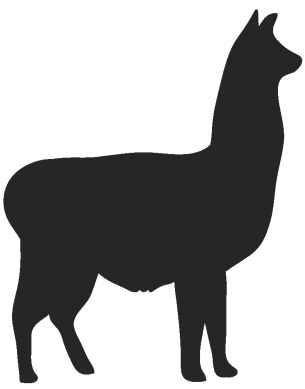
Foundation of Operations Research

Lorenzo Rossi and everyone who kindly helped!

2022/2023

Last update: 2022-10-04



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.

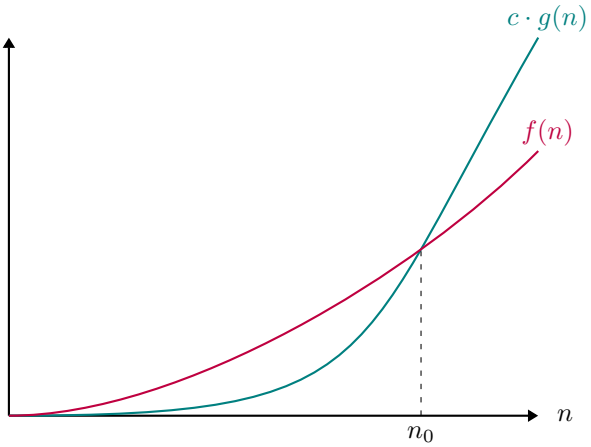


Figure 1: Big-O notation

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
  - an edge connecting nodes i and j is represented by $\{i, j\}$ if the graph is **undirected**
  - an edge connecting nodes i and j is represented by (i, j) if the graph is **directed**

Properties:

- Two **nodes** are **adjacent** if they are connected by an edge
- An **edge** e is **incident** in a node v if v is an **endpoint** of e
  - **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

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

  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

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

  connecting nodes, with $(v_i, v_{i+1} \in E)$ for $i = 1, \ldots, k-1$
- Nodes u and v are **connected** if exists a path connecting them
- A graph (N, E) is **connecting** 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 $E = \{\{v_i, v_j\} \,|\, v_i, v_j \in N \wedge i \leq j\}$
- Given a directed graph $G = (N, A)$ and $S \subseteq N$, the **outgoing cut** induced by S is the set of arcs:

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

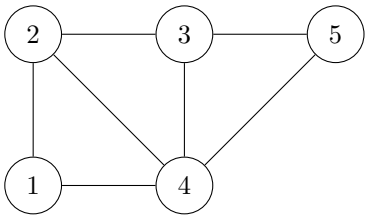
  the **incoming cut** induced by S is the set of arcs:

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

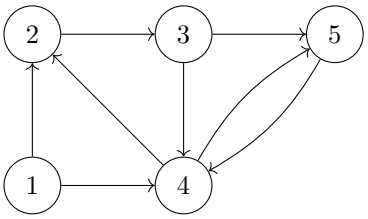
Some examples are shown in Figure 2.

Properties of graphs:

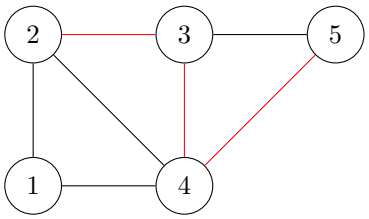
- A graph with n **nodes** has at most $m = \dfrac{n(n-1)}{2}$ **edges**
- A **directed** graph with n **nodes** has at most $m = n(n-1)$ **arcs**
  - a graph is **dense** if $m \approx n^2$
  - a graph is **sparse** if $m \ll n$



(a) Undirected graph

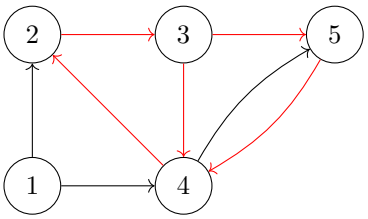


(b) Directed graph



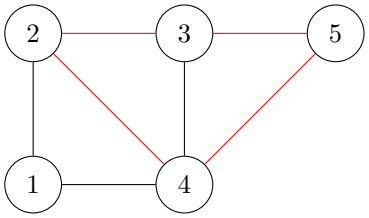
Connected graph, nodes 2 and 5 are connected

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



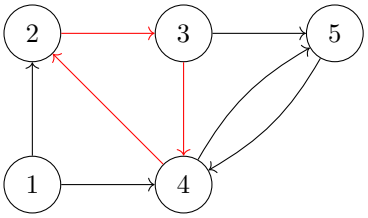
Not strongly connected graph

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



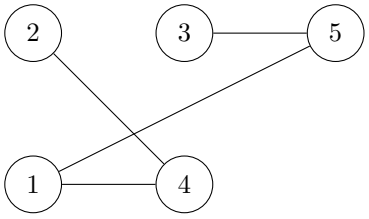
Cycle

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



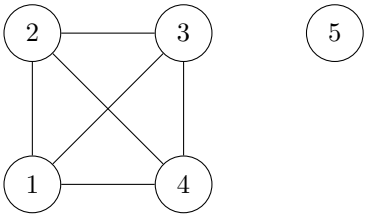
Circuit

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



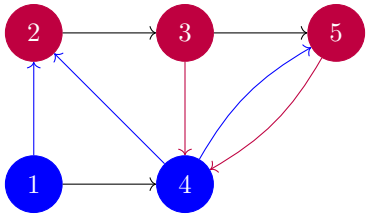
Bipartite graph

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



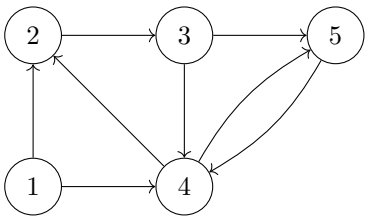
Complete graph

(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

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

Definition: given a directed graph $G = (N, A)$ and a node $s \in N$, find 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:

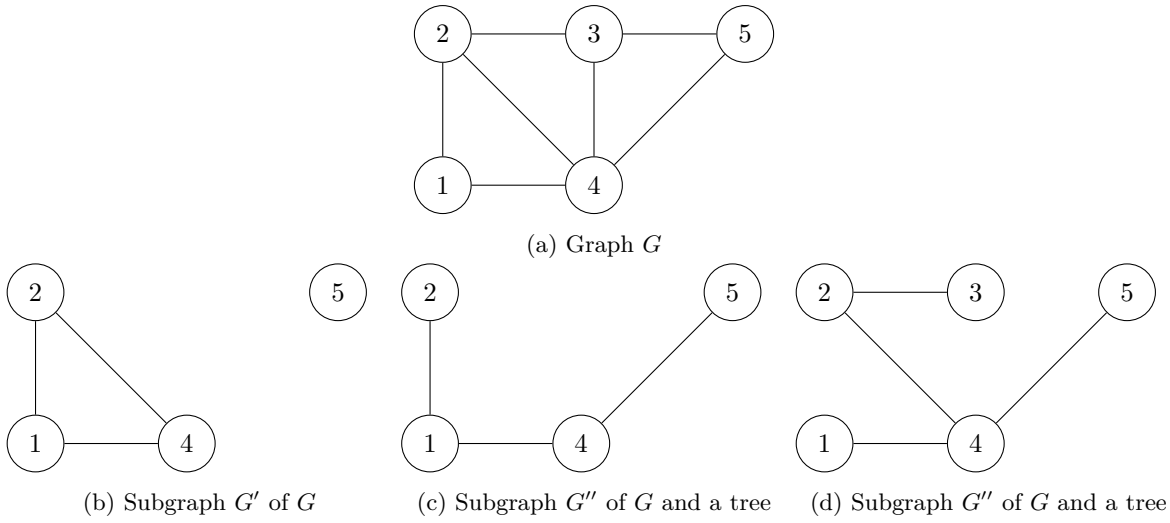
- a **queue** Q of nodes not yet processed is kept by the algorithm
- the queue uses a `FIFO` policy
- the nodes exploration is performed in a **breadth-first** manner

Algorithm The algorithm pseudocode is shown in Code 1.

```
1  Q := {s}
2  M := {}
3  while Q is not empty do
4    u := node ∈ Q
5    Q := Q \ {u}
6    M := M ∪ {u}
7    for (u, v) ∈ δ+(u) do
8      if v ∉ M and v ∉ Q then
9        Q := Q ∪ {v}
10     end
11   end
12 end
```

Code 1: Graph reachability

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) 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

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

2.3.1 Property 1 - number of edges

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

2.3.1.1 Proof

- **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

2.3.2 Property 2 - number of paths

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

2.3.3 Property 3 - new cycles

By adding a new edge to a tree, a new unique cycle is created. This cycle consists of the path in Property 2 - number of paths and the new edge.

2.3.4 Property 4 - exchange property

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

2.3.5 Property 5 - cut property

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.

2.3.5.1 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 $c(T^* \cup \{e\} \setminus \{f\}) < c(T^*)$, hence T^* is not optimal

2.4 Optimal cost spanning tree

Spanning trees have a number of applications:

- **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** ⇒ the output must be a **tree**

Formalized problem: 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:

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

2.4.1 Theorem 1 - number of nodes in spanning trees

The **Theorem 1**, formulated by *Cayley* in 1889, states that:
A complete graph with n nodes ($n \geq 1$) has n^{n-2} spanning trees.

2.4.2 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 ste, 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

→ *Input*: connected graph $G = (N, E)$ with edge costs.
→ *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.

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

Code 2: Prim's algorithm

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.

2.4.2.1 Correcteness of Prim's algorithm

Proposition: Prim's algorithm is exact.
The exactness does not depend on the choice of the first node nor on the selected edge of minimum cost $\delta(S)$. Each selected edge is part of the optimal solution as it belongs to a minimum spanning tree.

2.4.2.2 Optimality test

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.

2.4.2.3 Implementation in quadratic time

The Prim's algorithm can be implemented in quadratic time $(\mathcal{O}(n^2))$.

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.

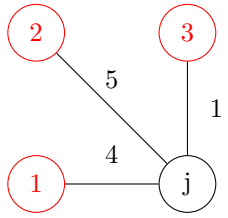


Figure 6: Data structure
nodes $1, 2, 3 \in S$, node $j \notin S$
$\text{closest}_j = 3 \quad c_{\text{closest}_j j} = 1$

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.

```
1  T := {}
2  S := {u}
3  // initialization
4  for j ∉ N \ S do
5    if {u, j} ∈ E then
6      C_j := c_{u, j}
7    else
8      C_j := +\infty
9    end
10   closest_j := u
11 end
12 for k := 1 to n - 1 do
13   min := +∞ // selection of min cost edge
14   for j := 1, ..., n do
15     if j ∉ S and C_j < min then
16       min := C_j
17       v := j
18     end
19   end
20   S := S ∪ {v} // extend S
21   T := T ∪ {{v, closest_v}} // extend T
22   for j := 1 to n do
```

```
23      if j ∉ S and c_vj < C_j then
24        C_j := c_vj
25        closest_j := v
26      end
27    end
28  end
```

Code 3: Prim's algorithm in quadratic time

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 ($\mathcal{O}\left(m \log\left(2\right)\right)$) *(using priority queues)* is possible.

2.4.3 Optimality condition

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 $c_e < c_f$.

2.4.4 Theorem 2 - Tree optimality condition

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

2.4.4.1 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:

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

Model: 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)*

Properties of optimal paths:

- 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

2.5.1 Property 6 - shortest path

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

2.5.2 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
- $L_j = \begin{cases} \text{cost of a shortest path from } s \text{ to } j & j \in S \\ \min\{L_i + c_{ij} \,|\, (i,j) \in \delta^+(S) & j \notin S\} \end{cases}$
  $\rightarrow$ 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: $L_u + c_{uv} = \min\{L_i + c_{ij} \,|\, (i,j) \in \delta^+(S)\}$
  $\rightarrow$ thus: $L_u + c_{uv} \leq L_i + c_{ij}, \forall \, (i,j) \in \delta^+(S)$
- $\text{predecessor}_j = \begin{cases} \text{predecessor of } j \text{ in the shortest path from } s \text{ to } j & j \in S \\ u \text{ such that } L_u + c_{uj} = \min\{L_i + c_{ij} \,|\, i \in S\} & j \notin S \end{cases}$

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 4.

```
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 4: Dijkstra's algorithm

2.5.2.1 Correcteness of Dijkstra's algorithm

Dijkstra's algorithm is correct.
Proof:

1. A the k-th step:
   - $S = \{s, i_1, \ldots, i_{k-1}\}$
   - $\begin{cases} \text{cost of a minimum cost path from } s \text{ to } j & j \in S \\ \text{cost of a minimum cost path with all intermediate nodes in } S & j \notin S \end{cases}$
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

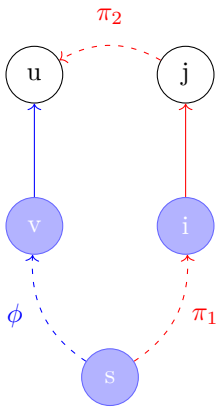
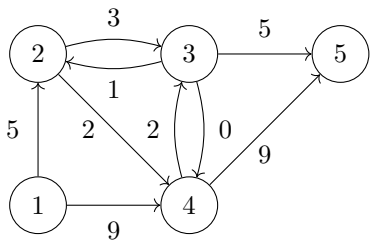
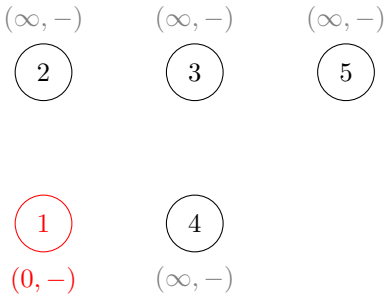


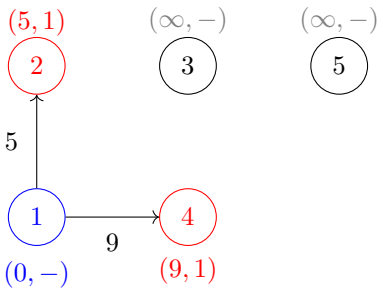
Figure 7: Proof of the induction step; nodes s, v, i are in cut S



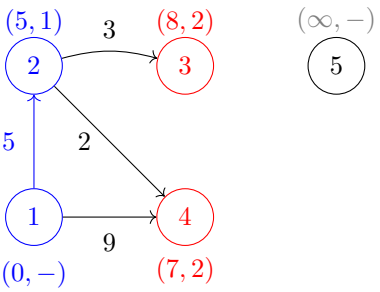
(a) Sample graph, with the cost of each arc



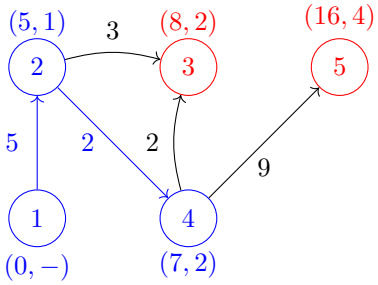
(b) step 1 of Dijkstra's algorithm



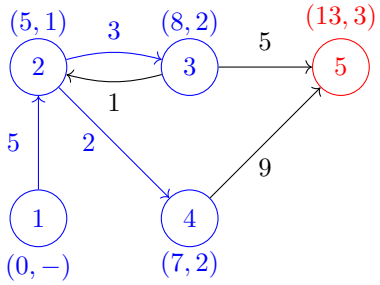
(c) step 2 of Dijkstra's algorithm



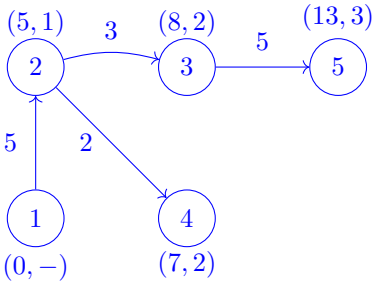
(d) step 3 of Dijkstra's algorithm



(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

2.5.2.2 Example of Dijkstra's algorithm

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

2.5.3 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 5.

```
1  for i := 1 to n do
2    for j := 1 to n do
3      p_id := i
4
5      if i = j then
6        d_ij := 0
7      else if (i, j) in A then
8        d_ij := c_ij
9      else
10       d_ij := +∞
11     end
12   end
13 end
14 for u ∈ N do
15   for i ∈ N \{ u } do
16     for j ∈ N \{ u }
17       if d_iu + d_uj < d_ij then
18         p_ij := p_uj
19         d_ij := d_iu + d_uj
20       end
21     end
```

```
22    for i ∈ N do
23      if d_ij < 0 then
24        error "negative cycle"
25      end
26  end
```

Code 5: Floyd-Warshall's algorithm

2.5.3.1 Correctness of Floyd-Warshall's algorithm

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 ($\{1, \ldots, u\}$)

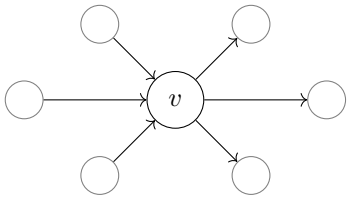
2.6 Optimal paths in directed, acyclic graphs

A directed graph $G = (N, A)$ is **acyclic** if it does not contain any circuit. A directed acyclic graph G is then referred to as a ***DAG***.
Property of *DAGs*: the nodes of any directed acyclic graph 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: 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.

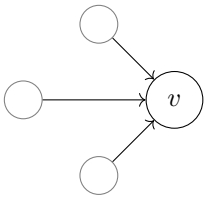
2.6.1 Topological ordering method

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 $\delta^-(v) = \emptyset$
   $\rightarrow$ 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.

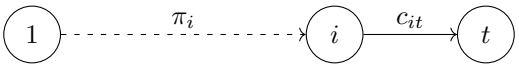


Figure 10: Shortest path from 1 to t

The strategy to find the shortest path is:

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 Algorithm 6.

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

Code 6: Shortest path in *DAG*

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

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.
Model: 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.

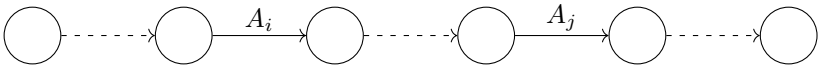


Figure 11: Precedence relation in project planning

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 $(v, j) \in \delta^+(v)$.

2.7.1 Property 7

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.

2.7.2 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: 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).

2.7.2.1 Property 8

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.

2.7.2.2 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)*

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 $\sigma ij = T_{max_j} - T_{min_i} - d_{ij}$

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

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

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

Code 7: Critical path method

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 - $\mathcal{O}(n + m)$

2.7.2.3 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.

2.7.2.4 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 $T_{max_i} + d_{ij}$