq 0$$

3.2 Assumptions of *LP* models

The *LP* model is based on the following **assumptions**:

- **Linearity** *(proportionality and additivity)* of the objective function and constraints
  - → **proportionality**: *contribution of each variable = constant × variable*. It does not account for economies of scale
  - → **additivity**: *total contribution* $= \sum_i$ *contribution of each variable* i. It does not account for competing activities (their sum is not necessarily the total contribution)
- **Divisibility** of the variables, as they can assume fractional (rational) values
- **Parameters** are assumed to be **constants** which can be estimated with a sufficient degree of accuracy
  - → more complex mathematical programs are needed to account for uncertainty in the parameter values

LP **sensitivity analysis** allows to evaluate how *"sensitive"* an optimal solution is with respect to small changes in the parameters of the model.

3.3 Equivalent Forms

The **General Form** *(or canonical form) (Equation 3.1)* of a *LP* can be expressed in the equivalent **Standard Form** *(Equation 3.2)*.

$$\begin{aligned} \min(\max)\ & z = c^T x & & \\ \text{s.t. } & A_1 x \geq b_1 & & \text{inequality constraints} \\ & A_2 x \leq b_2 & & \text{inequality constraints} \\ & A_3 x = b_3 & & \text{equality constraints} \\ & x_j \geq 0,\ j \in J \subseteq \{1, \ldots, n\} & & \text{non-negativity constraints} \\ & x_j \text{ free},\ j \in \{1, \ldots, n\} \setminus J & & \text{free variables} \end{aligned} \tag{3.1}$$

$$\begin{aligned} \min\ & z = c^T x & & \\ \text{s.t. } & Ax = b & & \text{equality constraints} \\ & x \geq 0 & & \text{non negative variables} \end{aligned} \tag{3.2}$$

The two forms are equivalent, as simple transformation rules allow to pass from one form to the other; the transformations may involve adding and or deleting variables and constraints, as the next Section shows.

3.3.1 Transformation rules

- $\max c^T x \Rightarrow \min -c^T x$
- $a^T x \leq b \Rightarrow \begin{cases} a^T x + s = b \\ s \geq 0 \end{cases}$ s is a **slack variable**
- $a^T x \geq b \Rightarrow \begin{cases} a^T x - s = b \\ s \geq 0 \end{cases}$ s is a **surplus variable**
- x_j **unrestricted** in sign $\Rightarrow \begin{cases} x_j = x_j^+ - x_j^- \\ x_j^+ \geq 0 \\ x_j^- \geq 0 \end{cases}$

  $\rightarrow$ after the substitution, x_j is deleted from the problem
- $a^T x \leq q \Leftrightarrow -a^T x \geq -b$
- $a^T x \geq q \Leftrightarrow -a^T x \leq -b$
- $a^T x = b \Leftrightarrow \begin{cases} a^T x \leq b \\ a^T x \geq b \end{cases} \Leftrightarrow \begin{cases} a^T x \leq b \geq b \\ -a^T x \geq -b \end{cases}$

3.4 Graphical solutions

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

$$\{x \in \mathbb{R}^n \mid f(x) = z\}$$

Consider a *LP* with inequality constraints *(as it's easier to visualize)*.

- A **hyperplane** is the set of points that satisfy the constraint $H = \{x \in \mathbb{R}^n \mid a^T x = b\}$
- An **affine half space** is the set of points that satisfies the constraint $H = \{x \in \mathbb{R}^n \mid a^T x \leq b\}$
  - $\rightarrow$ each inequality constraint $a^T x \leq b$ defines an affine half space in the variable space
  - $\rightarrow$ in R^2, an affine half space is a is a **half plane**
- The **feasible region** X of any *LP* is a polyhedron P defined by the intersection of a finite number of affine half spaces
  - $\rightarrow$ P can be **empty** or **unbounded**
- A subset $S \subseteq \mathbb{R}^n$ is **convex** if for each pair of points $y^1, y^2 \in S$ the line segment between y^1 and y^2 is contained in S
  - $\rightarrow$ given two points $y^1, y^2 \in S$, a **convex combination** of them is any point of the form

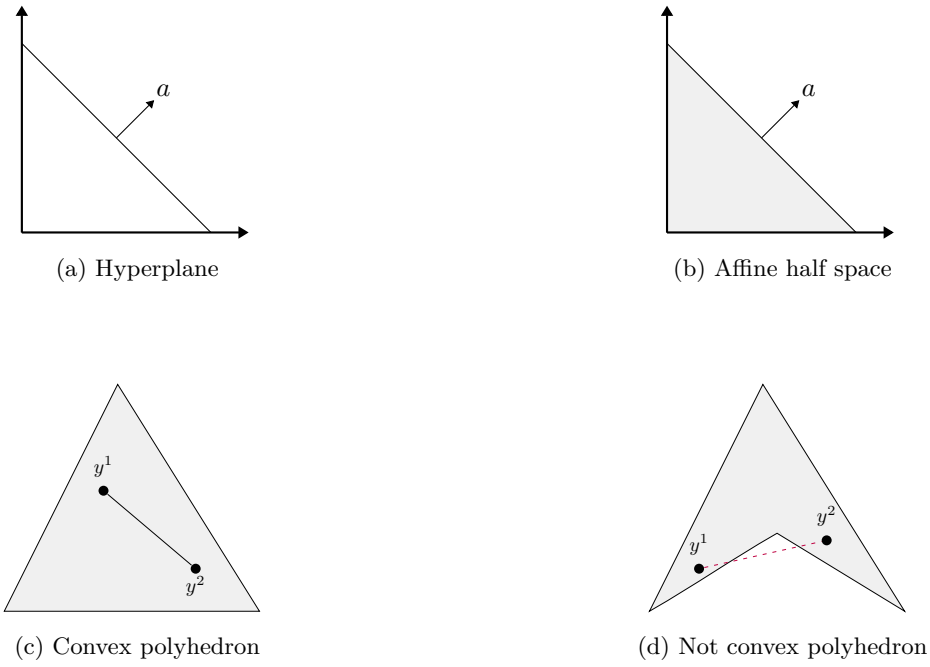
    $$z = \lambda y^1 + (1 - \lambda)\, y^2 \quad 0 \leq \lambda \leq 1$$

  - $\rightarrow$ a convex combination with $\lambda \neq 0, 1$ is called **strict**
  - $\rightarrow$ a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is **convex** if the space above the graph of f is convex
- The segment defined by all the convex combinations of y^1 and y^2, $y^1, y^2 \in S$, is called a **convex hull**
  - $\rightarrow$ $\left[y^1, y^2\right] = \left\{x \in \mathbb{R}^n \mid x = \alpha y^1 + (1 - \alpha)\, y^2 \wedge \alpha \in [0, 1]\right\}$
- A **polyhedron** P is a convex set of $\mathbb{R}^n$
  - any **half space** is **convex**
  - the **intersection** of a finite number of convex sets is also a **convex** set
- A **vertex of polyhedron** P is a point op P that cannot be expressed as a convex combination of other points of P
  - x is a vertex P iff

    $$x = \alpha y^1 + (1 - \alpha) y^2, \alpha \in [0, 1] \quad y^1, y^2 \in P \Rightarrow x = y^1 \vee x = y^2$$

- a non empty polyhedron $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ has a finite number ($n > 1$) of vertices

- A **polytope** is a bounded polyhedron
- Let x be an element of a polyhedron P. A vector $d \in \mathbb{R}^n$ is said to be a **feasible direction** at x if there exists $\theta \in \mathbb{R}, \theta > 0$ such that $x + \theta d \in P$.
- Given a polyhedron P, a vector $d \in \mathbb{R}^n, d \neq \overline{0}$ is an **unbounded feasible direction** of P if, for every point $x^0 \in P$, the ray $\left\{x \in \mathbb{R}^n \mid x = x^0 + \lambda d, \lambda \geq 0\right\}$ is contained in P



(a) Hyperplane

(b) Affine half space

(c) Convex polyhedron

(d) Not convex polyhedron

Figure 16: Illustrations of *LP* geometry definitions

3.4.1 Weyl-Minkoswki Theorem

The **Weyl-Minkoswki Theorem** describes the representation of a polyhedron. Its formulation is expressed in the *Theorem 3.1.*

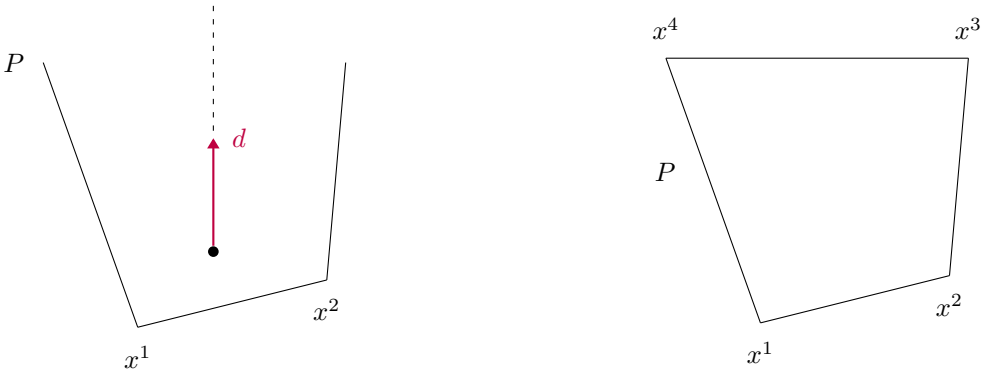
Theorem 3.1 (Weyl-Minkoswki)**.** *Every point x of a polyhedron P can be expressed as a convex combination of its vertices $x^1, \ldots, x^k$ plus (if needed) an unbounded **feasible direction** d of P:*

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

where the multipliers $\alpha_i \geq 0$ satisfy the constraint $\sum_{i=1} \alpha_i = 1$.

The unbounded feasible direction is needed if the polyhedron is unbounded; Figure 17 represent the cases of a bounded and an unbounded polyhedron.

Consequences: every point x of polytope P can be expressed as a convex combination of its vertices. Then the *Weyl-Minkowski theorem* can be used to describe any point. An example of this is shown in Figure 18.



(a) Unbounded polyhedron

(b) Bounded polyhedron

Figure 17: Illustration of the *Weyl-Minkowski* theorem

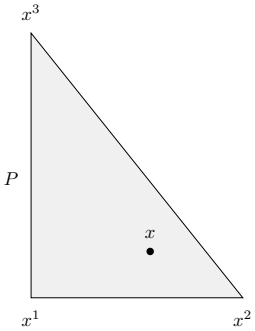


Figure 18: Example of polytope: $x = \alpha_1 x^1 + \alpha_1 x^2 + \alpha_3 x^3$ with $\alpha_1 + \alpha_2 + \alpha_3 = 1, \alpha_i \geq 0, d = 0$

3.4.2 Geometry of *LP*

Geometrically:

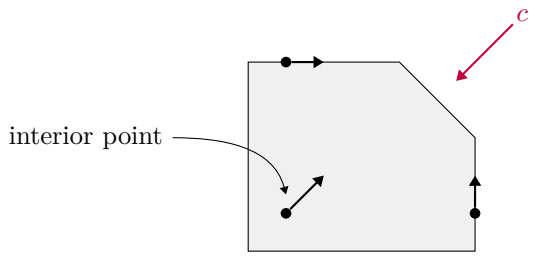
- An **interior point** $x \in P$ cannot be an optimal solution of the problem
  - it always exists an improving direction
  - consider Figure 19a where c represents the direction of fastest increase in z *(constant gradient)*
- In an optimal vertex, all **feasible direction are worsening directions**
  - consider Figure 19b where c represents the direction of fastest increase in z *(constant gradient)*
- The *Weyl-Minkowski* theorem implies that, although the variables can assume fractional values, any *LP* can be seen as a **combinatorial problem**
  - *"only"* the vertices of the polyhedron have to be considered in order to find the feasible solutions
  - the graphical method in only applicable for $n \leq 3$
  - the number of vertices often grows exponentially with respect to the number of variables

Furthermore, let P be a convex polytope of dimension d and let HS be a half space of P defined by hyperplane H. If the intersection $f = P \cap HS$ is a subset of H then:

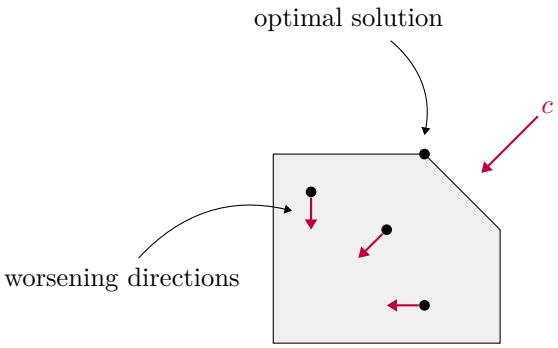
- f is a **face** of P
- H is the **support hyperplane** of f

The face is then defined according to its dimension:

- a **vertex** is a face of dimension 0 *(a point)*
- a **facet** is a face of dimension $d - 1$ *(a hyperplane)*
- an **edge** is a face of dimension 1 *(a line)*



(a) Interior point of polyhedron P



(b) Worsening directions of polyhedron P

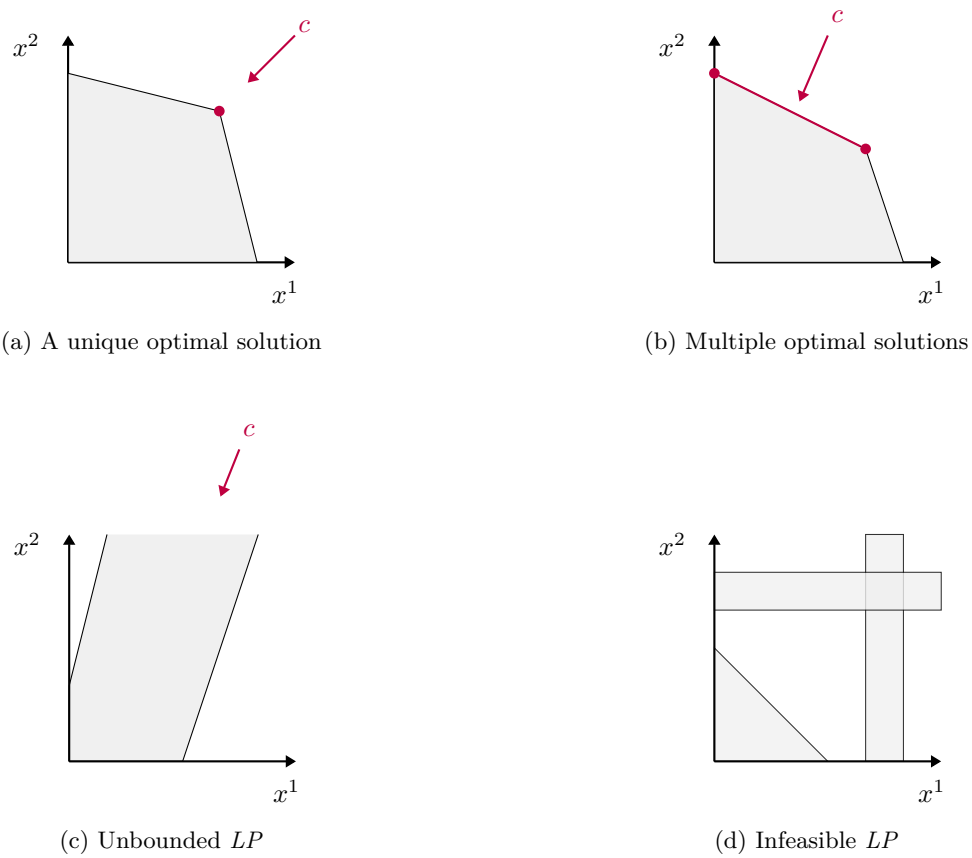
3.4.3 Four types of *LP*

There are four types of *LP*, depending on the number of solutions; all are illustrated in Figure 20. Since the objective of the problem is to minimize $f(x)$ *(as it's in form* $\min c^T x$*)*, better solutions are found by moving along the direction $-c$ *(the opposite of the gradient* $\nabla f(x)$*)*.

1. **A unique** optimal solution, Figure 20a
2. **Multiple** *(infinitely many)* optimal solutions, Figure 20b
3. **Unbounded** *LP*, Figure 20c

- this type of problem has unbounded polyhedron and unlimited objective function values

4. **Infeasible** *LP*, Figure 20d
   - this type of problem has an empty polyhedron and no feasible solution



(a) A unique optimal solution

(b) Multiple optimal solutions

(c) Unbounded *LP*

(d) Infeasible *LP*

Figure 20: Four types of *LP*

3.4.4 Basic feasible solutions and polihedra vertices

Due to the fundamental theorem of Linear Programming *(Theorem 3.1)*, to solve any *LP* problem it suffices to consider the *(finitely many)* vertices of the polyhedron P of feasible solutions; since the geometrical definition of vertex cannot be exploited algorithmically, an algebraic definition is needed.
A vertex corresponds to the intersection of the hyperplanes associated to n inequalities. If a polyhedron is expressed in standard form

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

it is possible to transform it into a inequality

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

and later transform it back into standard form

$$P' = \{x \in \mathbb{R}^n \mid Ax - s = b, s \geq 0, x \geq 0\}$$

where P' is the polyhedron of feasible solutions of the original problem. Finally, by renaming

$$A \coloneqq [\, A \,|\, I \,], \quad x \coloneqq \left(x^T \,|\, s^T\right)$$

the system of equation is represented in matrix form, where A has m rows.

Property 3.1. For any polyhedron $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$, where A has m rows:

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

3.4.4.1 Optimality of extreme points

Theorem 3.2. *Consider the LP problem of minimizing $c^T x$ over a polyhedron P. Suppose that P has at least one extreme point and that there exists an optimal solution. Then, there exists and optimal solution which is an extreme point of P.*

The above Theorem 3.2 **applies to polyhedra in standard form**, as well as to bounded polyhedra *(as they don't contain a line).*

Theorem 3.3. *Consider the LP problem of minimizing $c^T x$ over a polyhedron P. Suppose that P has at least one extreme point. Then, either the optimal cost is equal to $-\infty$, or there exists an extreme point which is optimal.*

For a general *LP* problem. if the feasible set has no extreme point, then Theorem 3.3 does not apply; however, the problem can be transformed into an equivalent one in standard form, establishing the following Corollary *(3.4).*

Corollary 3.4. *Consider the LP problem of minimizing $c^T x$ over a polyhedron P. Suppose that P has at least one extreme point. Then, either the optimal cost is equal to $-\infty$, or there exists an optimal solution.*

3.4.5 Algebraic characterization of vertices

Consider any polyhedron $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$, in standard form.

Assumption $A \in \mathbb{R}^{m \times n}$ is such that $m \leq n$ of rank m *(i.e. A is full rank).* This is equivalent to assume that there are no redundant constraints.

Solutions

- If $m = n$, there is a unique solution of $Ax = b$ ($x = A^{-1}b$)
- If $m < n$, there are ∞^{m-n} solutions of $Ax = b$
  - the system has $n - m$ **degrees of freedom**
  - by fixing the degrees of freedom to 0, a **vertex** is obtained

  The **basis** of matrix A is a subset of m columns of A that are linearly independent and form an $m \times m$ non singular matrix B.

$$A = [\overbrace{B}^{m} \mid \overbrace{N}^{m-n}]$$

3.4.6 Basic solutions

Let $x^T = [\overbrace{x_B^T}^{m \text{ components}} \mid \overbrace{x_N^T}^{m-n \text{ components}}]$. Then any system $Ax = b$ can be written as $B_{x_B} + N_{x_N} = b$, and for any set of values of x_N, if B is not singular, then $x_B = B^{-1}(b - N_{x_N})$.

Definitions

- A **basic solution** is a solution obtained by setting $x_N = 0$ and, consequently, $x_B = B^{-1}b$
  - two basic solutions are called adjacent if there are $n-1$ linearly independent constraints active in both of them
- A **feasible solution** is any vector $x \geq 0$ such that $Ax = b$
- A feasible solution with $x_B \geq 0$ is a **basic feasible solution** with basis B
  - $\forall\, j \notin B,\ x_j = 0$ *(all the non zero variables are in B)*
  - a **basic feasible solution** has n linearly independent constants
  - the variables in x_B are the **basic variables** and those in x_N are non **basic variables**
  - by construction, $\left(x_B^T, x_N^T\right)$ satisfy $Ax = b$
- A basic feasible solution is **degenerate** if it contains at least one basic variable with value 0
  - a basis uniquely determines a basic solution
  - tow different bases may lead to the same basic solution

Theorem 3.5 (Basic feasible solutions)**.** *$x \in \mathbb{R}^n$ is a basic feasible solution if and only if x is a vertex of the polyhedron $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$*

A basic feasible solution is now on referred to as *BFS* for brevity.

Property 3.2 (Number of solutions)**.** At most, there exists one basic feasible solution for each choice of the $n-m$ non basic variables out of the n variables:

$$\#\text{ basic feasible solutions} \leq \binom{n}{n-m} = \frac{n!}{(n-m)!\,(n-(n-m))!} = \binom{n}{m}$$

3.5 Simplex method

The Simplex method provides a systematic way to find the optimal solution of a *LP* problem; given an *LP* in standard form

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

it examines a sequence of basic feasible solutions with non increasing objective function values, until an optimal solution is reached or the problem is found to be unbounded. At each iteration, the basic feasible solution is replaced by a new one that is closer to the optimal solution.
In other, *simpler*, words, it generates a path *(a sequence of adjacent vertices)* along the edges of the polyhedron P that leads to the optimal solution.
The simplex method is articulated in 3 steps:

1. Find an **initial vertex** or establish that the *LP* is **unbounded**
2. Determine whether the current vertex is **optimal**
3. Move from a current vertex to a **better adjacent vertex** *(in terms of objective function value)* or establish that the *LP* is unbounded

The solution found in step (3) may be optimal only in a local sense; in *LP*, however, local optimality implies global optimality thanks to the following Property *(3.3)*.

Property 3.3 (Optimality of a vertex)**.** If the *LP* problem requires minimizing *(or maximizing)* a convex function over a convex set, then the local optimum is also the global optimum.

3.5.1 Optimality condition of a LP solution

Given a *LP*

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

Suppose that $x \in P$ is a basic feasible solution and move away from it in the direction of a vector $d \in \mathbb{R}^n$; clearly, d should not immediately lead to a point outside P, otherwise the solution would not be feasible any more. Then the Definition 3.2 applies.

Definition 3.2 (feasible direction)**.** Let x be an element of polyhedron P. A vector $d \in \mathbb{R}^n$ is said to be a **feasible direction** at x if there exists a positive scalar θ such that $x + \theta d \in P$.

Let x be a basic feasible solution to the standard form problem, let $B(1), \ldots, B(m)$ the indices of the basic variables and let $B = \left[A_{B(1)}, \ldots, A_{B(n)}\right]$ be the corresponding basis matrix.
In particular, $x_i = 0$ for all non basic variables, while the vector $x_B = \left(x_{B(1)}, \ldots, x_{B(m)}\right)$ of basic variables if given by

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

By selecting a non basic variable x_j (initially at zero) and increasing its value to θ while keeping all the other non basic variables to zero, a new basic feasible solution $x + \theta d$ is obtained. Algebraically:

$$d_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

At the same time, the vector x_B changes to $x_B + \theta d_B$, where $d_B = (d_{B(1)}, \ldots, d_{B(m)})$ is the vector with those components of d that correspond to the basic variables.
Given that only the feasible solutions are relevant, it's necessary true that $A(x + \theta d) = b$ and since x is feasible, $Ax = b$. At the same time, since $\theta > 0$, it must hold

$$Ad = 0 \tag{3.3}$$

Since $d_j = 1$ and $d_i = 0 \, \forall \, i \neq j$:

$$0 = Ad = \sum_{i=1}^{n} A_i d_i = \sum_{i=1}^{m} A_{B(i)} d_{B(i)} = B d_B + A_j$$

due to the invertibility of the basis matrix B:

$$d_B = -B^{-1} A_j \tag{3.4}$$

The direction vector d will be referred thereon to as the j-th basic direction.
While the equality constraints are respected, the non negativity constraints are not. During this movement, all the non basic variables stay at zero, while x_j is incremented: the focus is then on the basic variables. Two cases are possible:

1. x is a non degenerate basic solution: $x_B > 0$, so $x_B + \theta d_B \geq 0$ and feasibility is maintained with θ sufficiently small. d is a **feasible direction** at x.
2. x is degenerate: it's possible that a basic variable $x_{B(i)}$ is zero, while the correspondent component $d_{B(i)}$ of $d_B = -B^{-1} A_j < 0$. In this case, following the j-th basic direction, the non negativity constraint is violated and the solution is not feasible.

Now let d be the j-th basic direction at x. The rate of change of the objective function along d is given by $c'_B d_B + c_j$, where $c_B = \left(c_{B(1)}, \ldots, c_{B(,)}\right)$.
Using Equation 3.4, the rate of change of the objective function along d is:

$$c_j - c'_B B^{-1} A_j$$

While quantity is important enough to warrant a Definition *(3.3)*, intuitively, c_j is the cost per unit increase in the variable x_j, and the term $-c'_B B^{-1} A_j$ is the cost of the compensating change in the basic variables necessitated by the constraint $Ax = b$.

Definition 3.3 (reduced costs)**.** Let x be a basic solution with an associated basis matrix B. Let c_B be the vector of costs of the basic variables. For each j, the **reduced costs** $\bar{c}_j$ of the non basic variable x_j are defined as

$$\bar{c}_j = c_j - c'_B B^{-1} A_j$$

Consider *Definition 3.3* for the case of a basic variable. Since B is the matrix $[A_{B(1)}, \dots, A_{B(m)}]$, then $B^{-1}[A_{B(1)}, \dots, A_{B(m)}] = I$ *(is the identity matrix).* In particular, $B^{-1}A_{B(i)}$ is the i-th column of the identity matrix, which is the i-th unit vectore_i.
Therefore, for every basic variable $x_{B(i)}$, the reduced cost is:

$$\bar{c}_{B(i)} = c_{B(i)} - c'_B B^{-1} A_{B(i)} = c_{B(i)} - c'_B e_i = c_{B(i)} - c_{B(i)} = 0$$

so the reduced cost of a basic variable is always zero. The next Theorem *(3.6)* shows that the reduced cost of a non basic variable is always non negative.

Theorem 3.6. *Let x be a basic feasible solution x associated with a basis matrix B, and let $\bar{c}$ be the corresponding vector of reduced costs. Then:*

1. *If $\bar{c} \geq 0$, then x is optimal*
2. *If x is optimal and non degenerate, then $\bar{c} = 0$*

Note that Theorem 3.6 allows the possibility that x is a degenerate solution and $\bar{c} < 0$ for some non basic index j. Furthermore, it states that in order to decide whether a non degenerate basic feasible solution is optimal, it's sufficient to check whether all the reduced costs are non negative, which is the same as examining $n - m$ basic directions; checking if the degenerate basic feasible solution x is optimal is not as simple.
Finally, the Definition 3.4 states the requirements for a matrix to be optimal.

Definition 3.4 (optimal matrix)**.** A matrix B is **optimal** if and only if it is a basis matrix for an optimal basic feasible solution, i.e. the following conditions are both satisfied:

- $B^{-1}b \geq 0$
- $\bar{c}' = c' - c'_B B^{-1} A \geq 0$

Remarks

→ For **maximization** problems, the condition is $\bar{c}_N \leq 0$
→ This optimality condition is **sufficient** but generally not necessary

3.5.2 General case

Let's first of all assume that every basic feasible solution is non degenerate; this assumption will be relaxed later on.
Suppose that the current basic feasible solution is x, with reduced costs $\bar{c}_j$ for each of the non basic variables. If all of them are non negative, then x is optimal *(thanks to Theorem 3.6)* and the algorithm can terminate.
Otherwise, if $\exists\, \bar{c}_j < 0$ for a non basic variable x_j, then the j-th basic direction d is a feasible direction of cost decrease; while moving along it, x_j becomes positive and the value of all other non basic variables stays constant at 0. This situation is described as *"entering the basis"*.
By moving away from x along d, the traced points are in form $x + \theta d$, where $\theta \geq 0$. Since costs decrease along the direction d, the goal is to move as far as possible; the farthest point will be $x + \theta^* d$, where

$$\theta^* = \max \{\theta \geq 0 \mid x + \theta d \in P\}$$

The associated cost change is

$$\Delta c = \theta^* c' d = \theta^* \bar{c}_j$$

It's then necessary to determine the closed formula for θ^*. Given that $Ad = 0$ *(Equation 3.3)*, then $A(x + \theta d) = Ax = b,\ \forall\, \theta$; $x + \theta d$ can become infeasible only if one of its component becomes negative, i.e. if $x_i + \theta a_{ij} \leq 0$. Two cases can be distinguished:

1. $d \geq 0$, then $x + \theta d \geq 0$ for all $\theta \geq 0$, the vector is never infeasible and $\theta^* = \infty$
2. $d_i \leq 0$ for some i, then $x_i + \theta d_i \geq 0 \Rightarrow \theta \leq -x_i/d_i$. Since this constraint must be satisfied for all i, the largest possible value of θ is

$$\theta^* = \min_{i=1,\dots,m} \left\{ -\frac{x_i}{d_i} \,\middle|\, d_i \leq 0 \right\} \tag{3.5}$$

Recalling that x_i is a non basic variable, then either x_i is the entering variable and $d_i = 1$, or else $d_i = 0$; in either case it's non negative and the formula can be re written as

$$\theta^* = \min_{i=1,\dots,m} \left\{ -\frac{x_{B(i)}}{d_{B(i)}} \,\middle|\, d_{B(i)} < 0 \right\} \tag{3.6}$$

Note that $\theta^* > 0$ since $x_{B(i)} > 0 \; \forall i$.

3.5.3 Changes of basis

Assumption: this Section refers to the case where the *LP* is a minimization problem.

Once θ^* has been chosen *(Equation 3.6)*, assuming that's it's finite, the new feasible solution $y = x + \theta^* d$ is obtained by moving along the basic direction d from x. Since $x_j = 0$ and $d_j = 1$, then $y_j = \theta^* > 0$.
Let l be a minimizing index of 3.6; the following conditions must hold:

$$-\frac{x_{B(l)}}{d_{B(l)}} = \min_{i=1,\dots,m} \left\{ -\frac{x_{B(i)}}{d_{B(i)}} \,\middle|\, d_{B(i)} < 0 \right\} = \theta^*$$
$$d_{B(l)} < 0$$
$$x_{B(l)} + \theta^* d_{B(l)} = 0$$

The basic variable $x_{B(l)}$ is now 0, while the non basic variable x_j is $y_j = \theta^* > 0$: the basis has changed, and the new basis is $\overline{B} = B \setminus \{x_{B(l)}\} \cup \{x_j\}$ *($x_{B(l)}$ has left the basis, x_j has entered the basis)*. The new matrix is now:

$$\overline{B} = \left[A_{B(1)} \mid \cdots \mid A_{B(l-1)} \mid A_j \mid A_{B(l+1)} \mid \cdots \mid A_{B(m)} \right]$$

Equivalently, the set of basic indices $B = \{B(1), \dots, B(m)\}$ can be updated as follows:

$$\overline{B}(i) = \begin{cases} B(i) & \text{if } i \neq l \\ j & \text{if } i = l \end{cases}$$

The previous steps can then be summarized in the following Theorem.

Theorem 3.7.

- *The columns $A_{B(i)}$, $i \neq l$ and A_j are linearly independent; $\overline{B}$ is a basis matrix*
- *The vector $y = x + \theta^* d$ is a basic feasible solution associated to the basis $\overline{B}$*

3.5.3.1 An iteration of the Simplex method

A single iteration of the Simplex method consists of the following steps:

1. start with a basis consisting of the basics columns $A_{B(1)}, \dots, A_{B(m)}$ and an associated basic feasible solution x

2. compute the reduced costs $\overline{c}_j$ of the non basic variables x_j

→ if the reduced costs are all non negative, then the current solution is optimal and the algorithm terminates

→ otherwise, choose a non basic variable x_j with reduced cost $\overline{c}_j \leq 0$

3. compute $u = B^{-1} A_j$ and the basic direction d associated to x_j

→ if no component of u is negative, then $\theta^* = \infty$, the optimal cost is $-\infty$ and the algorithm terminates

4. some components of u are negative, so θ^* can be computed

$$\theta^* = \min_{i=1,\dots,m} \left\{ -\frac{x_{B(i)}}{u_i} \,\middle|\, u_i < 0 \right\}$$

5. let l be a minimizing index of Step (4); form a new basis replacing $A_{B(l)}$ with A_j. If y is the new basic feasible solution, then $y_j = \theta^*$ and $y_{B(i)} = x_{B(i)} + \theta^* u_i, i \neq l$

The method is initialized with an arbitrary basic feasible solution x and the corresponding basis B; the following Theorem *(3.8)* states that the algorithm terminates after a finite number of iterations.

Theorem 3.8. *Assume that the feasible set is non empty and that every basic feasible solution is non degenerate. Then, the Simplex method terminates after a finite number of iterations; at termination, there are two possibilities:*

1. *an optimal basis B and the optimal solution x is associated to it are found*
2. *a vector d satisfying $Ad = 0, d \geq 0, c'd < 0$ is found, and the optimal cost is $-\infty$*

3.5.4 Pivoting operation

The pivot operation is a single elementary row operation that transforms the inverse of the previous basis into the inverse of the new basis; it's the same operation used in the Gaussian elimination method to solve systems of linear equations.

Given $Ax = b$:

1. Select a coefficient $\overline{a}_{rs} \neq 0$, the **pivot**
2. Divide the r-th row by $\overline{a}_{rs}$
3. For each row $i \neq r$ with $\overline{a}_{rs} \neq 0$, subtract the resulting r-th row multiplied by $\overline{a}_{is}$

This move does not affect the feasible solutions. Only a finite number of pivots exists for each basis.

An example of pivot operation is shown in Figure 21.

(a) Initial system (b) Pivot operation

Figure 21: Pivot operation example

3.5.5 Moving to an adjacent vertex

Goals

→ improve the objective function value
→ preserve the feasibility

The general description of the Simplex method allows some freedom: Step (2) allows to choose any j whose reduced cost $\overline{c}_j$ is negative and Step (5) allows to choose any l minimizing θ^*. Rules for making these choices are called **pivot rules**.

1. Which non basic variable **enters** the basis?
   - any one with reduced cost $\overline{c}_j < 0$
   - the one yielding the maximum $\Delta z = \theta^* \overline{c}_j$ with respect to the current basic feasible solution
   - **Bland's rule**: $s = \min\{j \mid \overline{c}_j < 0\}$
     $\rightarrow$ $\overline{c}_j > 0$ for maximization problems
2. Which basic variable **leaves** the basis?
   - **min ratio text**: index i with smallest positive ratio $\frac{x_{B_i}}{\overline{a}_{is}} = \theta^*$ among those with $\overline{a}_{is} > 0$
   - **Bland's rule**: $r = \min\left\{ i \,\middle|\, \frac{\overline{b}_i}{\overline{a}is} = \theta^* \, \overline{a}_{is} > 0 \right\}$
   - also randomly

Property 3.4 (Unboundedness)**.** If the objective function is unbounded, the algorithm will never stop: if $\exists \overline{c}_j < 0$ with $\overline{a}_{ij} \leq 0 \forall i$ no element of the j-th column can play the role of the pivot. This condition is verified if problem is **unbounded**.

3.5.6 Tableau representation

Tableau is a matrix representation of the LP problem. Let

$$z = c^T x$$
$$Ax = b$$

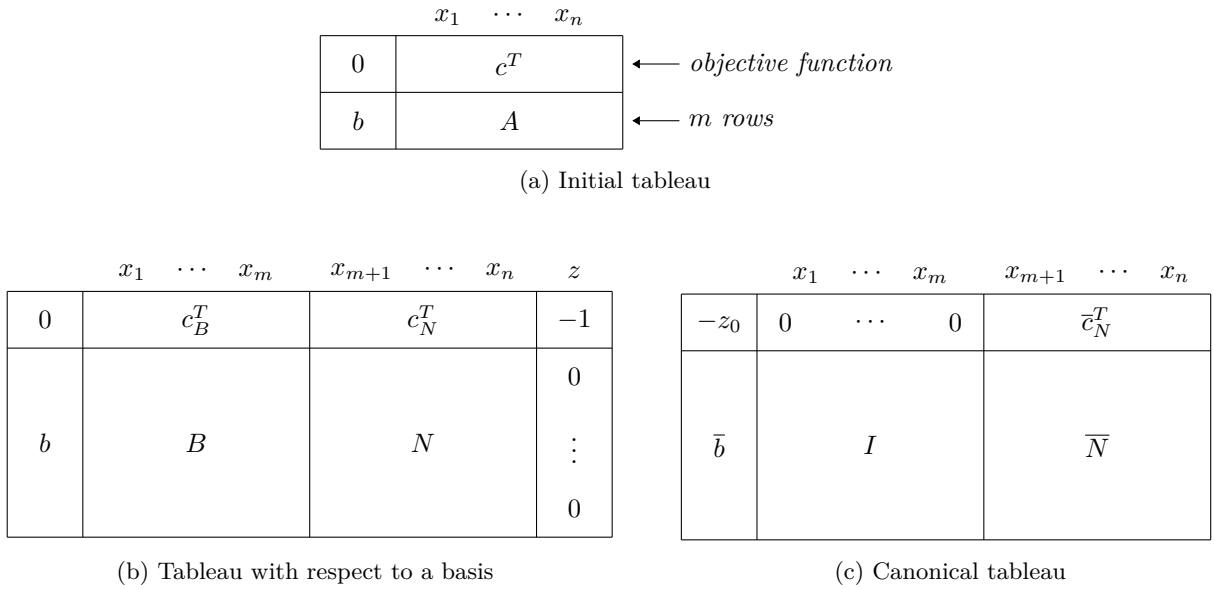
with implicit non negativity constraints be the LP problem.
The initial tableau is the matrix represented in Figure 22a. The first column contains the right hand side of the objective function and the right hand side vector.
Consider a basis B and a partition $A = [B\ N]$, with $0 = c^T x - z$. The corresponding tableau is the matrix represented in Figure 22b.
By pivoting operations *(or pre multiplication by B^{-1})*, the tableau is in the canonical form with respect to B. The said tableau is the matrix represented in Figure 22c.

3.5.7 The simplex algorithm

In Code 10 the simplex algorithm is implemented.

```
B[1], ..., B[m] := initial basis
construct the initial tableau A in canonical form with respect to B
unbounded := false
optimal := false
while optimal = false and unbounded = false do
  if a[0, j] ∀ j = 1, ..., m then
    optimal := true // for LP with min
  else
    select a non basic variable x_s with a[0, s] < 0 // negative reduced cost
    if a[i, s] ≤ 0 ∀ i = 1, ..., m then
      unbounded := true
    else
      r := argmin {a[i, 0] / a[i, s] ∀ i = 1, ..., m with a[i, s] > 0}
      pivot(r, s) // update the tableau
      B[r] := s
    end
  end
end
```

Code 10: The simplex algorithm



(a) Initial tableau

(b) Tableau with respect to a basis

(c) Canonical tableau

Figure 22: Tableau representation

3.5.8 Degenerate basic feasible solutions and convergence

As already announced in Section 3.4.6, a basic feasible solution x is degenerate if it contains at least one basic variable $x_j = 0$. A solution x with more than $n - m$ zeroes corresponds to several distinct bases.
More than n constraints *(the m of $Ax = b$ and more than $n - m$ among the n of $x \geq 0$)* on the same vertex are satisfied with equality. In the presence of degenerate basic feasible solutions *(BFS)*, a basis change may not decrease the objective function value: if the current *BFS* is degenerate, the only admissible value of θ^* is 0 and the new *BFS* is the same as the old one.
Note that a degenerate *BFS* can arise from a non degenerate one: even if $\theta^* > 0$, several basic variables may go to 0 when x_s is increased to θ^*. It's possible to cycle through a sequence of degenerate basis associated to the same vertex.

Definition 3.5. Consider the standard form polyhedron $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$, and let x be a basic solution. Let m be the number of rows of A. The vector x is a degenerate basic solution if more than $n - m$ of the components of x are zero.

A geometrical example of degenerate solution is the one in Figure 23.

3.5.8.1 Anti cycling rule

Several anti cycling rules have bern proposed for the choice of the variables that enter end exit the bases *(indices r and s in Algorithm 10)*.
The simplest one is the **Bland's rule**: among all candidate variables for x_s and x_r, the one with the smallest index is chosen.

Property 3.5. The simplex algorithm with Bland's rule terminates after less than $\binom{n}{m}$ iterations.

3.5.9 Two phase simplex algorithm

The two phase simplex algorithm is a modification of the simplex algorithm that allows to solve the *LP* problem with equality constraints.

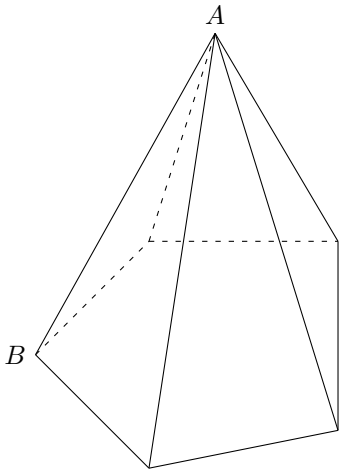


Figure 23: A is a degenerate BFS. B is a non degenerate BFS.

Phase 1: Determine an initial basic feasible solution

- given the problem with equality constraints
$$\begin{aligned} & z = c^T x \\ & Ax = b\,, b \geq 0 \\ & x \geq 0 \end{aligned}$$
- an auxiliary LP with artificial variables is constructed
$$\begin{aligned} \min \quad & v = \sum_{i=1}^{m} y_1 \\ \text{s.t.} \quad & Ax + ly = b \\ & x \geq 0, y \geq 0 \end{aligned}$$

Phase 2: Solve the auxiliary problem

- if $v^* > 0$, the problem is infeasible
- if $v^* = 0$, $y^* = 0$ and x^* is a basic feasible solution of the original problem
  - if y_i is non basic $\forall\, i$, with $1 \leq i \leq m$, the corresponding columns are deleted and a tableau in canonical form is obtained; the row of z must be determined via substitution
  - if there is a basic y_i *(the basic feasible solution is degenerate)*, then a pivot operations is performed on the row of y_i to exchange the basic variable with a non basic one

3.6 Linear Programming duality

To any *maximization LP* problem, a corresponding *(with the same parameters) minimization* problem can be associated; in the same way, a *minimization* problem can be associated to a *maximization* problem.
The two problems are called **dual** of each other. Despite having different spaces and objective functions, but in general the optimal objective function values coincide.

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \qquad (3.7)$$

$$\begin{aligned} \min \quad & b^T y \\ \text{s.t.} \quad & A^T y \geq c \\ & y \geq 0 \end{aligned} \qquad (3.8)$$

Definition 3.6. The problem in 3.6 is called the **primal problem** and the problem in 3.6 is called the **dual problem**.

Definition 3.7. The dual of the dual problem coincides with the primal problem.

3.6.1 General transformation rules

Table 2 summarizes the general transformation rules for the primal and dual problems.

PRIMAL	*minimize*	*maximize*	**DUAL**
constraints	$\geq b_i$ $\leq b_i$ $= b_i$	≥ 0 ≤ 0 free	**variables**
variables	≥ 0 ≤ 0 free	$\leq c_j$ $\geq c_j$ $= c_j$	**constraints**

Table 2: Transformation rules for the dual problem

3.6.2 The duality theorem

3.6.2.1 Weak duality

Theorem 3.9 (Weak duality)**.** *If x is a feasible solution to the primal problem (Equation 3.6) and y is a feasible solution to the dual problem (Equation 3.6), then*

$$p'b \leq c'x$$

Proof. For every pair $x \in X$ and $y \in Y$, where

$$X = \{x \mid Ax \geq b, x \geq 0\} \neq \emptyset \quad Y = \left\{y \mid A^T y \leq c, y \geq 0\right\} \neq \emptyset$$

the following relations hold:

$$Ax \geq b, x \geq 0, A^T y \leq c, y \geq 0 \Rightarrow b^T y \leq x^T A^T y \leq x^T c = c^T x$$

□

Consequences of the duality theorem If $x \in X$ is a feasible solution of the primal problem *(3.6)*, then $y \in Y$ is a feasible solution of the dual problem *(3.6)*, and the values of the respective objective functions coincide, $c^T x = b^T y$, then x is optimal for the primal problem and y is optimal for the dual problem. Optimal solutions are denoted by x^* and y^*.
The following corollaries are immediate consequences of the duality theorem.

Corollary 3.10. *If the optimal cost in the primal problem is $-\infty$, then the dual problem must be infeasible*

Corollary 3.11. *If the optimal cost in the primal problem is $+\infty$, then the dual problem must be infeasible*

3.6.2.2 Strong duality

Theorem 3.12 (Strong duality)**.** *If $X = \{x \mid Ax \geq b, x \geq 0\} \neq \emptyset$ and $\min\left\{c^T \mid x \in X\right\}$ is finite, there exists $x^* \in X$, $y^* \in Y$ such that $c^T x^* = b^T y^*$.*

Proof. Derive an optimal solution of the dual problem *(3.6)* from one of the primal problem *(3.6)* and

$$x' = \begin{bmatrix} x_B^* \\ x_N^* \end{bmatrix} \quad x_B^* = B^{-1}b,\ x_N^* = 0$$

as optimal feasible solution of the primal problem, provided by the Simplex algorithm with the Bland's rule. Consider $\overline{y}^T = c_B^T B^{-1}$.

- Verify that $\overline{y}$ is a feasible solution of the dual problem:
  1. for the non basic variables

$$\overline{c}_N^T = c_N^T - \left(c_B^T B^{-1}\right) N = c_N^T - \overline{y}^T N = \overbrace{\geq 0^T}^{x^* \text{ optimal}} \Rightarrow \overline{y}^T N \leq c_N^T$$

  2. for the basic variables

$$\overline{c}_B^T = c_B^T - \left(c_B^T B^{-}1\right) B = c_B^T - \overline{y}^T B = 0^T \Rightarrow y^T B \leq c_B^T$$

- According to the duality theorem,

$$\overline{y}^T b \leq c^T x^* \Rightarrow \overline{y}^T b = c^T x^*$$

Hence $\overline{y}$ is an optimal solution of the dual problem: $\overline{y} = y^*$. □

Consequences of the strong duality theorem Since in a linear programming problem, only one of the three following cases can occur:

- There exists an optimal solution
- The problem is unbounded
  - the optimal cost is $+\infty$ for maximization problems
  - the optimal cost is $-\infty$ for minimization problems
- The problem is infeasible

This leads to nine possible combinations of solutions of the primal and the dual problems; all of them are shown in Table 3.

		dual		
		finite optimum	*unbounded*	*infeasible*
primal	*finite optimum*	possible	impossible	impossible
	unbounded	impossible	impossible	possible
	infeasible	impossible	possible	possible

Table 3: Possible combinations of solutions of the primal and the dual

3.7 Complementary slackness conditions

An important relation between primal and dual optimal solutions is provided by the complementary slackness conditions, shown in Theorem 3.13.

Theorem 3.13 (Complementary slackness)**.** *Let $x \in X^*$ and $y \in Y^*$ be feasible solutions to the primal and dual problem, respectively. Then x and p are optimal for their respective problems if and only if*

$$y_i^* \left(a_i' x - b_i\right) = 0 \quad \forall\, i \in \{i = 1, \ldots, m\}$$
$$\left(c_j - p' A_j\right) x_j^* = 0 \quad \forall\, j \in \{j = 1, \ldots, n\}$$

where a_i denotes the i-th row of A and A_j the j-th column of A.

At optimality, the product of each variable with the corresponding slack variable of the constraint of the relative dual problem must be zero.