Math 2014 Algorithms

Dr. Brendan Ames

Table of contents

Preface

This document collects lecture notes for **Math 2014 Algorithms**.

Please follow the page navigation for each topic covered in the module.

1 Introduction to Algorithms

1.1 Problems and Problem Instances

A **problem** P is a general class of questions to answer.

Problems are **(infinite) families of general questions**.

We call a version of the problem with specific values a **problem instance** I (or just **instance**).

Example 1.1 (Quadratic Equations)**.** Finding the roots of a **quadratic equation** $ax^2+bx+c=0$ is an example of a **problem**.

Finding the roots of a ***specific*** quadratic function is an **instance** of this problem. For example, finding x such that $2x^2+8x+5=0$ is a problem instance.

1.2 Algorithms

We are interested in finding **procedures** for solving **every possible instance** of a **given problem**. Such a procedure is called an **algorithm**.

Definition 1.1 (Algorithm)**.** An **algorithm** for a given problem is a **finite sequence** of operations which return the **correct solution for all problem instances**.

Example 1.2 (An Algorithm for Quadratic Equations)**.** We solve quadratic equations of the form $ax^2+bx+c=0$ using the **quadratic formula**:

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}.$$

We can think of evaluating this formula as an algorithm consisting of the steps:

1. Compute $u:=b^2-4ac$.
2. Compute $v:=2a$.
3. Compute $z:=\sqrt{u}=\sqrt{b^2-4ac}$.
4. Compute
$$x_1=\frac{-b+z}{v}, \qquad x_2=\frac{-b-z}{v}.$$

1.3 Goals of the Module

We are interested in:

- Designing algorithms for solving specific problems.
- Proving that a proposed algorithm produces correct solutions.
- **Complexity of algorithms:** Analysing how long it takes for the algorithm to terminate, i.e., how many operations are required in general.
- **Complexity of problems:** how long it may take to solve a problem, regardless of which algorithm is used.

1.4 Combinatorial Optimization

1.4.1 Combinatorial Optimization Problems

We will focus on **combinatorial optimization problems**:

$$\min c(X) \text{ such that } X \in \mathcal{X},$$

where $c : \mathcal{X} \mapsto \mathbf{R}$ is a **cost function** and $\mathcal{X}$ is a **finite set** of **feasible solutions**.

1.4.2 Solving Combinatorial Problems

- Since $\mathcal{X}$ is finite, we can always solve by **complete enumeration**: exhaustively calculating $c(X)$ for each $X \in \mathcal{X}$ to find one with minimum cost.
- In practice, $\mathcal{X}$ is extremely large and complete enumeration is prohibitively expensive.
- In many cases, $\mathcal{X}$ is defined implicitly as description and enumerating all possible solutions is a difficult task on its own.
- We want **faster, specialized algorithms** exploiting mathematical structure of the given problem.

1.4.3 Examples of Combinatorial Problems

Combinatorial optimization is ubiquitous in operations research and management science[1]

Example 1.3 (Travelling Salesperson Problem)**.** Find a tour of shortest length visiting each location exactly once. Figure 1.1 gives an instance of the TSP and a solution where a tour of 15 cities in Germany is sought.

Figure 1.1: An optimal traveling salesperson tour through Germany's 15 largest cities (among over 43 billion possible routes).

Example 1.4 (The Shortest Path Problem)**.** Find shortest/minimum length route in network from origin to destination.

[1] https://en.wikipedia.org/wiki/Combinatorial_optimization

Example 1.5 (Knapsack/Assignment Problems)**.** Find maximum value/minimum cost distribution of limited resources.

1.4.4 Types of Problems

We will focus on three primary forms of combinatorial optimization problems.

Definition 1.2 (Decision Problems)**.** Given set $\mathcal{X}$, cost $c : \mathcal{X} \mapsto \mathbf{R}$ and scalar $L \in \mathbf{R}$:

$$\text{Determine if there is a } X \in \mathcal{X} \text{ with } c(X) \leq L.$$

A **decision problem** takes problem instance defined by $\mathcal{X}$, c, and L as input. An algorithm for this problem would output **Yes** or **No** depending on the instance.

Definition 1.3 (Decision problem – Search version)**.** Given $\mathcal{X}$, c, and L as input. Find $X \in \mathcal{X}$ with $c(X) \leq L$.

Definition 1.4 (Optimization problem – Search version)**.** Given $\mathcal{X}$ and c as input. Find $X \in \mathcal{X}$ ***minimizing*** $c(X)$.

1.4.5 Exact and Approximation Algorithms

There are two primary classes of algorithms that we will consider: **exact** and **approximation algorithms**.

Definition 1.5 (Exact Algorithms)**. Exact algorithms** provide an **exact solution** to the problem.

Definition 1.6 (Approximation Algorithms)**. Approximation Algorithms** provide an **approximate solution** to the problem, but not the exact solution in general.

- An α**-approximate algorithm** returns $\tilde{X}$ within α-ratio of optimal value: $c(X^*) \leq c(\tilde{X}) \leq \alpha \cdot c(X^*)$, for some $\alpha > 1$.

Aside from these two classes, we also have **heuristics** which provide potential solutions without guarantees of quality.

Definition 1.7 (Heuristics)**. Heuristic algorithms** or **heuristics** provide solutions without guarantee of closeness to the optimal solution $c(X^*)$.

1.4.6 Deterministic vs Random Algorithms

We can also classify algorithms based on whether steps are performed deterministically or at random.

Definition 1.8 (Deterministic Algorithms)**. Deterministic Algorithms** follow the same series of steps for given input:

$$\text{if A = B do X; else do Y}$$

Definition 1.9 (Randomized Algorithms)**. Randomized algorithms** have steps depending on random operations:

$$\text{Toss coin: if heads, do X; else do Y}$$

A randomized exact/α-algorithm may only return exact or α solution within a certain probability.

1.4.7 Offline vs Online vs Robust Algorithms

Finally, we can further classify algorithms based on how they process information into problem instances.

Definition 1.10 (Offline Algorithm)**.** The problem instance I is **completely known at the beginning of the algorithm.**

Definition 1.11 (Online Algorithm)**.** The problem instance I is not completely known at start of the algorithm. Instead, the algorithm adapts to I as I unfolds over time.

Definition 1.12. Robust Algorithm: Instance I is not completely known at the beginning of the algorithm and it is not revealed over time.

The algorithm finds a solution that works for **all possible realizations** that I can take.

1.4.8 Example – Canadian Traveller Problem (CTP)

Shortest Path Problem with added complexity of not knowing which roads/routes are unavailable due to the weather beforehand[2].

[2] https://en.wikipedia.org/wiki/Canadian_traveller_problem

1.4.8.1 Deterministic/Off-line Version

We want to find the shortest or minimum length path in a network from origin to destination.

If the network routes are deterministic and are known then we have a **deterministic exact algorithm** for the shortest path problem. We'll discuss this algorithm at length later in the term.

1.4.8.2 The Random Case

Let's suppose that the routes are random or partially observed. For example, this could occur when inclement weather causes certain roadways to close, but they are not known until the road closure is encountered en route.

- An **online algorithm** would find a partial path using known routes (so far), and then dynamically update the solution as conditions change or become known.
- A **robust algorithm** would find a path/itinerary that works under any weather conditions.

2 Introduction to Graphs

2.1 Directed Graphs

2.1.1 Introduction

Definition 2.1 (Directed Graphs)**.** A **directed graph** is an ordered pair $G := (V, A)$ composed of:

- a set V of **vertices** or **nodes** of size/cardinality $n := |V|$;
- a set A of **ordered pairs** called **arcs** of cardinality $m := |A|$.

For an arc $(i, j) \in A$, we call i the **tail** and j the **head**.

We will focus on graphs **without self-loops**: (i, i) is not an arc!

Example 2.1. Consider $G = (V, A)$ with

$$V = \{1, 2, 3, 4, 5\},$$

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

We can **represent** or **visualize** G as **nodes** V joined by **lines/arrows** corresponding to arcs A. Figure 2.1 gives several different representations of this graph.

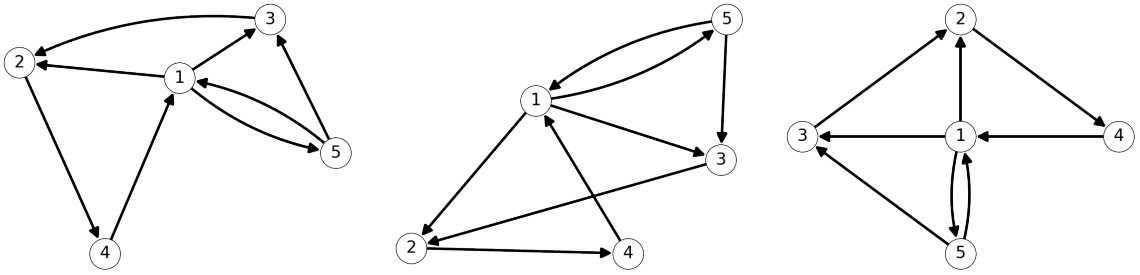
2.1.2 Adjacency

Arcs in a graph define pairwise relationships between nodes.

Definition 2.2 (Adjacent nodes)**.** Node $i \in V$ is **adjacent** to node $j \in V$ if arc (i, j) belongs to A.

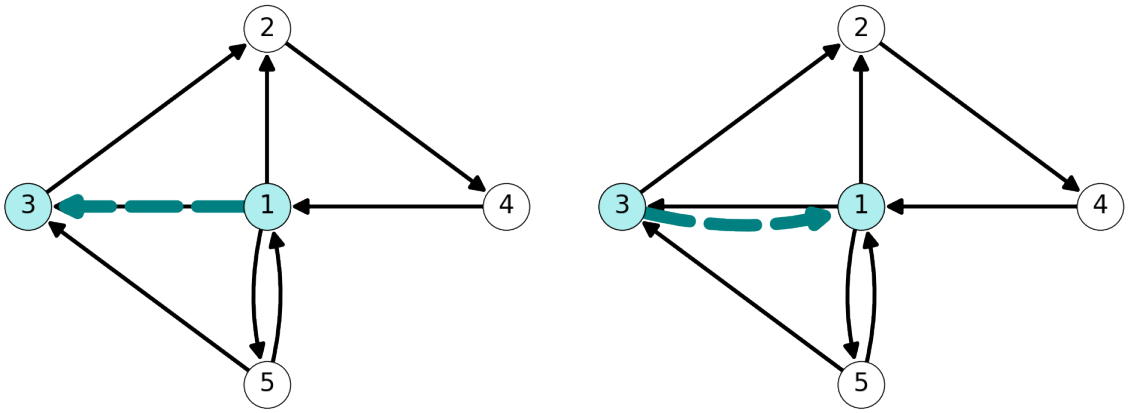
In this case, the arc $(i, j) \in A$ is **incident** to node j.

Example 2.2. Consider the graph $G = (V, A)$ given in Example 2.1. Node 1 is adjacent to 3 because the arc $(1, 3) \in A$ is included in the graph. However, Node 3 is *adjacent to* 1 since $(3, 1) \notin A$. See Figure 2.2 for illustrations of these relationships.



(a) A Visualization of the Graph (b) Another Visualization (c) A Third Visualization

Figure 2.1: Three representations of the graph given in Example 2.1.



(a) 1 is adjacent to 3 (b) 3 is not adjacent to 1

Figure 2.2: Adjacency relations of nodes 1 and 3 in G given in Example 2.1.

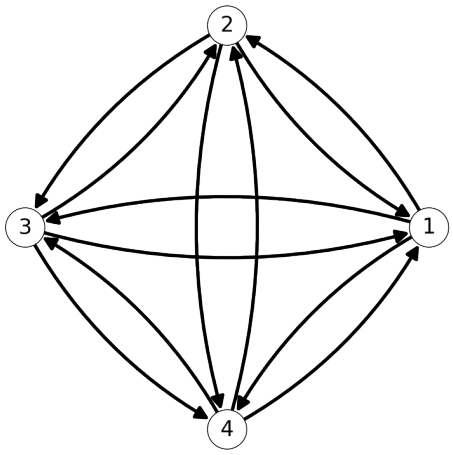
2.1.3 Complete Graphs

Definition 2.3 (Complete Graphs)**.** A directed graph $G = (V, A)$ is **complete** if:

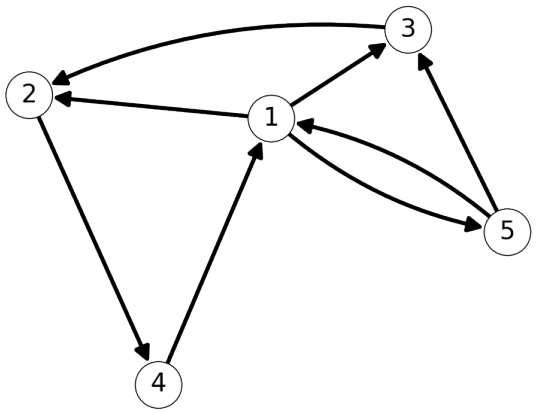
- A contains an arc for each pair of nodes in V;
- Equivalently, every node in V is adjacent to every other node.

Example 2.3. The graph given in Example 2.1 is ***not complete*** or ***incomplete***. Indeed, there are several potential arcs, e.g., $(4, 5)$, which are not present in this graph.

However, Figure 2.3a provides a visualization of the complete graph on 4 nodes. Each of the possible arcs between pairs of nodes is present.



(a) A Complete Graph with 4 nodes



(b) An Incomplete Graph

Figure 2.3: The complete graph with $n = 4$ nodes and the incomplete graph from Example 2.1.

2.2 Number of Arcs in a Complete Graph

Theorem 2.1. *For any directed graph G, we have $m \leq n(n-1)$. If G is **complete** then $m = n(n-1)$.*

Proof. Each of the n nodes *could* be adjacent to each of the other $n-1$ nodes. Therefore the number of edges is bounded above by

$$m \leq n(n-1).$$

When the graph is complete, every possible edge is present, i.e., every node is adjacent to every other node. Therefore,

$$m = n(n-1)$$

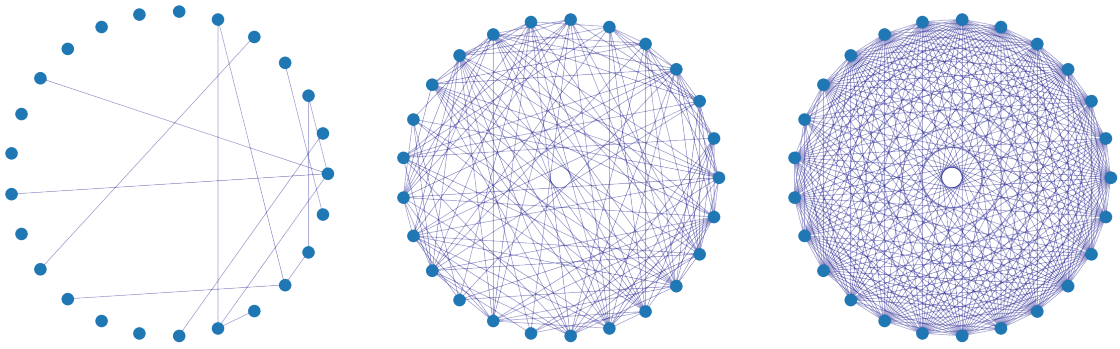
if G is complete. On the other hand, $m < n(n-1)$ if G is not complete. □

2.3 Dense and Sparse Graphs

Definition 2.4 (Dense/Sparse Graphs)**.** A directed graph G is **sparse** if $m << n(n-1)$. Otherwise G is **dense**.

Example 2.4. Figure 2.4 provides visualisations of three graphs with increasing density.

- A sparse graph with 5% of possible edges present (Figure 2.4a).
- A dense graph with 50% of possible edges present (Figure 2.4b).
- A very dense graph with 95% of possible edges present (Figure 2.4c).



(a) Sparse graph (b) Dense graph (c) Very dense graph

Figure 2.4: Three graphs with $n = 25$ with increasing density.

2.4 Paths and Connectivity

We can extend our definition of adjacency to describe pairwise relationships of nodes via sequences of arcs.

Definition 2.5 (Path)**.** A **path** is a sequence of arcs

$$(i_1, i_2), (i_2, i_3), \dots, (i_k, i_{k+1})$$

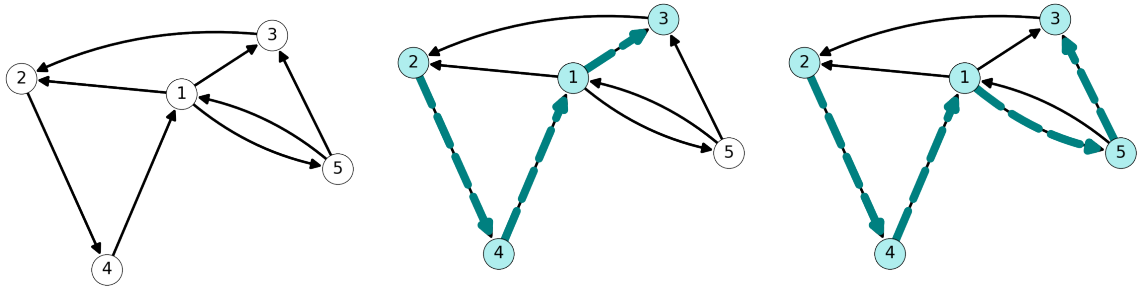
with $k+1 \geq 2$ nodes with **distinct origin** i_1 and **destination** i_{k+1}.

We can also use the notation

$$(i_1, i_2, \dots, i_k, i_{k+1})$$

to denote a (i_1, i_{k+1})-path.

Example 2.5. The graph visualised in Figure 2.5 contains several paths from 2 to 3. For example, both $P_1 = ((2,4),(4,1),(1,3))$ and $P_2 = ((2,4),(4,1),(1,5),(5,3))$ are 23-paths.



(a) Graph G (b) Path P_1 $((2,4),(4,1),(1,3))$ =(c) Path P_2 $((2,4),(4,1),(1,5),(5,3))$ =

Figure 2.5: Two 23-paths, P_1, P_2, in graph G.

Definition 2.6 (Connectivity)**.** Node v is **connected** to node w if there is path in G with origin $i_1 = v$ and destination $i_{k+1} = w$.

Definition 2.7. A graph is **connected** if every pair of nodes is connected.

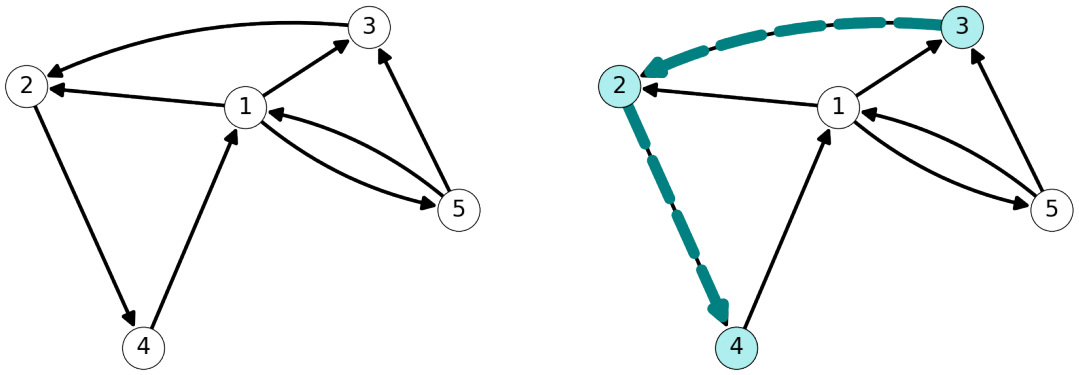
Example 2.6. Node 2 is connected to Node 3 in the graph given in Example 2.5. Indeed, we saw that there are at least two 23-paths in Example 2.5.

Moreover, the graph G is connected because we every pair of nodes is connected via an arc. Indeed, we can use the subpaths of the paths P_1, P_2 and the 34-path

$$P_3 = ((3,2),(2,4))$$

to reach every node from every other node. For example, we can use parts of P_2 and P_3 to find the 14-path

$$P_{14} = ((1,5),(5,3),(3,2),(3,4)).$$



(a) Graph G (b) $P_3 = ((3,2),(2,4))$

Figure 2.6: A 34-path P_3 in G. This, along with P_1, P_2 from Example 2.5 provides paths between every pair of nodes in G; hence, G is connected.

2.4.1 Cycles

If a path begins and ends at the same node, then we call it a *cycle*.

Definition 2.8 (Cycles)**.** A **cycle** is a sequence

$$(i_1, i_2), (i_2, i_3), \ldots, (i_k, i_{k+1})$$

of $k \geq 2$ consecutive and distinct arcs with $i_{k+1} = i_1$ (i.e., the origin and destination coincide).

Note that a cycle may visit some nodes more than once (other than the origin/destination).

Example 2.7. The graph G given in Figure 2.7 contains the cycles $C_1 = (2, 5, 4, 3, 2)$ and $C_2 = (1, 3, 5, 4, 2, 1)$. Note that C_2 visits node 2 *twice*. This implies that we also have cycles $C_3 = (1, 3, 2, 1)$ and $C_4 = (2, 5, 4, 2)$.

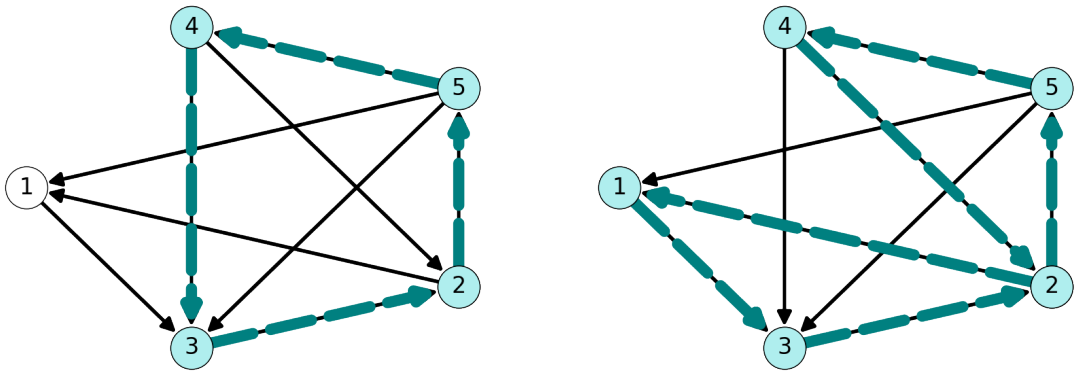
2.5 Directed Cuts

Having introduced notions of connectivity, we now define sets of arcs whose removal disconnect the graph.

Definition 2.9 (Forward Cut)**.** Let $S \subseteq V$, that is, S is a **subset** of the node set V.

The **set of arcs** with **tail** in S and **head** in $V \setminus S$ is the **forward directed cut induced by** S:

$$\delta^+(S) := \{(i,j) \in A : i \in S \text{ and } j \in V \setminus S\}.$$



(a) Cycle $C_1 = (2, 5, 4, 3, 2)$ (b) Cycle $C_2 = (1, 3, 5, 4, 2, 1)$

Figure 2.7: A graph G containing cycles C_1 and C_2.

Informally, the forward directed cut is the set of arcs that ***leave*** S.

Definition 2.10 (Backward Cut)**.** The **backward directed cut induced by** S is the set of arcs with tail in $V \setminus S$ and head in S:

$$\delta^-(S) := \delta^+(V \setminus S) = \{(i, j) \in A : i \in V \setminus S \text{ and } j \in S\}.$$

The backward directed cut is the set of arcs ***entering*** S.

Example 2.8. Consider $S = \{4, 5\}$ for the graph given in Figure 2.8a. This graph and set of nodes has forward and backward cuts

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

2.5.1 Stars and Degrees

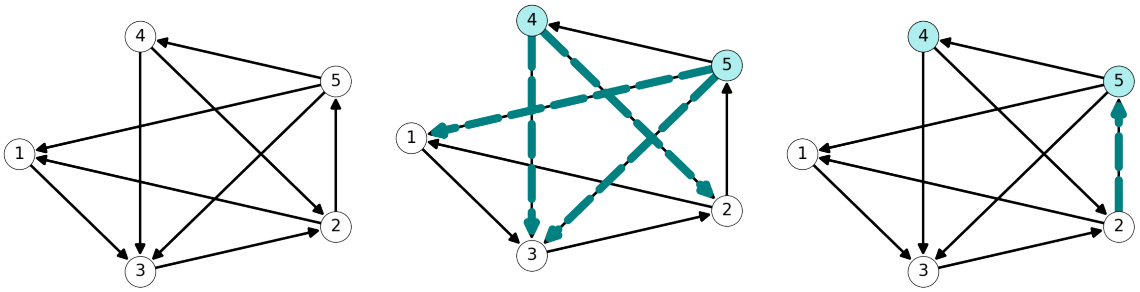
Definition 2.11 (Stars)**.** Let $i \in V$. The **forward** and **backward stars** of i are the cuts

$$\delta^+(\{i\}) \text{ and } \delta^-(\{i\})$$

respectively.

Definition 2.12 (Degree)**.** The **out-degree** and **in-degree** of i are the number of edges with i as **tail** and **head**, respectively:

$$|\delta^+(\{i\})| \text{ and } |\delta^-(\{i\})|.$$

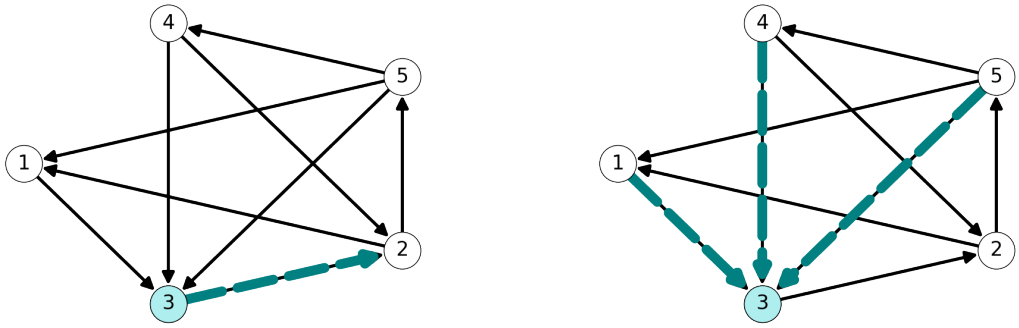


(a) Graph G (b) $\delta^+(\{4,5\})$ $\{(4,2),(4,3),(5,1)\}$ = (c) $\delta^-(\{4,5\}) = \{(2,5\}$

Figure 2.8: Forward and backward cuts of $S = \{4,5\}$ is graph G.

Example 2.9. Consider $v = 3$, i.e., $S = \{v\} = \{3\}$, in the graph G considered in Example 2.8:

- The forward star is $\delta^+(\{3\}) = \{(3,2)\}$.



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

Figure 2.10: Forward and backward star of $\{3\}$ in G. Note that $\{3\}$ has out-degree 1 and in-degree 3.

2.6 Undirected Graphs

Definition 2.13 (Undirected Graphs)**.** An **undirected graph** is an ordered pair $G := (V, E)$ composed of

- a set of **vertices** V $(n = |V|)$
- a set $E \in V \times V$ of **unordered pairs** called **edges** $(m = |E|)$.

For an edge $\{i, j\}$, we call i and j its **endpoints**.

Example 2.10. Consider $G = (V, E)$ with

$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{12, 13, 15, 23, 24, 25, 34, 35, 45\}.$$

Figure 2.11 gives a visualisation of this graph.

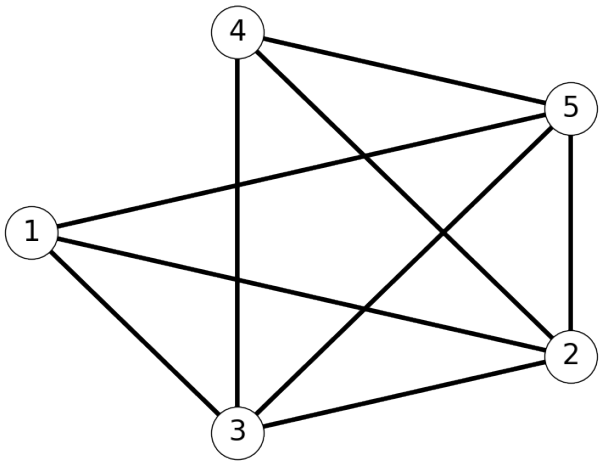


Figure 2.11: Visualisation of graph G given in Example 2.10

2.7 Properties of Undirected Graphs

2.7.1 Adjacency

Properties ofdirected graphs generalize to undirected graphs with minor differences:

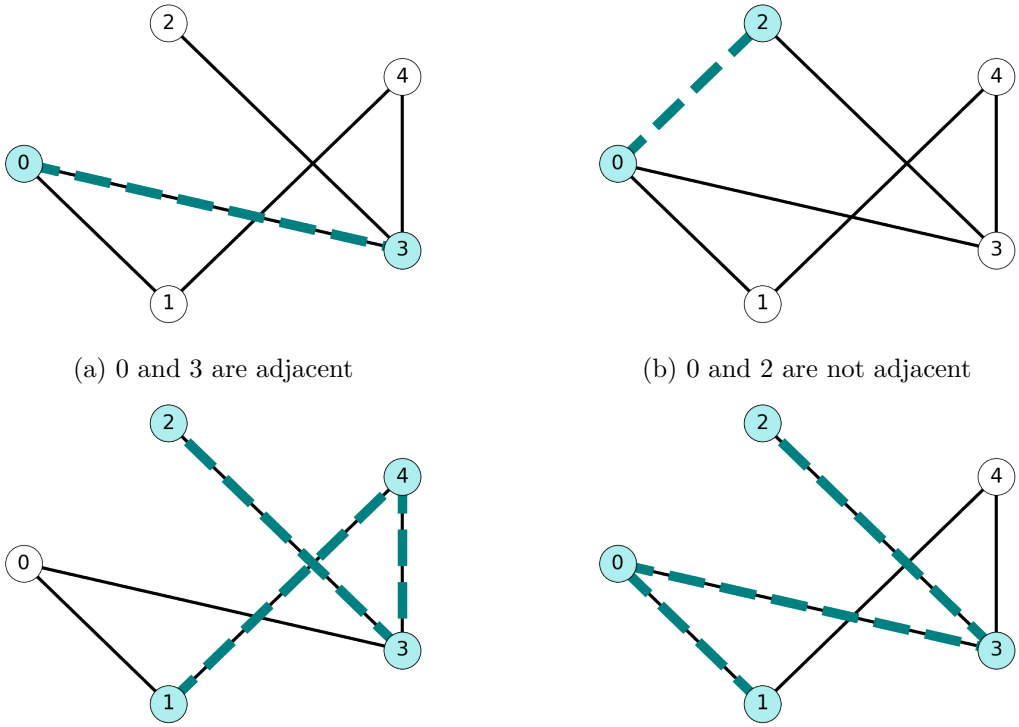
- $ij \in E$ is **incident** to **both** i and j.
- If $ij \in E$ then i and j are **(mutually) adjacent**.
- If there is a path from i to j then i and j are **(mutually) connected**.
- $G = (V, E)$ is **complete** if each pair of vertices in V is adjacent:

$$E = \{\{i, j\} : i, j \in V,\ i \neq j\}.$$

Theorem 2.2. *Every undirected graph G has $|E| \leq n(n-1)/2$ edges.*

Example 2.11. Consider the undirected graph G given in Figure 2.12:

- 0 and 3 are adjacent because $\{0, 3\} \in E$.
- 0 and 2 are *not* adjacent because $\{0, 2\} \notin E$.
- 1 and 2 are connected. Indeed, G contains 12-paths $(1, 4, 3, 2)$ and $(1, 0, 3, 2)$.



(a) 0 and 3 are adjacent

(b) 0 and 2 are not adjacent

(c) 12-path $(1, 4, 3, 2)$

(d) 12-path $(1, 0, 3, 2)$

Figure 2.12: Adjacency and connectivity properties of graph G.

2.7.2 Cuts, Stars, and Degree in Undirected Graphs

Definition 2.14 (Undirected Cuts)**.** The **undirected cut** induced by $S \subseteq V$ is the set of edges with one end point in S and one in $V \setminus S$:

$$\delta(S) := \{ij \in E : i \in S,\ j \notin S\}.$$

Definition 2.15 (Stars)**.** The **star** of node $i \in V$ is the set $\delta(\{i\})$.

Definition 2.16 (Degree)**.** The **degree** of $i \in V$ is the cardinality of its star: $|\delta(\{i\})|$.

Example 2.12. Consider the graph given in Figure 2.13:

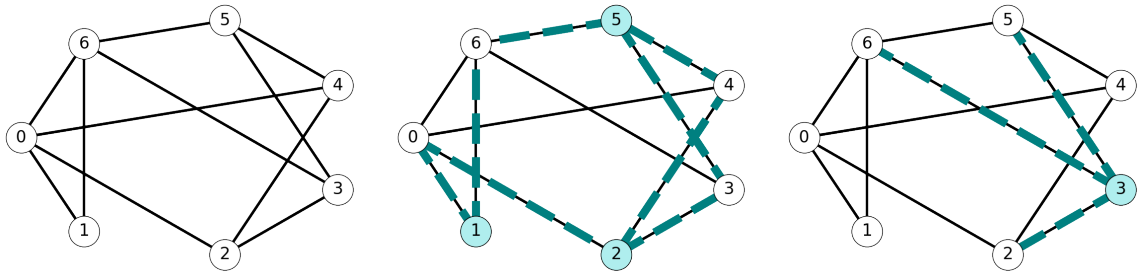
- The cut for $S = \{1, 2, 5\}$ is

$$\delta(S) = \{01, 16, 02, 23, 24, 53, 54, 56\}$$

- The star for node 3 is

$$\delta(\{3\}) = \{23, 53, 63\}.$$

- Node 3 has degree 3.



(a) Undirected graph G (b) Undirected cut $\delta(S)$ (c) Undirected star $\delta(\{3\})$

Figure 2.13: Undirected cut and star in an undirected graph.

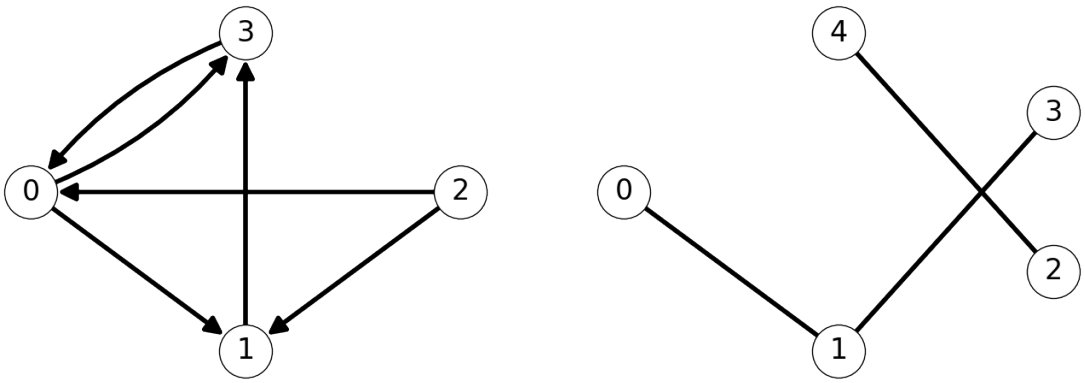
2.8 Adjacency Lists and Matrices

Definition 2.17 (Adjacency Lists)**.** An **adjacency list** L is a list of size n where each component $L(i)$ is a list of nodes adjacent to i:

- $L(i) = \{j : (i, j) \in \delta^+(\{i\})\}$ for **directed graphs**;
- $L(i) = \{j : \{i, j\} \in \delta(\{i\})\}$ for **undirected graphs**.

Definition 2.18 (Adjacency Matrices)**.** The **adjacency matrix** $A = A(G)$ of $G = (V, E)$ is a **binary matrix** $A \in \{0, 1\}^{n \times n}$ such that

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



(a) A directed graph (b) An undirected graph

Figure 2.14: Graphs for examples of adjacency lists.

Example 2.13. Consider the directed graph given in Figure 2.14a.

This graph has the adjacency lists

$$L(0) = \{1, 3\}, \qquad L(1) = \{3\},$$
$$L(2) = \{0, 1\}, \qquad L(3) = \{0\}.$$

Moreover, the adjacency matrix of this graph is

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

Example 2.14. Consider the directed graph given in Figure 2.14b.

This graph has adjacency lists

$$L(0) = \{1\}, \qquad L(1) = \{0, 3\}, \qquad L(2) = \{4\}$$
$$L(3) = \{1\}, \qquad L(4) = \{2\}.$$

The adjacency matrix is

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

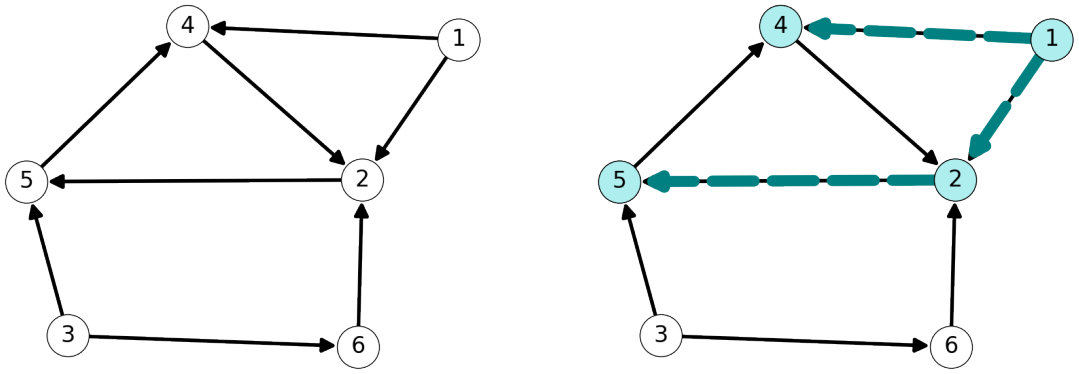
3 Graph Reachability

3.1 The Graph Reachability Problem

Definition 3.1 (The Graph Reachability Problem)**.** Given a directed graph $G = (V, A)$ and a node s. Find the set M of all nodes that are **reachable** from s :

- i.e., all nodes that are connected to s.

Example 3.1. Consider the graph G given in Figure 3.1a. The set of reachable nodes from $s = 1$ is $M = \{1, 2, 4, 5\}$ (see Figure 3.1b).



(a) Directed graph G.

(b) $M = \{1, 2, 4, 5\}$ for $s = 1$.

Figure 3.1: Graph G and reachable set M from node $s = 1$.

3.2 Algorithm Idea

The following steps give an intuitive process for finding reachable nodes from a given node s.

- **Explore** s**:** start from s and follow the arcs in its forward star to find its **neighbors**.
- **Repeat** with each **neighbor of neighbors** of s.

- **Repeat** with **neighbor of neighbors of neighbors** of s, etc.

3.3 Illustration of Idea

Let's apply this idea for the graph G given in Figure 3.1a with source node $s = 1$.

3.3.1 Step 1

The forward star of s is

$$\delta^+(\{1\}) = \{(1,2),(1,4)\}.$$

This implies that 2 and 4 are both reachable from 1. That is, M contains $\{1,2,4\}$. See Figure 3.2a.

3.3.2 Step 2

We need to explore further from nodes 2 and 4. Let's start with 2. The only node adjacent to 2 is node 5. We can conclude that M contains $\{1,2,4,5\}$. See Figure 3.2b.

3.3.3 Step 3

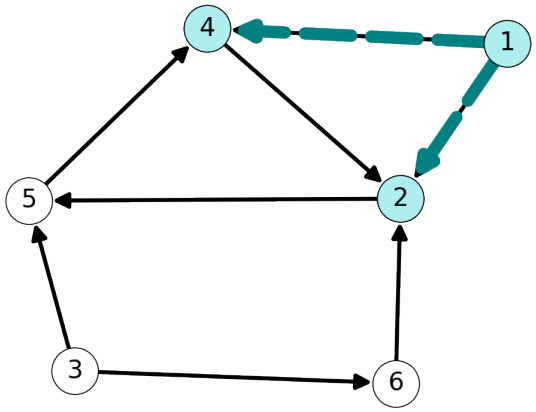
Let's explore from node 4: 2 is the only node adjacent to 4. Our partially computed reachable set M is unchanged. See Figure 3.2c.

3.3.4 Step 4

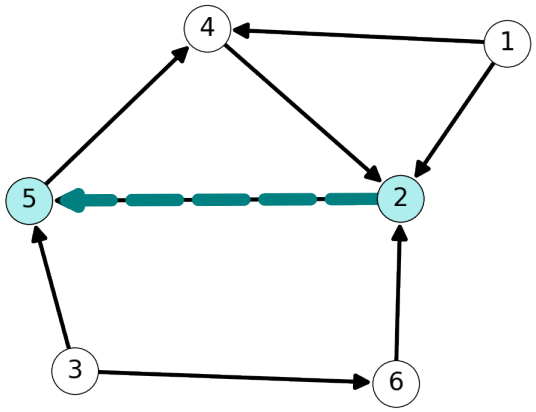
We haven't explored from node 5 yet. Let's do that now. The only arc incident with 5 is $(5,4)$. Thus, 4 is adjacent to 5 and, hence, reachable from 1. We already knew that 4 is in M, so M is unchanged. See Figure 3.2d.

3.3.5 Step 5

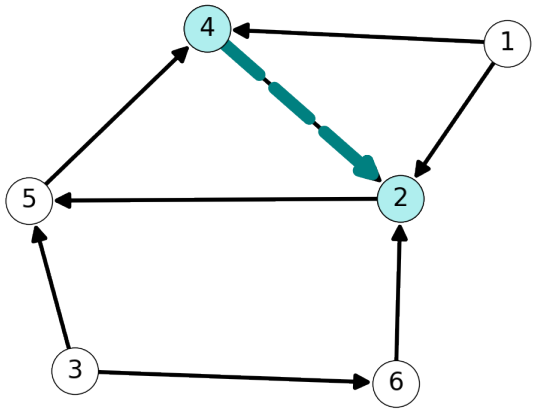
We have a choice of nodes to explore from: $\{2,4,5\}$. We have already explored each of these nodes and appear to be stuck in a loop. Can we conclude that $M = \{1,2,4,5\}$?



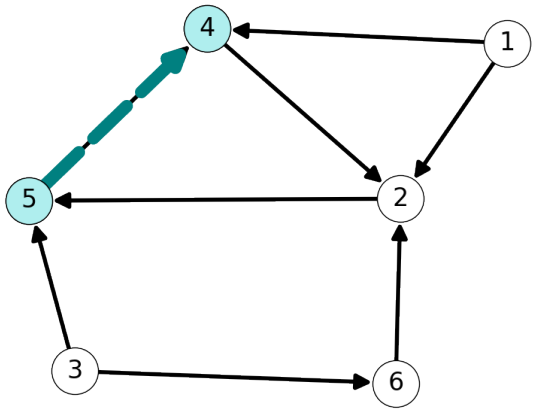
(a) Step 1: $M \supseteq \{1, 2, 4\}$ so far



(b) Step 2: $M \supseteq \{1, 2, 4, 5\}$



(c) Step 3: $M \supseteq \{1, 2, 4, 5\}$



(d) Step 4: $M \supseteq \{1, 2, 4, 5\}$

Figure 3.2: Steps of the intuitive reachability process. We need to revise the algorithm to decide when to terminate.

3.4 Avoiding Cycles

3.4.1 Goal and Notation

We need to avoid exploring nodes we have already explored.

Let's define M as the set of nodes we have **reached** and **explored**.

- When we reach a node, we can check if it is already in M.
- If it is, we've explored it already and shouldn't again.

Let's introduce another set Q as a **queue** of nodes which have been **reached** but **not explored**.

3.4.2 Algorithm Idea

Initialize Q as $Q = \{s\}$. Each iteration:

- Choose vertex $i \in Q$ to explore.
- Add to Q any neighbors of i that are not already in Q or M.
- Add i to M since it has been explored.

If $Q = \{\} = \emptyset$ (empty set), then **stop**:

- We've explored all nodes reachable from s.
- M is the set of nodes reachable from s.

3.4.3 Graph Reachability Algorithm (Pseudocode)

```
Initialize Q = {s} and M = {}.

while Q != {}:

    # select a node i in Q
    Q = Q - {i} # remove i from Q.

    for j in L(i): # j is adjacent to i.

        if (j not in M) and (j not in Q):
            Q = Q + {j} # add j to Q.
```

```
    M = M + {i} # add i to M.
```

3.5 Returning to the Example

Let's consider the graph G given in Figure 3.1a and apply the algorithm to find the set of nodes reachable from $s = 1$.

3.5.1 Iteration 1

We initialize Q and M as

$$Q = \{1\}, \qquad M = \emptyset.$$

Let's explore node 1:

- Node 1 has adjacency list $L(1) = \{2, 4\}$.
- Let's add {2,4} to Q.
- Since we have explored node 1, we remove 1 from Q and add 1 to M.

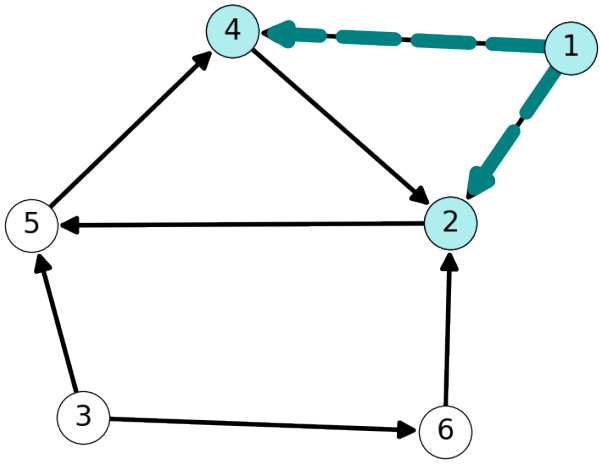


Figure 3.3: Iteration 1: Explore node 1

3.5.2 Iteration 2

After the first iteration, we have

$$Q = \{2, 4\}, \qquad M = \{1\}.$$

We can continue from node 2 or node 4. Let's explore node 2:

- $L(2) = \{5\}$.
- Add 5 to Q.
- Remove 2 from Q and add 2 to M.

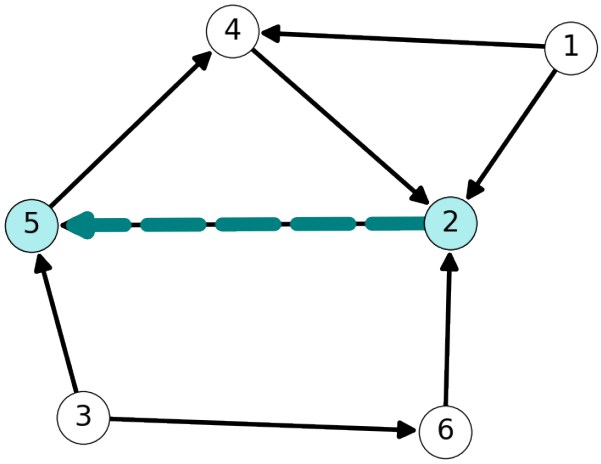


Figure 3.4: Iteration 2: Explore node 2

3.5.3 Iteration 3

We now have

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

Let's explore node 4:

- $L(4) = \{2\}$.
- We already have $2 \in M$; we do not add 2 to Q.
- We remove 4 from Q and add 4 to M.

3.5.4 Iteration 4

After the first three iterations, we have

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

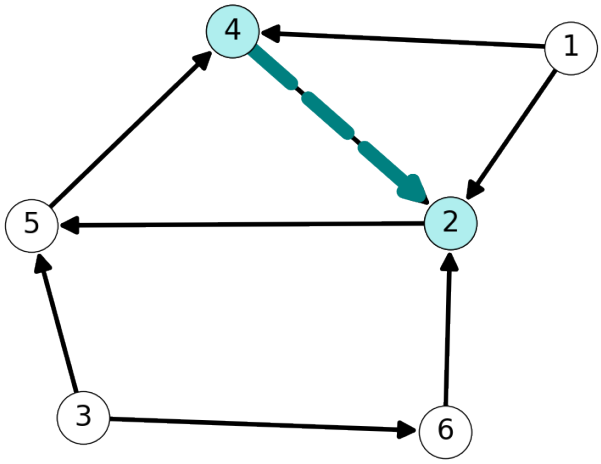


Figure 3.5: Iteration 3: Explore node 4

Exploring node 5, we note:

- $L(5) = 4$.
- Since $4 \in M$ already, we move 5 to M.

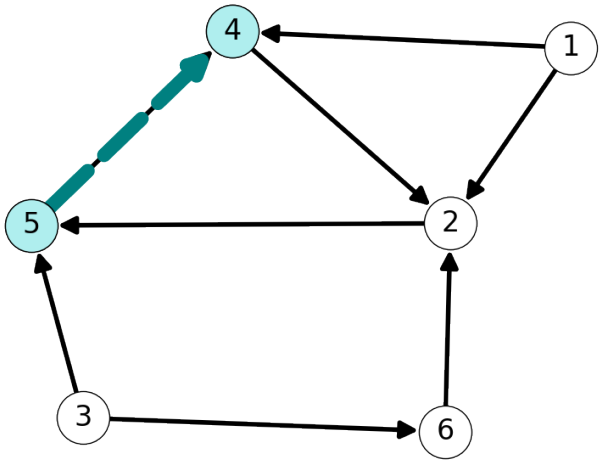


Figure 3.6: Iteration 4: Explore node 5

3.5.5 Iteration 5 – Termination

At this stage Q is empty, so we cannot proceed. We conclude that $M = \{1, 2, 4, 5\}$!

4 Complexity

4.1 Big-O Notation

Definition 4.1 (Big-O Notation)**.** Given $f, g : \mathbf{R} \mapsto \mathbf{R}$, we say

$$f = O(g),$$

i.e., f **is of order** g, if there are n_0 and $c \in \mathbf{R}_+$ such that

$$f(n) \leq c\ g(n) \quad \text{for all } n \geq n_0.$$

Big-O notation captures the asymptotic behaviour of a function f when compared to another function g.

4.1.1 Properties of Big-O Notation

Let f and g be positive functions ($f, g : \mathbf{R} \mapsto \mathbf{R}_+$).

1. If $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \ell \in (0, \infty)$ then

   $$f = O(g) \text{ and } g = O(f).$$

   For example, $f(n) = 2n$ and $g = 3n + \sqrt{n}$ satisfy $f = O(g)$ and $g = O(f)$.

2. If $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$ then

   $$f = O(g) \text{ but } g \neq O(f).$$

   For example, $f(n) = n$ and $g(n) = 3n^3 + 2n$ satisfy $f = O(g)$, but $g \neq O(f)$.

3. Multiplicative and additive constants can be ignored:

   $$af(n) + b = O(f(n)).$$

   For example, $f(n) = 5n^2 + 2 = O(n^2)$.

4. Lower order terms can be ignored: if $g = O(f)$ then

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

   For example, $6n^4 + e^n = O(e^n)$.

4.2 Elementary Operations and Computational Complexity

4.2.1 Computational Complexity

We can count the number of **elementary operations** or **EOs** carried out by an algorithm:

- Arithmetic operations
- Accesses to memory
- Writing operations, etc.

Let $f_A(n)$ and $f_B(n)$ be the **number of EOs required in worst case** (the instance that requires the most EOs) by algorithms A and B to solve P with instance size n.

Definition 4.2. We call $O(f_A)$ and $O(f_B)$ the **computational complexity** of A and B. We choose $O(f_A)$ and $O(f_B)$ to be as small as possible.

4.2.2 The Cobham-Edmonds Thesis (1965)

We say that problem P is **tractable** or **well-solved** if:

1. There is an algorithm A for solving P;
2. A has **polynomial computational complexity**.

4.2.3 Converting EOs and Run-Time

If each EO takes 1 μs (1 microsecond):

n	$f_A(n) = n^2$	$f_A(n) = 2^n$
10	0.1 ms	1 ms
20	0.4 ms	1.0 s
30	0.9 ms	17.9 min
40	1.6 ms	12.7 days
50	2.5 ms	35.7 years
60	3.6 ms	366 centuries

4.2.4 A Comparison of Complexity and Scaling

Assume again that each EO requires 1 μs.

The following figure compares how complexity scales as a function of n for

- **logarithmic**,
- **polynomial,**
- **polylogarithmic,** and
- **exponential** functions.

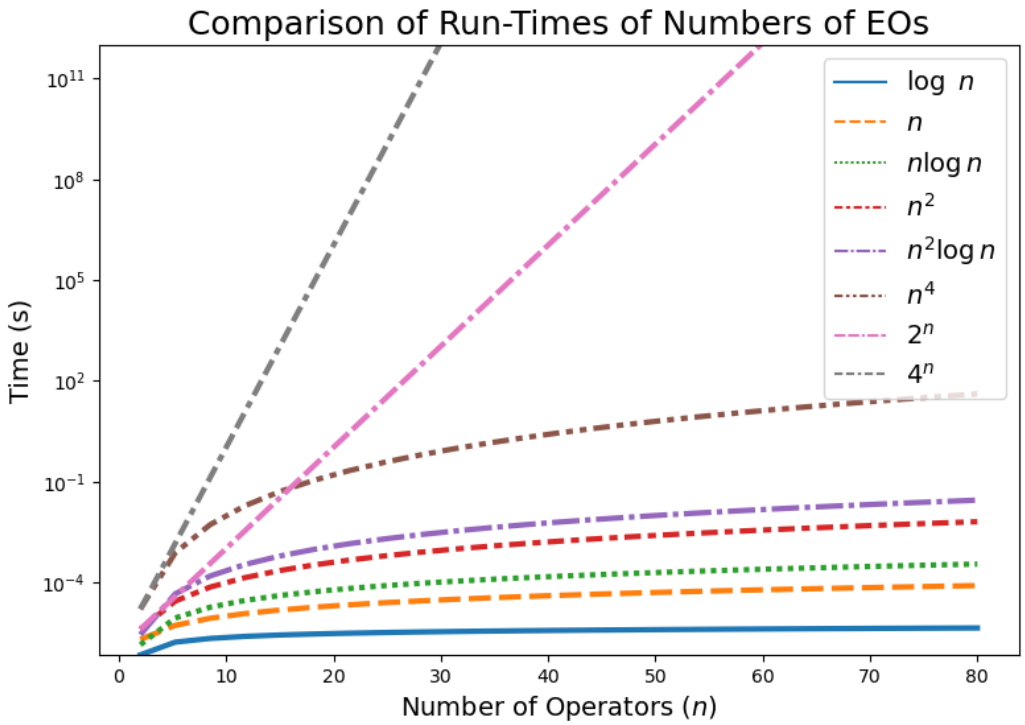


Figure 4.1: Scaling of functions of n

4.3 Examples

4.3.1 Complexity of Solving Quadratic Equations

Recall the following algorithm for solving quadratic equation $ax^2 + bx + c = 0$:

```
Compute u = b*b - 4*a*c

Compute v = 2*a

Compute w = sqrt(u)

Compute x_plus = -(w - b)/v

Compute x_minus = - (w + b)/v
```

Let's count the number of operations used in each step of the algorithm:

1. Computing `u` requires 3 products and one subtraction (4 arithmetic operations total).
2. Computing `v` requires 1 product.
3. Computing `w` requires 1 call to the `sqrt` function, which uses a finite number of arithmetic operations. Let's treat this as 1 operation for the purposes of estimating the complexity of applying the quadratic formula.
4. Each of `x_plus` and `x_minus` require 3 arithmetic operations.

The total number of arithmetic operations used by the quadratic formula is $12 = O(1)$.

4.3.2 Complexity of Graph Reachability

Let's analyse the complexity of the ***graph reachability algorithm*** given below:

```
Initialize Q = {s} and M = {}.

while Q != {}:

    # select a node i in Q
    Q = Q - {i} # remove i from Q.

    for j in L(i): # j is adjacent to i.

        if (j not in M) and (j not in Q):
            Q = Q + {j} # add j to Q.

    M = M + {i} # add i to M.
```

Let's first count the number of operations need for each iteration of the algorithm:

1. Choosing a node `i` in `Q`, removing `i` from `Q`, and adding `i` to `M` requires $O(1)$ operations each.
2. The loop iterating over the adjacency list $L(i)$ runs $|L(i)| \leq n-1$ times. Each occurence of the loop code uses $O(1)$ operations.
3. We perform $|Q| \leq |V| = n$ steps of the outer-most loop.

Therefore, we have the upper bound on complexity

$$O(n(n-1)) = O(|E|).$$

4.3.2.1 An Improved Estimate

The steps of the inner for loop is executed for each node i at most once. This implies that each arc (i, j) is explored at most once. Therefore, the steps of this loop are executed at most m times total.

This implies that the total complexity is

$$O(m) + O(n) = O(m + n).$$

This is much smaller than the previous estimate if the graph is sparse, i.e., $O(m) << O(n(n-1))$.

5 The Shortest Path Problem

5.1 Preliminaries

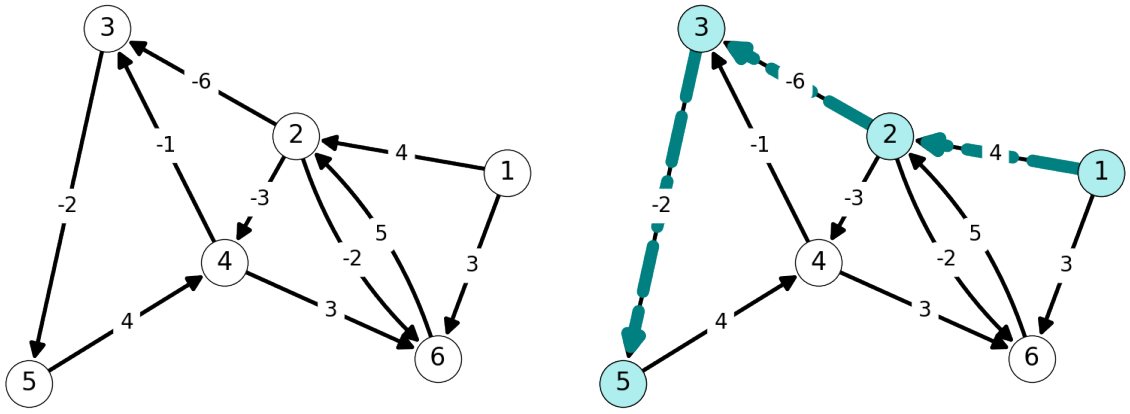
Definition 5.1 (The Shortest Path Problem – Unrestricted Lengths (SPP-U))**.** Given a **directed graph** $G = (V, A)$ with:

- two nodes s and t;
- **length function** $\ell : A \mapsto \mathbf{R}$ (unrestricted in sign).

Find an (s, t)-path with **minimum total length**.

Example 5.1. Consider the graph given in Figure 5.1a. The shortest $(1, 5)$-path in this graph is $(1, 2, 3, 5)$ with value equal to

$$\ell_{12} + \ell_{23} + \ell_{35} = 4 - 6 - 2 = -4.$$



(a) A directed graph G with edge lengths ℓ

(b) The shortest $(1, 5)$-path

Figure 5.1: The shortest $(1, 5)$-path in G is $(1, 2, 3, 5)$ with length -4.

5.1.1 Applications

5.1.1.1 Logistics and Transportation

In this field, we want to find (s,t)-paths minimizing:

- Travel time; ℓ_{ij} is average time traveling along arc (i,j).
- Fuel consumption.
- Likelihood of delays; etc.

5.1.1.2 Minimum Cardinality Path

We can find a path with **minimum number of arcs** by assigning $\ell_{ij} = 1$ for all $(i,j) \in A$.

5.1.1.3 Component of Other Algorithm

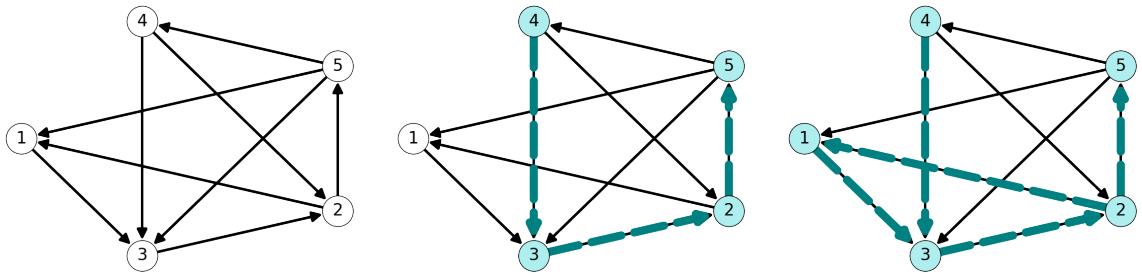
SPP appears as a **subproblem** when solving other combinatorial optimization problems (more details later).

5.1.2 Simple Paths and Cycles

Definition 5.2. An (s,t)-path P is **simple** if it does not visit the same node twice.

Note: if P visits the same node twice then P must contain at least one cycle.

Example 5.2. Figure 5.2 gives an example of a simple path and a not simple path within a directed graph.



(a) Graph G (b) A simple path (c) A not-simple path

Figure 5.2: Simple and not simple $(4,5)$-paths in directed graph G.

5.1.3 Unboundedness and Negative Cycles

Theorem 5.1. *Suppose that G contains an (s,t)-path P with a* ***negative length cycle****. Then* ***SPP-U*** *is* ***unbounded****.*

Proof. Suppose that C is a negative length with length

$$\ell_C = \sum_{ij \in C} \ell_{ij} < 0.$$

Suppose further that P is an (s,t)-path that traverses C exactly k times for some integer $k > 0$. That is, C is contained in P exactly k times.

We can always augment this path to get a path with strictly smaller length. Indeed, consider the path $\tilde{P}$ which contains all arcs of P, but traverses C exactly $k+1$ times. Then the length of $\tilde{P}$ satisfies

$$\text{length } \tilde{P} = \text{length } P + \ell_C < \text{length } P$$

since $\ell_C < 0$. □

Theorem 5.2. *If no (s,t)-path contains a negative-length cycle, then G admits a* ***simple shortest*** *(s,t)-****path****.*

If no (s,t)-path in G contains a negative length cycle, then we can solve the **SPP-U** by restricting our search to simple paths.

Proof. Let P be a shortest (s,t)-path in G. Let's also assume that every cycle C in P has nonnegative length $\ell_C \geq 0$.

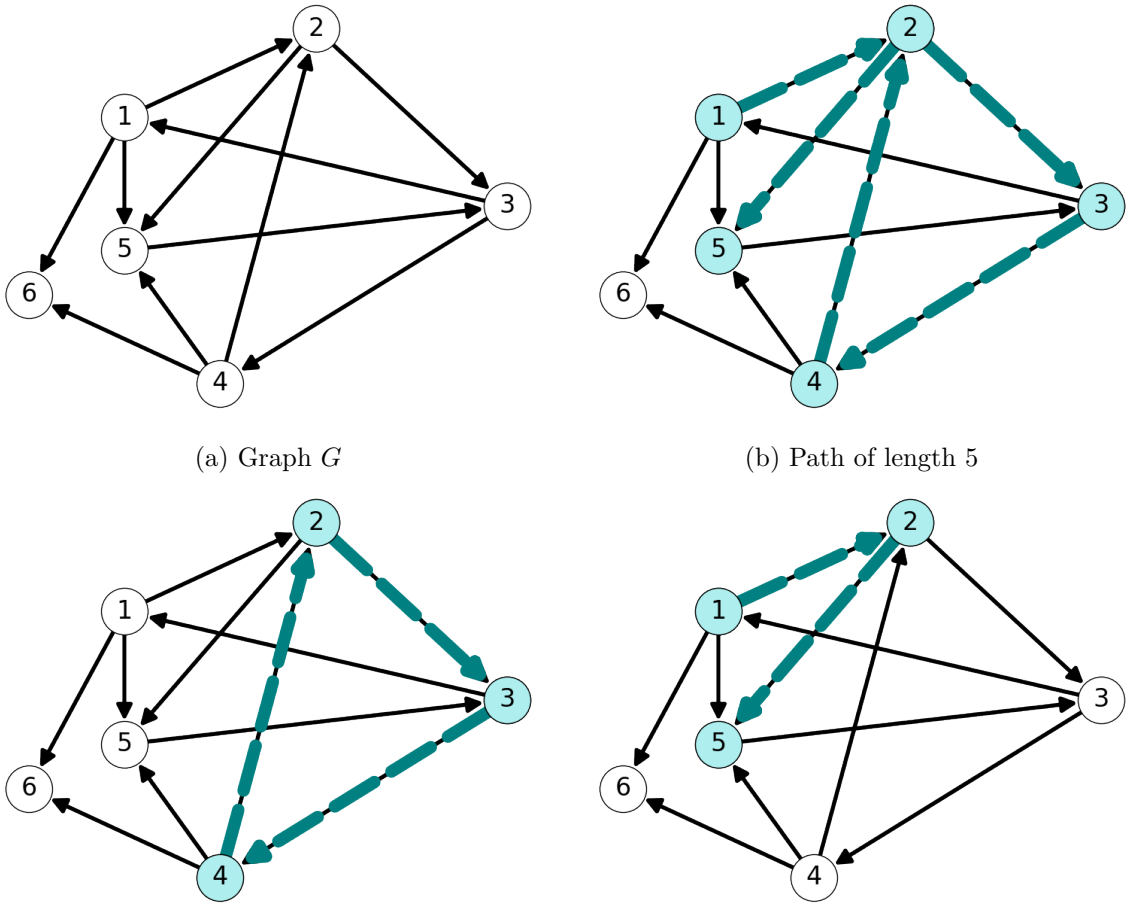
Let's assume that P contains a cycle C starting and ending with node u. That is, we can think of P as the union of the directed (s,u)-path P_{su}, the cycle C, and the (u,t)-path P_{ut}. Let ℓ^* denote the value of this path.

Now consider the (s,t)-path given by following P_{su} to u, then P_{ut} to t. This is the path obtained by removing C from P. This gives an (s,t)-path with length

$$\ell^* - \ell_C \leq \ell^*.$$

Thus, removing cycle C from P does not increase the length of the path. We can repeat this process until we have obtained a simple path with minimum path length ℓ^*, or we obtain a simple path with length strictly less than ℓ^* (a contradiction). □

Example 5.3. To illustrate this phenomena, consider the graph given in Figure 5.3. This graph has $(1,5)$-path $(1,2,3,4,2,5)$ with length 5 (assuming all arcs have length 1). However, this path contains the cycle $C = (2,3,4,2)$. Removing the cycle C gives the shorter $(1,5)$-path $(1,2,5)$ (with length 2).



(a) Graph G

(b) Path of length 5

(c) Cycle C

(d) Path with cycle removed

Figure 5.3: Illustration of the cycle removal process to obtain a shorter path.

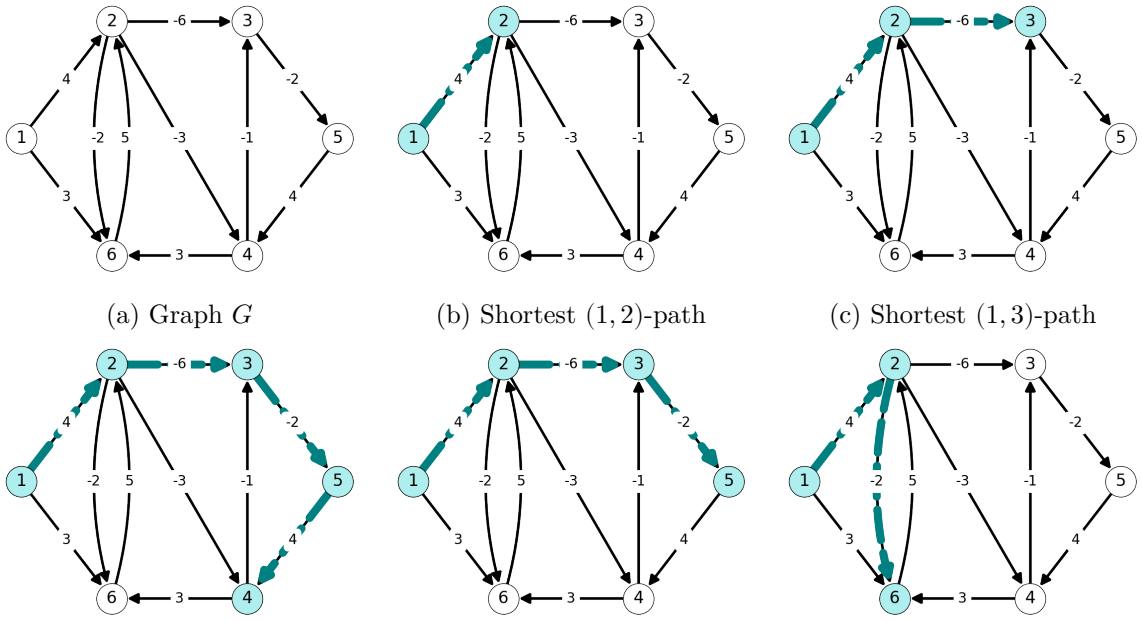
5.2 The Bellman-Ford Algorithm

5.2.1 Single Source Shortest Path Problem – Unrestricted Lengths (SSPP-U)

Definition 5.3. Given a **directed graph** $G = (V, A)$ with **length function** $\ell : A \mapsto \mathbf{R}$ (unrestricted in sign).

The **Single Source Shortest Path Problem (SSPP-U)** aims to find an (s, t)-path with **minimum total length** for every node t in $V \setminus \{s\}$.

Example 5.4. Figure 5.4 gives the shortest $(1, t)$-paths in the graph given in Figure 5.4a for $t = 2, 3, 4, 5, 6$.



(a) Graph G (b) Shortest $(1, 2)$-path (c) Shortest $(1, 3)$-path

(d) Shortest $(1, 4)$-path (e) Shortest $(1, 5)$-path (f) Shortest $(1, 6)$-path

Figure 5.4: Shortest $(1, t)$-paths in graph G.

5.2.2 Subpath Optimality

The following theorem characterizes a very useful properties of shortest paths: any subpath of a shortest path is itself a shortest path.

Theorem 5.3. *Let $P = (s, i_2), (i_2, i_3), \ldots, (i_{k-1}, t)$ be a* ***shortest*** *(s,t)-****path****.*

Consider pair of nodes i_u, i_v visited by P with $u < v$.

Then the subpath from i_u to i_v is a ***shortest*** *(i_u, i_v)* ***path****.*

Proof. Suppose, on the contrary, that the subpath S from i_u to i_v in a shortest (s,t)-path is ***not*** the shortest (i_u, i_v)-path. In particular, suppose that T is the shortest (i_u, i_v)-path.

We can construct another (s,t)-path *tildeP* using P and T. Indeed, let $\tilde{P} = P \setminus S \cup T$ be the path obtained by removing S from P and adding T. Then $\tilde{P}$ is also a (s,t)-path. The length of $\tilde{P}$ is equal to

$$|P| - |S| + |T| < |P| - |S| + |S| = |P|$$

since $|T| < |S|$. This implies that $\tilde{P}$ is shorter than P; this is a contradiction. Therefore, S must be the shortest (i_u, i_v)-path. □

5.2.3 Shortest Paths of Fixed Length

Definition 5.4. For all $i \in V$, we define $f_k(i)$ as the **length of a shortest (s,i)-path containing at most k arcs**,

We set $f_k(i) = \infty$ if there is no (s,i)-path with length at most k.

Lemma 5.1. *If $k = n - 1$, then $f_{n-1}(i)$ is the* ***length of a shortest*** *(s,i)-****path*** *(without restriction on the number of arcs).*

Proof. We can ignore paths that aren't simple. On the other hand, simple paths contain at most n nodes. Indeed, a simple path does not contain a loop and, hence, visits at most n nodes. Therefore, the shortest (s,i)-path contains at most $n-1$ arcs. □

5.2.4 The Bellman-Ford Theorem

The following theorem is the basis for our algorithm for calculating shortest paths.

Theorem 5.4. *$f_k(i)$ can be computed* **recursively** *as*

$$f_k(i) = \min\left\{ f_{k-1}(i),\ \min_{(j,i)\in\delta^-(i)} \left\{ f_{k-1}(j) + \ell_{ji} \right\} \right\}.$$

Proof. Assume that $f_{k-1}(i)$ has been computed for all $i \in V \setminus \{s\}$. The shortest (s,i)-path with at most k arcs …

1. is the shortest (s, i)-path with at most $k-1$ arcs; or
2. contains one more arc than the shortest (s, i)-path with at most $k-1$ arcs.

In the second case, we can decompose the shortest (s, i)-path as:

- shortest (s, j)-path with $k-1$ arcs for some node j; and
- arc (j, i).

We choose node j so that j gives

$$\min_{(q,i)\in\delta^-(i)} f_{k-1}(q) + \ell_{qi}.$$

□

5.2.5 The Bellman-Ford Algorithm

5.2.5.1 Setup

Definition 5.5. Let $P_k(i)$ be the **predecessor** of i in the shortest (s, i)-path with at most k arcs found by the algorithm (so far).

We will maintain **two tables**:

- One encoding $f_k(i)$ for each i and k;
- The other encoding $P_k(i)$ for all i and k.

5.2.5.2 Updates

The algorithm calculates $f_k(i)$ and $P_k(i)$ for all $i \in V$ recursively from $k = 1$ to $k = n-1$.

We'll update $f_k(i)$ from $f_{k-1}(i)$ by:

- Initially setting $f_k(i) = f_{k-1}(i)$;
- Scanning all arcs (j, i) in $\delta^-(i)$ and update $f_k(i)$ if necessary.

5.2.5.3 The Bellman-Ford Algorithm (Pseudocode)

```
f_0(s) = 0 and P_0(s) = ~
for i in V \{s}:
    f_0(i) = +inf and  P_0(i) = ~

# Calculate shortest paths of length k.
for k in {1, 2, ..., n-1}:

    # Update shortest si-path.
    for i in V:
        f_k(i) = f_{k-1}(i)
        P_k(i) = P_{k-1}(i)

        # Check each edge incident at i.
        for (j,i) in delta^{-}(i):
            # Update if length-k path is shorter than k-1.
            if f_{k-1}(j) + l_{ji} < f_k(i)
                f_k(i) = f_{k-1}(j) + l_{ji}
                P_k(i) = j
```

5.2.6 Example

5.2.6.1 Iteration 1 ($k = 1$)

Note that nodes 2 and 6 are adjacent to 1. Thus, they are reachable from 1 by a path with length at most 1. The *only* paths from 1 to another node are the single arcs $(1, 2)$ and $(1, 6)$, of lengths 4 and 3 respectively. Figure 5.5 highlights these paths, and we update the tables Table 5.1 and Table 5.2 accordingly.

Table 5.1: Lengths of shortest paths of length at most 1.

	1	2	3	4	5	6
f_0	0	∞	∞	∞	∞	∞
f_1	0	4	∞	∞	∞	3

Table 5.2: Predecessors in shortest paths of length at most 1.

	1	2	3	4	5	6
P_0	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
P_1	$\sim$	1	$\sim$	$\sim$	$\sim$	1

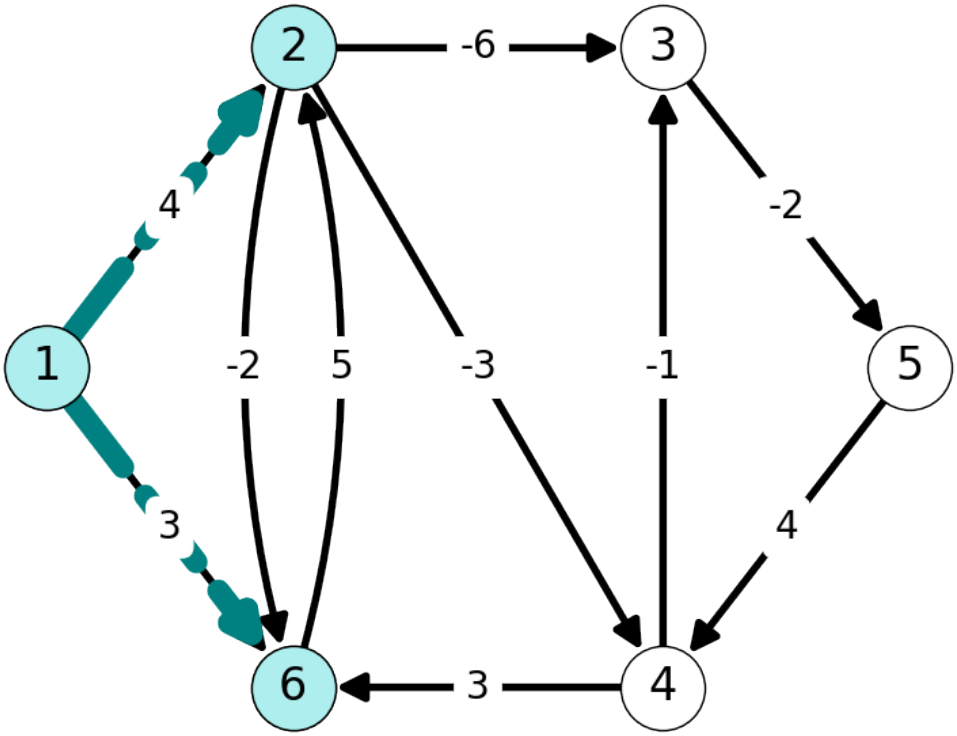


Figure 5.5: Shortest Paths of length $k = 1$

5.2.6.2 Iteration 2 ($k = 2$)

Nodes 3 and 4 can both be reached from 1 using a path of length 2 (with lengths -2 and 1 respectively). Similarly, we have path $(1, 2, 6)$ of length 2:

$$\ell_{12} + \ell_{26} = f_1(2) + \ell_{26} = 4 - 2 = 2 < f_1(6) = 3.$$

This implies that the path $(1, 2, 6)$ is the shortest $(1, 6)$-path with at most 2 arcs. We update the shortest path and predecessor tables below.

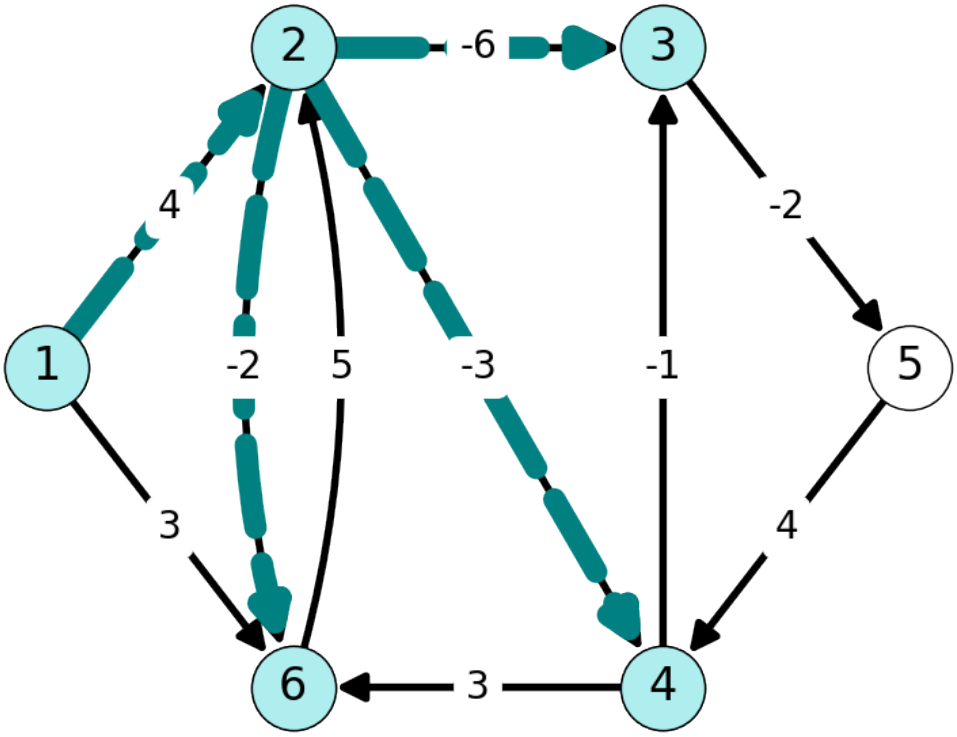


Figure 5.6: Shortest Paths of length at most $k = 2$

Table 5.3: Lengths of shortest paths of length at most 2.

	1	2	3	4	5	6
f_0	0	∞	∞	∞	∞	∞
f_1	0	4	∞	∞	∞	3
f_2	0	4	-2	1	∞	2

Table 5.4: Predecessors in shortest paths of length at most 2.

	1	2	3	4	5	6
P_0	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
P_1	$\sim$	1	$\sim$	$\sim$	$\sim$	1
P_2	$\sim$	1	2	2	$\sim$	2

5.2.6.3 Iteration 3 ($k = 3$)

Node 5 is reachable from 1 by the path $(1, 2, 3, 5)$ using 3 arcs; this path has length -4. On the other hand, every other (s, i)-path consisting of 3 arcs is longer than the shortest (s, i)-path of length at most 2.

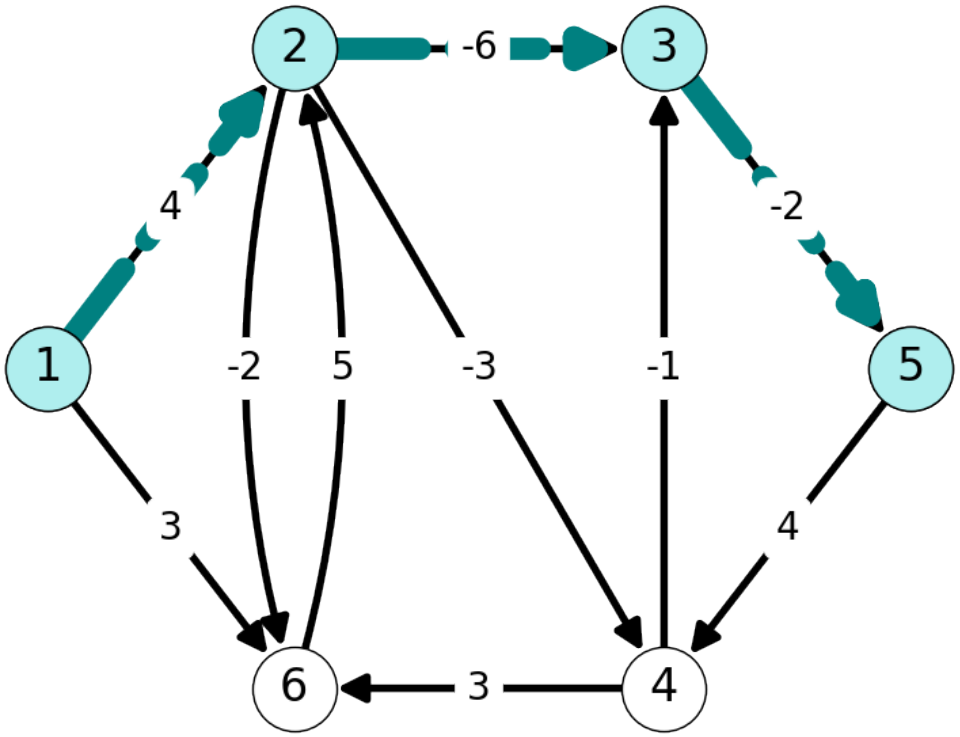


Figure 5.7: Shortest Paths of length at most $k = 3$

Table 5.5: Lengths of shortest paths of length at most 3.

	1	2	3	4	5	6
f_0	0	∞	∞	∞	∞	∞
f_1	0	4	∞	∞	∞	3
f_2	0	4	-2	1	∞	2
f_3	0	4	-2	1	-4	2

Table 5.6: Predecessors in shortest paths of length at most 3.

	1	2	3	4	5	6
P_0	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
P_1	$\sim$	1	$\sim$	$\sim$	$\sim$	1
P_2	$\sim$	1	2	2	$\sim$	2
P_3	$\sim$	1	2	2	3	2

5.2.6.4 Iteration 4 ($k = 4$)

Note that we have the $(1, 4)$-path $(1, 2, 3, 5, 4)$ containing $k = 4$ arcs. This path has length

$$f_3(5) + \ell_{54} = -4 + 4 = 0 < f_3(4) = 1 = f_2(4).$$

We set $f_4(4) = 0$ and $P_4(4) = 5$.

After checking all other paths containing $k = 4$ arcs, we do not change the list of shortest paths.

Table 5.7: Lengths of shortest paths of length at most 4.

	1	2	3	4	5	6
f_0	0	∞	∞	∞	∞	∞
f_1	0	4	∞	∞	∞	3
f_2	0	4	-2	1	∞	2
f_3	0	4	-2	1	-4	2
f_4	0	4	-2	0	-4	2

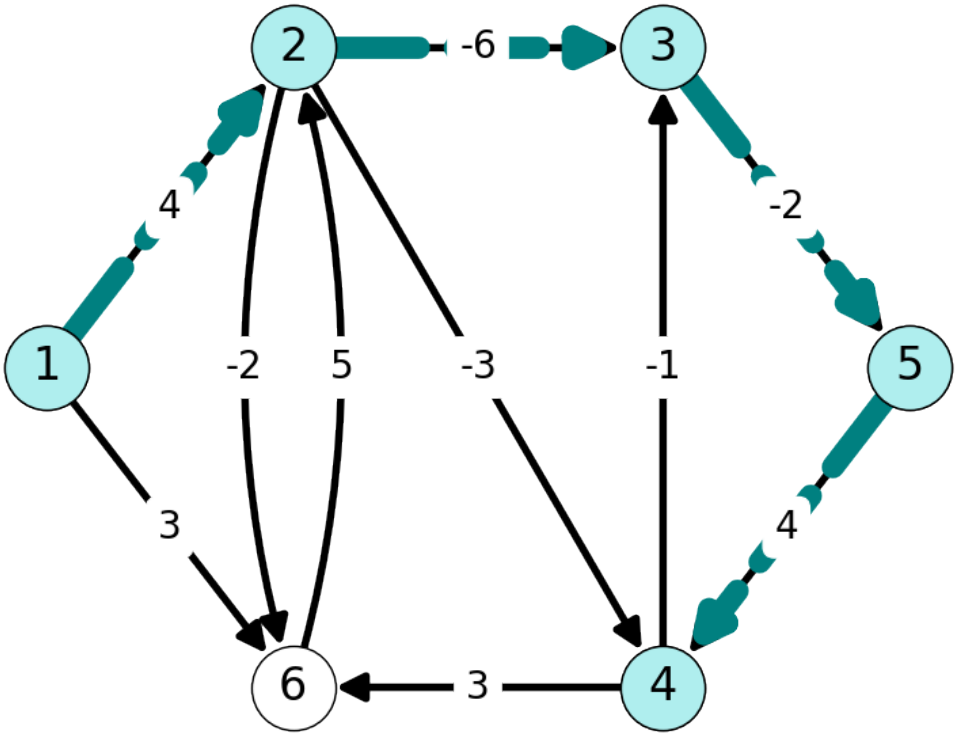


Figure 5.8: Shortest Paths of length at most $k = 4$

Table 5.8: Predecessors in shortest paths of length at most 4.

	1	2	3	4	5	6
P_0	0	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
P_0	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$	$\sim$
P_1	$\sim$	1	$\sim$	$\sim$	$\sim$	1
P_2	$\sim$	1	2	2	$\sim$	2
P_3	$\sim$	1	2	2	3	2
P_4	$\sim$	1	2	5	3	2

5.2.6.5 Termination

The shortest path using at most 4 arcs are also shortest paths using at most 5 arcs for this graph. Since $k = 5 = n - 1$, we have found the shortest paths of any length by @lem.

5.2.7 Complexity of the Bellman-Ford Algorithm

Let's analyses how many operations are needed by the Bellman-Ford Algorithm.

```
f_0(s) = 0 and P_0(s) = ~
for i in V \{s}:
    f_0(i) = +inf and  P_0(i) = ~

# Calculate shortest paths of length k.
for k in {1, 2, ..., n-1}:

    # Update shortest si-path.
    for i in V:
        f_k(i) = f_{k-1}(i)
        P_k(i) = P_{k-1}(i)

        # Check each edge incident at i.
        for (j,i) \in delta^{-}(i):
            # Update if length-k path is shorter than k-1.
            if f_{k-1}(j) + l_{ji} < f_k(i)
                f_k(i) = f_{k-1}(j) + l_{ji}
                P_k(i) = j
```

Initiailization (Lines 1-3) requires $O(n)$ elementary operations.

The loop from Line 9 to Line 18 is repeated for each $i \in V$ ($O(n)$ times) and consists of the following steps for each i:

- Initializing $f_k(i) = f_{k-1}(i)$ and $P_k(i) = P_{k-1}(i)$ (requires $O(1)$ operations).
- Checking if each arc (j, i) with tail i yields a shorter path with k arcs:
  - Comparing $f_{k-1}(j) + \ell_{ji}$ with $f_k(i)$ requires $O(1)$ operations for each j.
  - We have $|\delta^-(i)| = O(n)$ of these arcs.

It follows that the loop from Lines 9–18 requires

$$O(n)\Big(O(1) + O(n)\Big) = O(n^2)$$

for each $k \in \{1, 2, \dots, n-1\}$. Taking the total over all k, the Bellman-Ford Algorithm requires $O(n^3)$ elementary operations.

5.2.7.1 Complexity in Terms of Arcs

Each arc $(j, i) \in A$ is considered exactly once for each $k \in \{1, 2, \dots, n-1\}$ in the loop from Lines 9–18. This implies that the steps of this loop require $O(|A| + n) = O(m + n)$ elementary operations. This implies that the total complexity of the Bellman-Ford Algorithm require

$$O(nm + n^2)$$

elementary operations, which is much smaller than $O(n^3)$ if $m << n^2$.

5.2.8 Detecting Negative Length Cycles

Recall: SPP-U (and **SSPP-U**) are **unbounded** if the graph contains a **negative length** cycle. Indeed, we can endlessly apply rounds of update operations and reduce the shortest path lengths $f_k(i)$ indefinitely.

We can detect whether the problem is unbounded by running one more iteration of the algorithm beyond iteration $k = n - 1$.

- If the graph contains a negative-length cycle then an update will take place.
- This indicates the problem is unbounded and we can stop the algorithm.

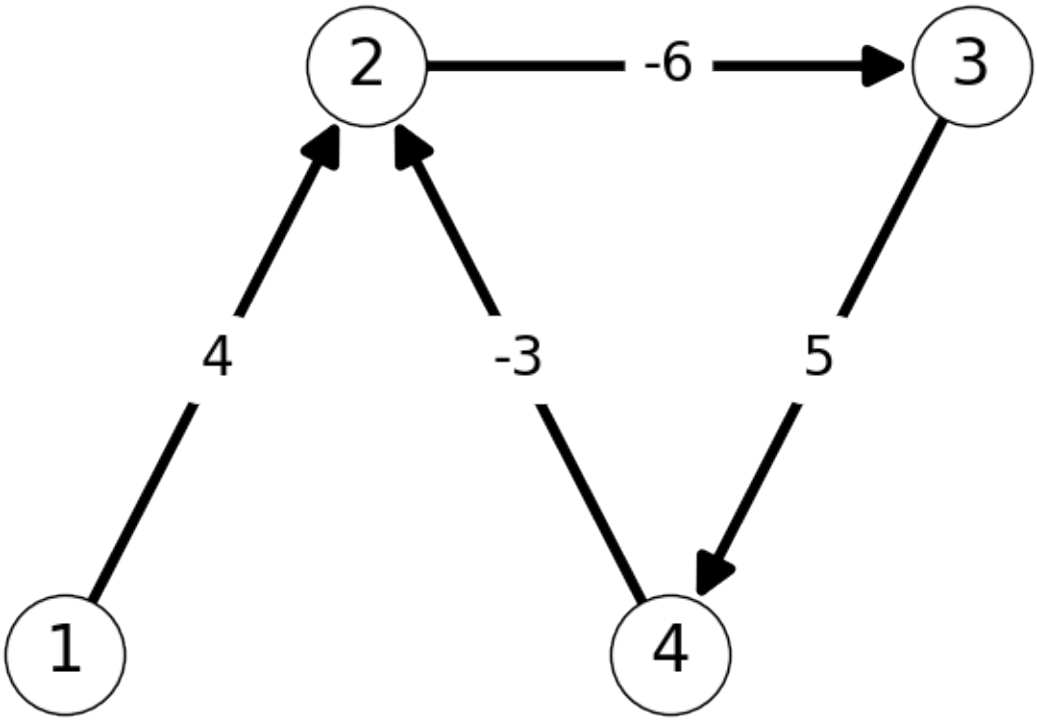


Figure 5.9: Graph with negative cycle

5.2.8.1 Example

Consider the graph given in Figure 5.9. This graph has cycle $C = (2, 3, 4, 2)$ with length $-4 < 0$.

Let's apply the Bellman-Ford algorithm to find the shortest $(1, t)$-paths in this graph.

5.2.8.1.1 Iteration 1 ($k = 1$)

The only node adjacent to 1 is 2. Thus, the

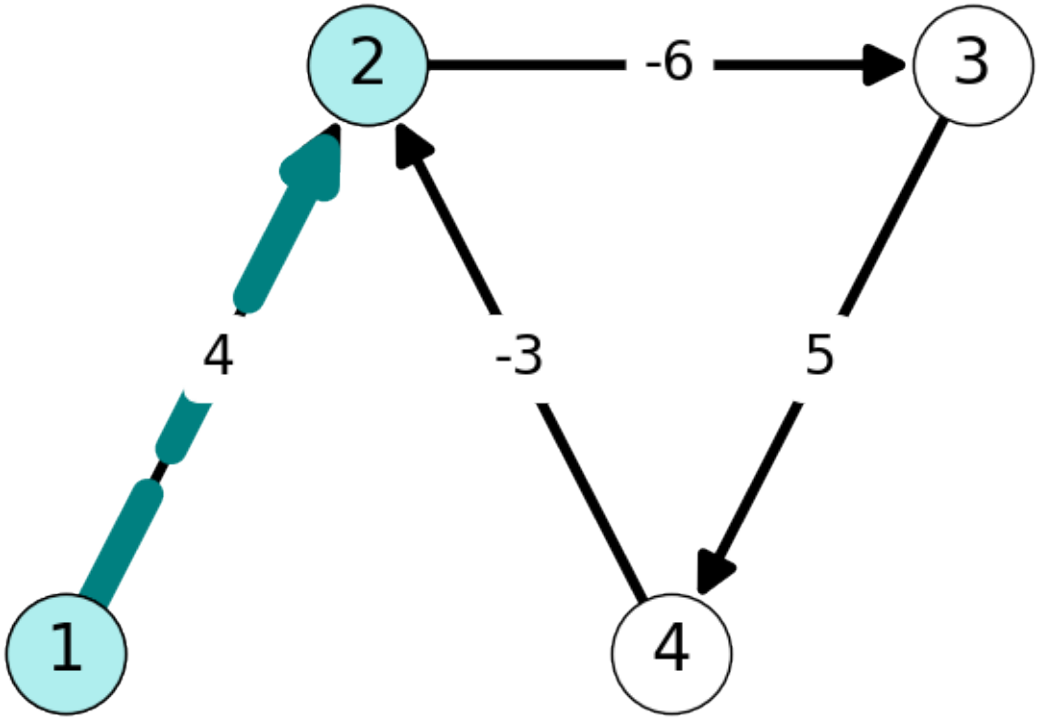


Figure 5.10: Shortest Paths of length at most $k = 1$

Table 5.9: Length of shortest paths of length at most 1 in graph with negative cycle.

	1	2	3	4
$f_0(i)$	0	∞	∞	∞
$f_1(i)$	0	4	∞	∞

Table 5.10: Predecessors in shortest paths of length at most 1 in graph with negative cycle.

	1	2	3	4
$P_0(i)$	$\sim$	$\sim$	$\sim$	$\sim$
$P_1(i)$	$\sim$	1	$\sim$	$\sim$

5.2.8.1.2 Iteration 2 ($k = 2$)

Next, we can reach 3 from 1 via a path consisting of 2 arcs. This path has length -2.

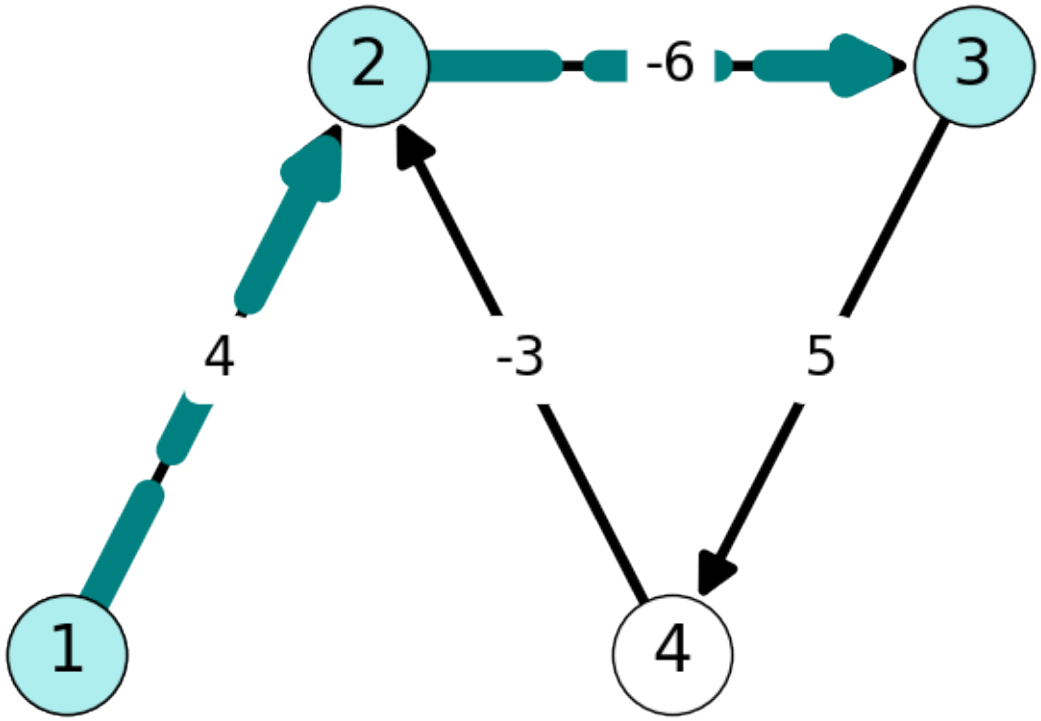


Figure 5.11: Shortest Paths of length at most $k = 2$

Table 5.11: Length of shortest paths of length at most 2 in graph with negative cycle.

	1	2	3	4
$f_0(i)$	0	∞	∞	∞
$f_1(i)$	0	4	∞	∞
$f_2(i)$	0	4	-2	∞

Table 5.12: Predecessors in shortest paths of length at most 2 in graph with negative cycle.

	1	2	3	4
$P_0(i)$	$\sim$	$\sim$	$\sim$	$\sim$
$P_1(i)$	$\sim$	1	$\sim$	$\sim$
$P_2(i)$	$\sim$	1	2	$\sim$

5.2.8.1.3 Iteration 3 ($k = 3$)

Node 4 is the only node that is reachable from 1 by a path containing 3 arcs; the path $(1, 2, 3, 4)$ has length 3.

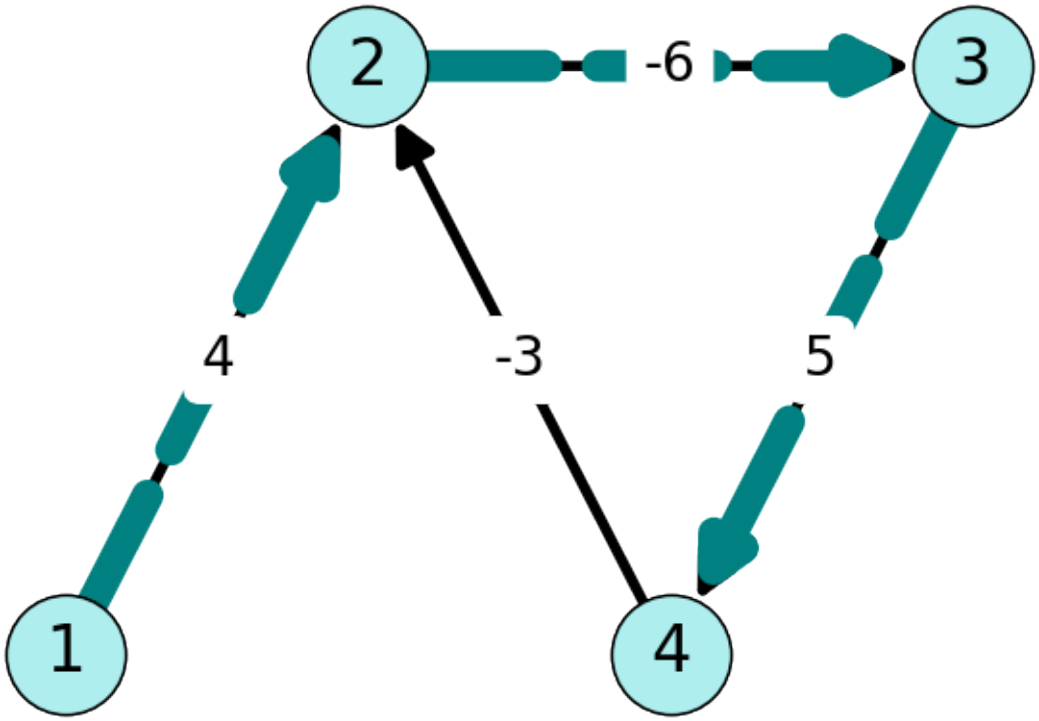


Figure 5.12: Shortest Paths of length $k = 3$

Table 5.13: Length of shortest paths of length at most 3 in graph with negative cycle.

	1	2	3	4
$f_0(i)$	0	∞	∞	∞
$f_1(i)$	0	4	∞	∞
$f_2(i)$	0	4	-2	∞

	1	2	3	4
$f_3(i)$	0	4	-2	3

Table 5.14: Predecessors in shortest paths of length at most 3 in graph with negative cycle.

	1	2	3	4
$P_0(i)$	$\sim$	$\sim$	$\sim$	$\sim$
$P_1(i)$	$\sim$	1	$\sim$	$\sim$
$P_2(i)$	$\sim$	1	2	$\sim$
$P_3(i)$	$\sim$	1	2	3

5.2.8.1.4 Iteration 4 ($k = 4$)

Let's try one more iteration!

Note that

$$f_3(4) + \ell_{42} = 3 - 3 = 0 < f_2(2).$$

Table 5.15: Length of shortest paths of length at most 4 in graph with negative cycle.

	1	2	3	4
$f_0(i)$	0	∞	∞	∞
$f_1(i)$	0	4	∞	∞
$f_2(i)$	0	4	-2	∞
$f_3(i)$	0	4	-2	3
$f_4(i)$	0	0	-2	3

Table 5.16: Predecessors in shortest paths of length at most 4 in graph with negative cycle.

	1	2	3	4
$P_0(i)$	$\sim$	$\sim$	$\sim$	$\sim$
$P_1(i)$	$\sim$	1	$\sim$	$\sim$
$P_2(i)$	$\sim$	1	2	$\sim$
$P_3(i)$	$\sim$	1	2	3
$P_4(i)$	$\sim$	4	2	3

This implies that the path $(1, 2, 3, 4, 2)$ with 5 arcs has smaller total length than the path $(1, 2)$. This contradicts Lemma 5.1 and, hence, implies that the shortest path problem is unbounded

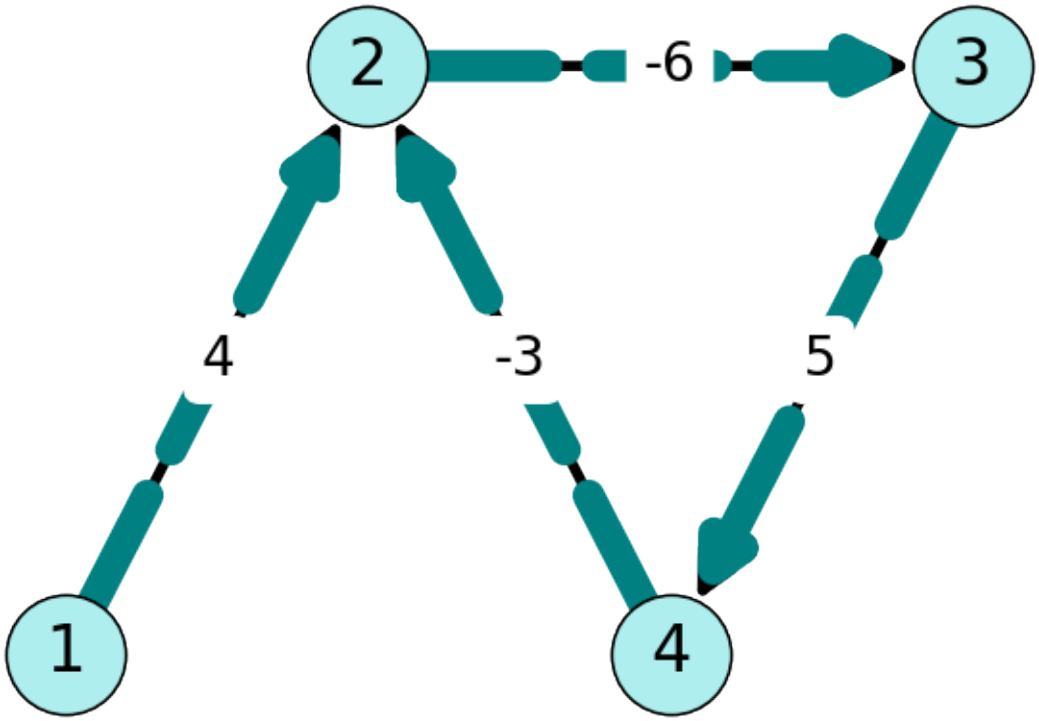


Figure 5.13: Shortest Paths of length $k = 4$

for this graph. Indeed, further iterations will lead further decrease in $f_k(i)$ for i in the negative length cycle $(2, 3, 4, 2)$. We can stop the Bellman-Ford Algorithm after $k = n$ iterations and declare the problem instance unbounded.

6 Minimum Cost Spanning Trees

6.1 Preliminaries

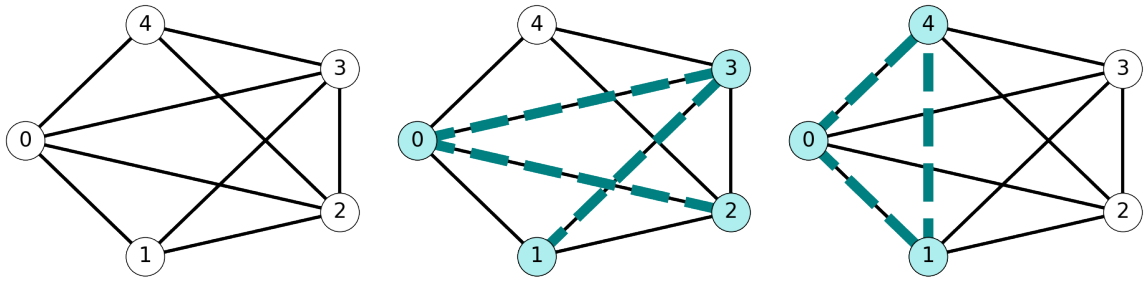
6.1.1 Subgraphs

Definition 6.1. Let $G = (V, E)$ be an **undirected graph**.

We call $G_S = (V_S, E_S)$ a **subgraph** of G if

1. G_S is a graph;
2. $V_S \subseteq V$ and $E_S \subseteq E$.

Example 6.1. Consider the graph $G = (V, E)$ given in Figure 6.1a. The graph given in Figure 6.1b is a subgraph of G. However, the graph given in Figure 6.1c is *not* a subgraph of G since $\{1, 4\} \notin E$.



(a) Graph G (b) A subgraph (c) Not a subgraph

Figure 6.1: A subgraph of G and a graph that is not a subgraph of G.

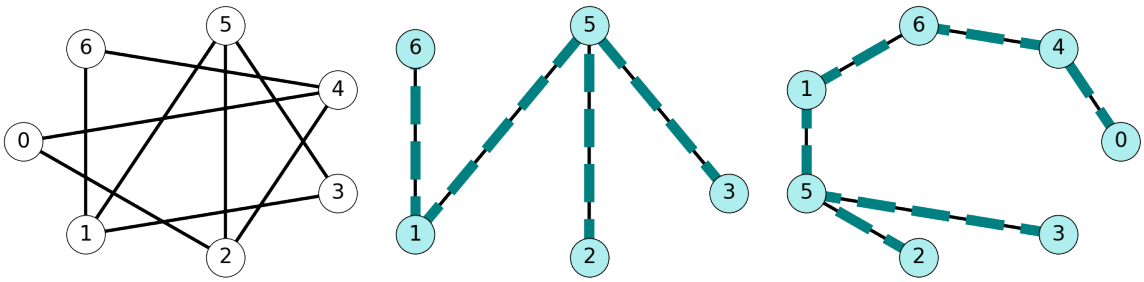
6.1.2 Trees

Definition 6.2. Let $G = (V, E)$ be an **undirected graph**.

The **subgraph** $G_T = (V_T, E_T)$ is a **tree** if

1. G_T is a **connected**;
2. G_T is **acyclic**, i.e., contains no cycles.

A tree G_T is a **spanning tree** if $V_T = V$ (G_T spans/reaches all nodes in V).



(a) Graph G (b) A Tree (c) A Spanning Tree

Figure 6.2: Trees in given graph G.

Example 6.2. Consider the graph $G = (V, E)$ given in Figure 6.2a. The subgraph given in Figure 6.2b is a connected and acyclic; therefore, it is a tree. On the otherhand, the subgraph given in Figure 6.2c is a tree and has node set V; therefore, it is a *spanning* tree.

6.1.3 Motivating Example

We want to build a new high-speed network at the University:

1. should connect all buildings; while
2. costing as little as possible.

We can model this problem as that of finding a ***minimum cost spanning tree***.

Consider the graph G with:

- Buildings as vertices;
- Potential connections as edges.

We want to find a **connected, acyclic subgraph** to minimize cost while ensuring all buildings are connected. Such a tree is highlighted in red in Figure 6.3.

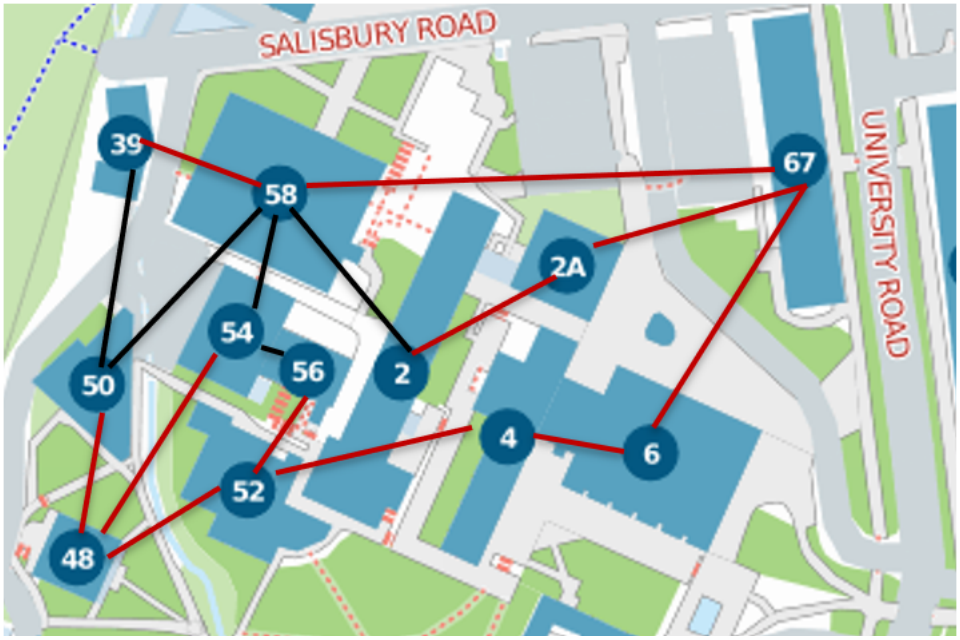


Figure 6.3: Campus map with minimum spanning tree

6.1.4 The Minimum Cost Spanning Tree (MST) Problem

Definition 6.3. Given an undirected graph $G = (V, E)$ and **cost function** $c : E \mapsto \mathbf{R}$.

The **minimum cost spanning tree problem** aims to find a **spanning tree** $G_T = (V_T, E_T)$ of **minimum total cost**:

$$\min \; c(E_T) = \sum_{e \in E_T} c_e.$$

Theorem 6.1 (Cayley – 1889)**.** *A **complete undirected graph** with n nodes contains n^{n-2} spanning trees.*

Noncomplete graphs contain fewer spanning trees, but the number is still exponentially large in n. This implies that we cannot solve MST using complete enumeration for even small graphs.

6.1.5 Leaves

Definition 6.4. The nodes of a graph with **degree 1** are called **leaves**.

Theorem 6.2. *A tree contains at least one leaf.*

Proof. Suppose, on the contrary, that tree T does not contain a leaf. Then every node in T has degree at least 2. This implies that T has a cycle; a contradiction. □

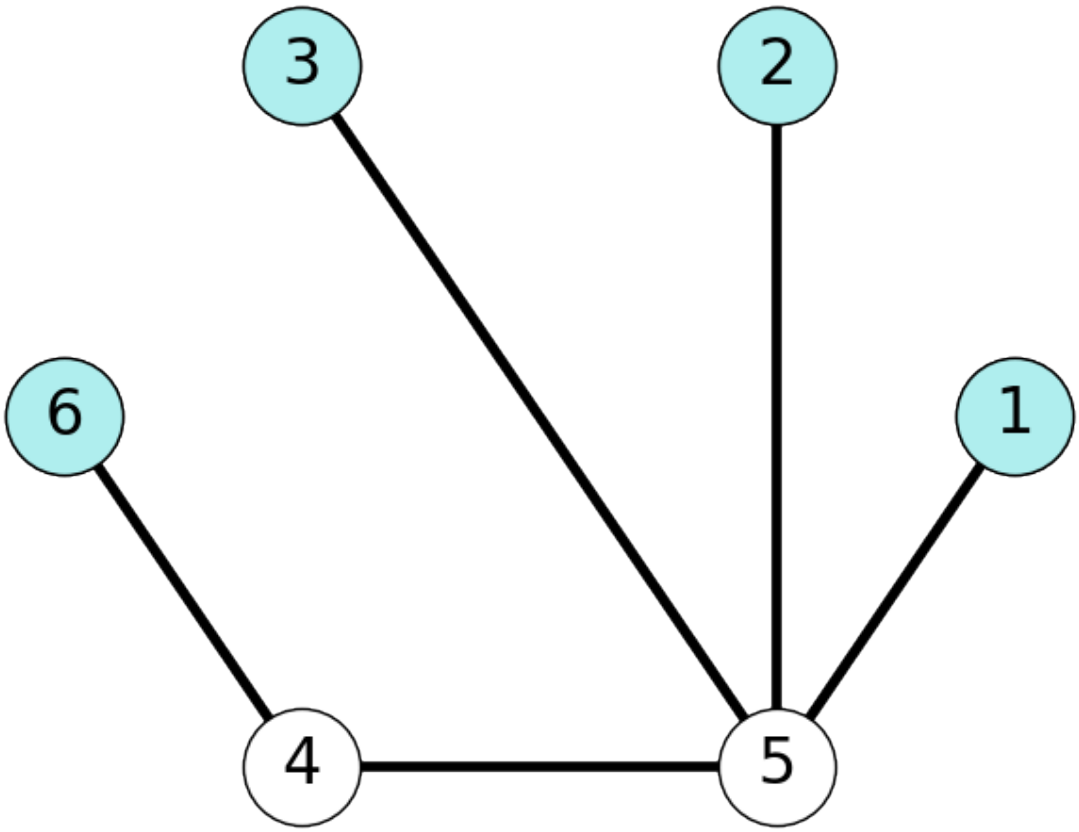


Figure 6.4: A tree with leaves ${1, 2, 3, 6}

6.1.6 The Number of Edges of A Tree

Theorem 6.3. *A tree G_T with n nodes has $m = n - 1$ edges.*

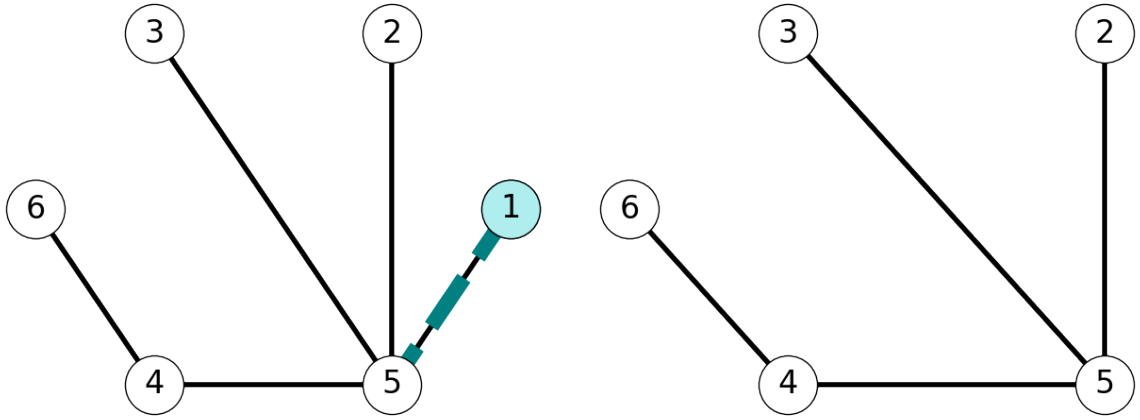
Proof. We'll prove Theorem 6.3 by induction.

As a base case, consider $n = 1$. This tree consists of a single node and $0 = n - 1$ edges.

Now suppose that every tree with k nodes has $n - 1$ edges. Consider tree T with $k + 1$ nodes. We want to show that T has k edges.

To do so, note that T contains at least one leaf by Theorem 6.2. Without loss of generality, let's assume that node 1 is a leaf; if not, we can relabel vertices so that 1 is a leaf. Let's remove leaf 1 and its incident edge (there is only one such edge because 1 has degree-1).

After deleting this node and edge, we have a connected, acyclic graph with k nodes. By the inductive hypothesis, this tree has $k-1$ edges. Since we deleted 1 node and 1 edge, we can conclude that T had k edges. This completes the proof. □

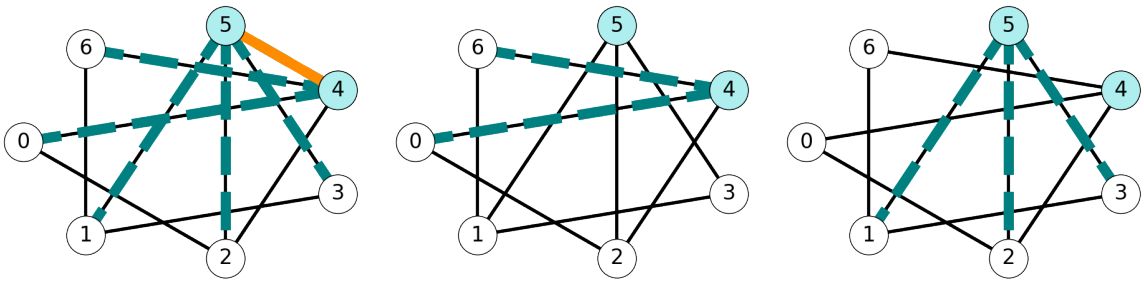


(a) Tree T with leaf 1 (b) After removing $\{1, 5\}$

Figure 6.5: Illustration of the pruning process. After removing node 1 and edge $\{1, 5\}$, we are left with a tree with 5 nodes.

6.1.7 The Swap Property

Lemma 6.1. *Given tree $G_T = (V_T, E_T)$, removing an edge $e \in E_T$ creates two subtrees.*



(a) A tree (b) A subtree (c) Another subtree

Figure 6.6: Subtrees obtained after deleting edge $\{4, 5\}$.

6.1.8 Swapping Edges Within a Cut

Lemma 6.2. *Let S be the vertex set of one of the two subtrees.*

For every edge $f \in \delta(S)$ other than e, the set

$$E'_T := E_T \cup \{f\} \setminus \{e\}$$

is the edge set of a ***tree spanning the same set of nodes****.*

Example 6.3. Consider the graph G with tree T given by Figure 6.7a. The example in Figure 6.6 illustrates that we obtain two subtrees after removing edge $e = 45$.

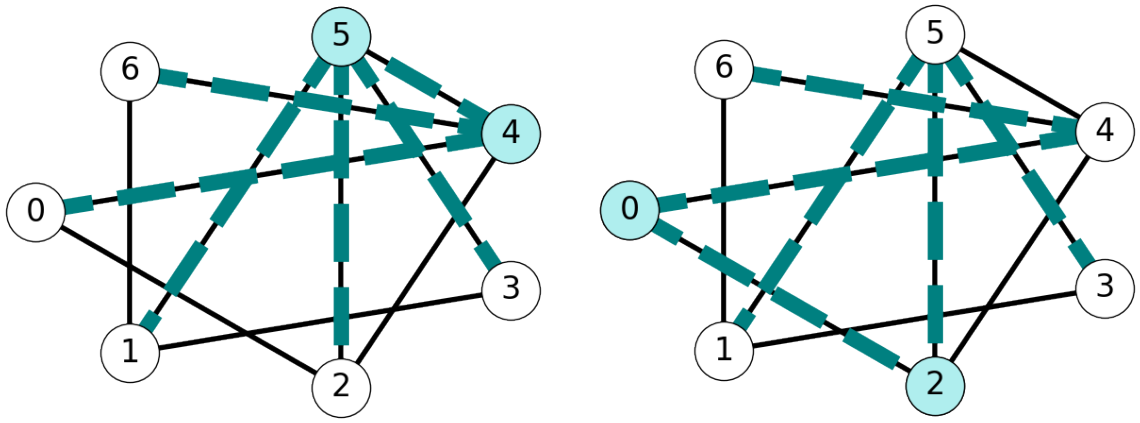
The set $S = \{1, 2, 3, 5\}$ is the vertex set of one of these trees. The cut induced by S is

$$\delta(S) = \{02, 16, 24\}.$$

Lemma 6.2 implies that exchanging 02 with 45 yields a new tree $\tilde{T}$ with edge set

$$E_{\tilde{T}} = E_T \setminus \{4, 5\} \cup \{0, 2\} = \{15, 25, 35, 46, 02\}.$$

See Figure 6.7 for an illustration of this process.



(a) Tree T (b) Tree $\tilde{T}$ after exchange

Figure 6.7: Example of swapping edges 02 and 45 using cuts.

6.2 The Jarnik-Prim-Dijkstra Algorithm

6.2.1 Jarnik's Theorem

Theorem 6.4 (Jarnik's Theorem)**.**

- *Let F be the edge set of a tree strictly contained in an MST.*
- *Let $S \subseteq V$ be the set of nodes it spans.*

For every edge $e \in \delta(S)$:

- *$F \cup \{e\}$ is part of a MST if and only if e has minimum cost in $\delta(S)$.*

Proof. We start by proving the "only if" part. Let's suppose that $F \cup \{e\}$ is part of a MST. We want to show that e has minimum cost in $\delta(S)$ in this case. We'll use a proof by contradiction. Let's assume that there is $f \in \delta(S)$ with $c_f < c_e$.

Let $T = (V, E_T)$ be a minimum spanning tree such that $F \subseteq E_T$ *and* $F \cup \{e\} \subseteq E_T$. By the Swap Property (Lemma 6.1) the subgraph

$$\tilde{T} = (V, E_T \setminus \{e\} \cup \{f\})$$

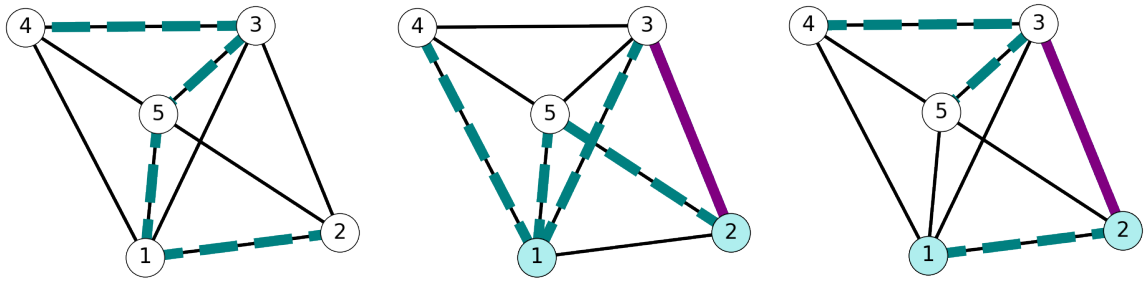
is also a spanning tree. Moreover, the cost of $\tilde{T}$ satisfies

$$c(\tilde{T}) = c(T) - c_e + c_f < c(T)$$

since $c_e > c_f$.

This implies that T is *not* a minimum cost spanning tree; a contradiction. Therefore, we can conclude that if $F \cup \{e\}$ is part of a minimum spanning tree then $c_e \leq c_f$ for all $f \in \delta(S)$.

This is illustrated in Figure 6.8. Consider the spanning tree T given in Figure 6.8a and the edge set $F = \{12\} \subseteq E_T$; here, $S = \{1, 2\}$. Note that $e = 15$ is in the both E_T and $\delta(S)$, while $f = 23$ belongs to $\delta(S)$, but not E_T. The swap property implies that we can obtain a spanning tree $\tilde{T}$ by exchanging the edges e and f. If $c_f < c_e$ then $\tilde{T}$ is a spanning tree with strictly lower cost than T.



(a) T, $S = \{1, 2\}$, and $e = 15$ (b) $\delta(S)$ and $f = 23$ (c) $\tilde{T}$ and $f = 23$

Figure 6.8: Graph with spanning tree T containing $F = \{12\}$ and edge $e = 15$ and edge $f = 23$. If $c_e > c_f$, then $\tilde{T}$ is a spanning tree is smaller cost than T.

We next prove the "if" part. That is, we prove that if e has minimum cost in $\delta(S)$ then $F \cup \{e\}$ is part of a minimum spanning tree. To do so, suppose that $c_e \leq c_f$ for all $f \in \delta(S)$. We want to show that $F \cup \{e\}$ is part of a MST.

We consider two cases. First, suppose that $e \in E_T$, where T is the MST containing f. This immediately implies that e belongs to a MST and we're done.

Next, suppose that $e \notin E_T$. Specifically, let $e = uv$ for nodes $u, v \in V$ such that $e \notin E_T$ and $e \in \delta(S)$; we assume that $u \in S$ and $v \in V \setminus S$.

Let P be the (u, v)-path joining the end points of e belonging to T. Such a path always exists because T is a spanning tree. Since P starts in S at u and ends in $V \setminus S$ at v there is at least one edge f in both P and $\delta(S)$, i.e., there is edge $f \in P \cap \delta(S)$.

By the swap property, the subgraph

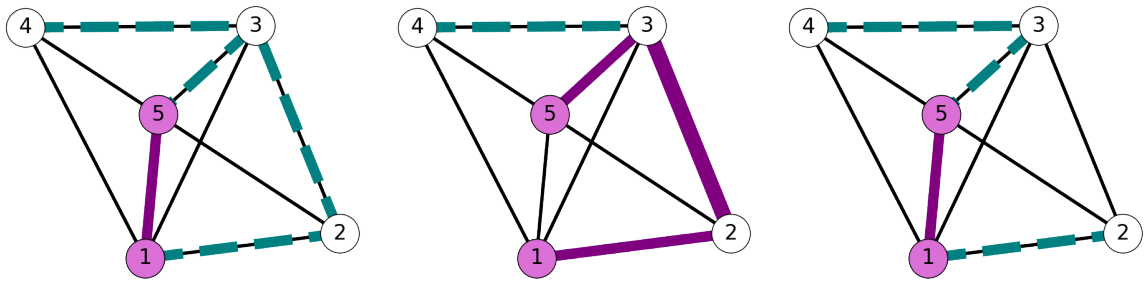
$$T' = (V, E_T \setminus \{f\} \cup \{e\})$$

is a spanning tree. Moreover,

$$c(T) \geq c(T') = c(T) - c_f + c_e \leq c(T)$$

since T is a minimum spanning tree and $c_e \leq c_f$. This is only possible if $c_e = c_f$ and T' is also a minimum spanning tree.

To illustrate this phenomena, consider the edge $e = 15$ in Figure 6.9. This edge is *not* included in the minimum spanning tree T. However, there is a path $P = (1, 2, 3, 5)$ in T containing the edge $f = 23 \in \delta(\{1, 2\})$. Applying the swap property and the assumption the $c_e \leq c_f$ shows that T' is also a minimum cost spanning tree.



(a) Minimum spanning tree T not including $e = 15$.

(b) Path in T from endpoints of e containing $f = 23$.

(c) Minimum spanning tree T' obtained by exchanging e and f.

Figure 6.9: Minimum spanning trees T and T' obtained by exchanging arcs e and f with minimum cost in $\delta(\{1, 2\})$.

□

6.2.2 The Jarnik-Prim-Dijkstra (JPD) Algorithm

Theorem 6.4 suggests the following algorithm for identifying a minimum cost spanning tree in a given graph.

```
Initialize E_T = {} and S = {1}.

While |E_T| < n-1:

    Choose e = {v,w} in delta(S) of minimum cost c_e.

    Update E_T = E_T + {e}

    Update S = S + {w}.
```

6.2.2.1 Complexity of the JPD Algorithm

For each step of the while loop, choosing e in Line 5$ costs $O(m)$ operations to search over the set of edges in $\delta(S)$. The loop is repeated $O(n)$ times, so the total complexity is $O(mn)$ elementary operations. This may be as large as $O(n^3)$ when the graph is very dense of as small as $O(n^2)$ when the graph is very sparse but still connected.

6.2.3 Example: MST for Flight Routing

The following graph $G = (V, E)$ gives available/potential flight routes between 8 cities, with distances in miles. The MST of G gives an airline a means of servicing each city while minimizing travel distance and fuel costs.

Let's apply the Jarnik-Prim-Dijsktra Algorithm to find the minimum cost spanning tree in G.

Step 1

We start with $S = \{1\}$, which induces the cut $\delta(S) = \{14, 18\}$. Since

$$c_{14} = 355 < 695 = c_{18},$$

we add 14 to E_T and add $\{4\}$ to S.

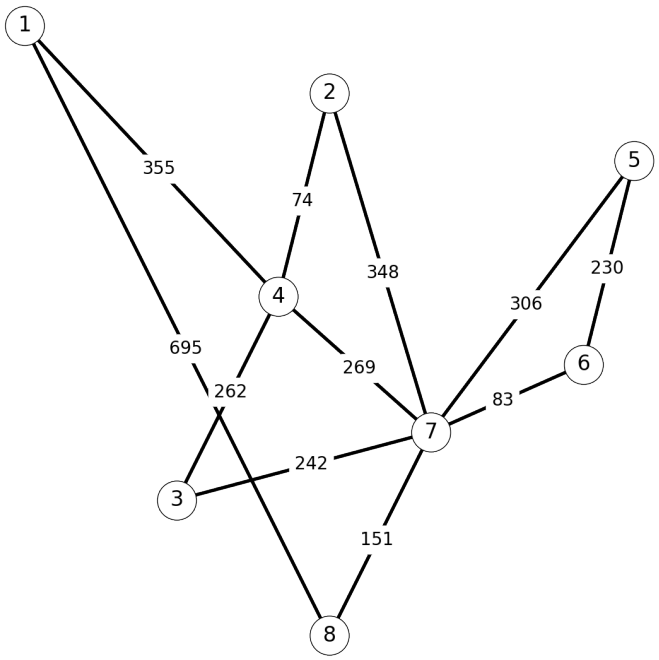


Figure 6.10: Potential flight routes

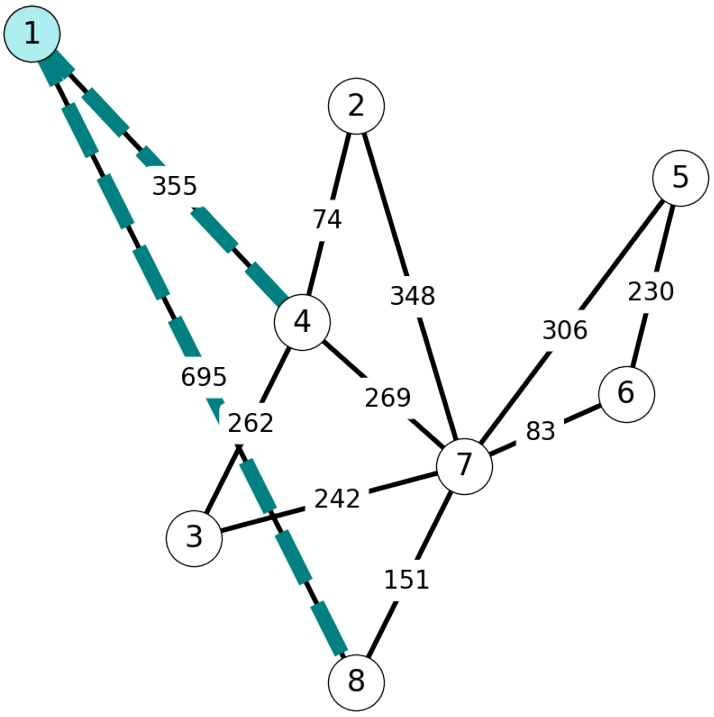


Figure 6.11: Step 1 of the JPD Algorithm

Step 2

Next, $\delta(S) = \{18, 24, 34, 74\}$. We add 2 to S and 24 to E_T because $c_{24} = 74$ is the minimum cost edge in $\delta(S)$.

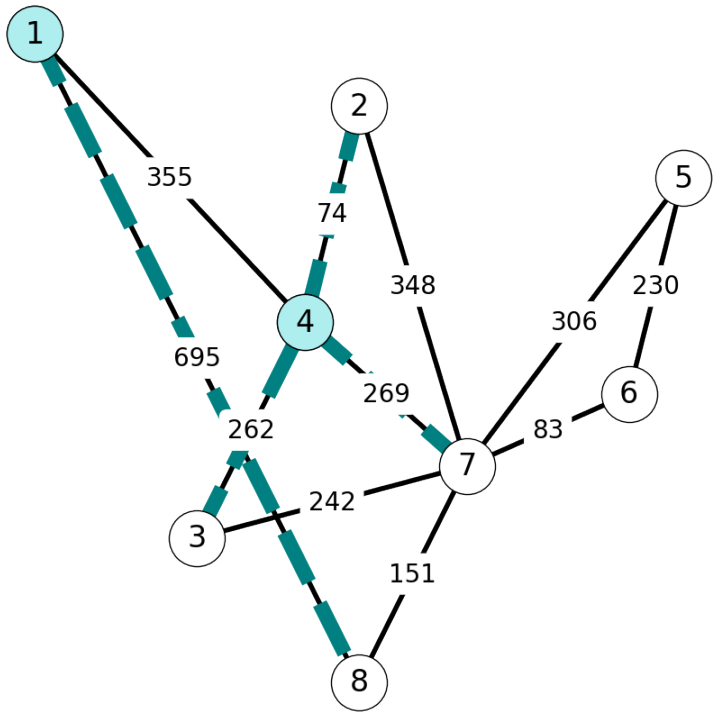


Figure 6.12: Step 2 of the JPD Algorithm

Step 3

Next, $c_{34} = 262$ has minimum cost among edges in $\delta(S) = \{18, 34, 47, 27\}$. We add 34 to E_T and 3 to S.

Step 4

We next add 37 to E_T and 7 to S because $c_{37} = 242$ has minimum cost among edges in $\delta(S) = \{37, 47, 27, 18\}$.

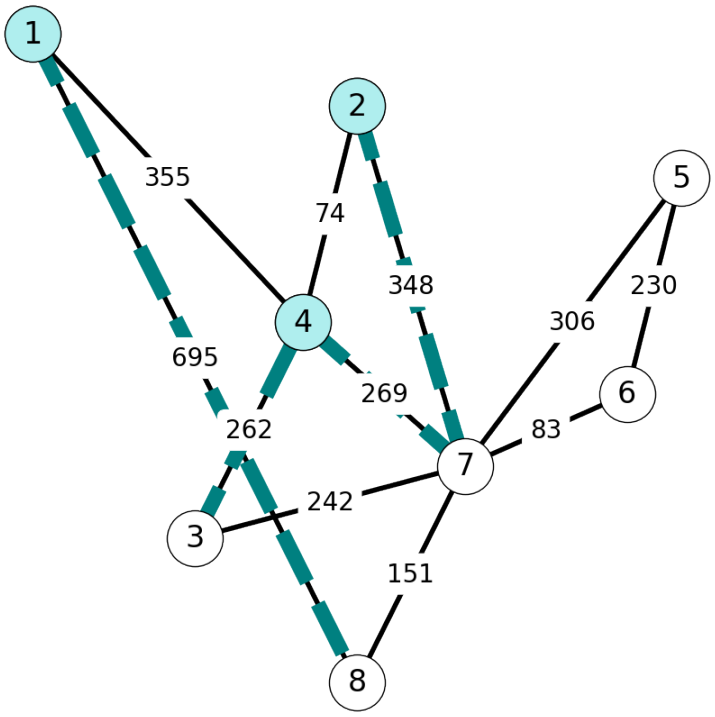


Figure 6.13: Step 3 of the JPD Algorithm

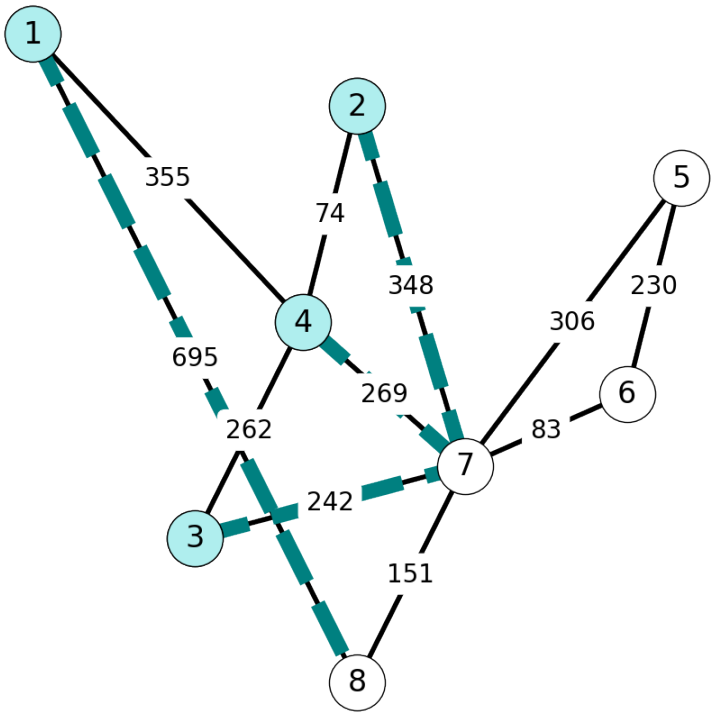


Figure 6.14: Step 4 of the JPD Algorithm

Step 5

We next add 67 to E_T and 6 to S because $c_{67} = 83$ has minimum cost among edges in $\delta(S) = \{18, 75, 76, 78\}$.

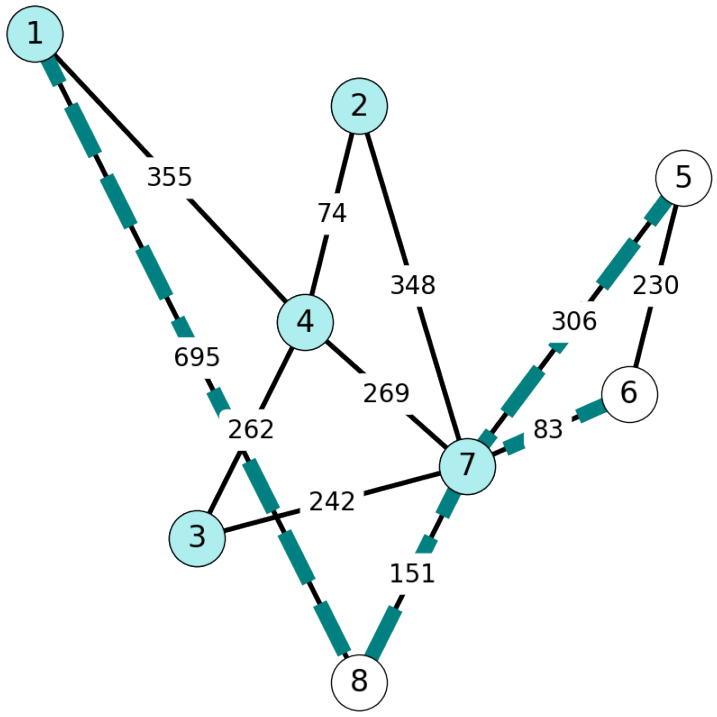


Figure 6.15: Step 5 of the JPD Algorithm

Step 6

Next, $c_{78} = 151$ is minimum cost among edges in $\delta(S) = \{18, 56, 57, 78\}$. Add 8 to S and 78 to E_T.

Step 7

Finally, $\delta(S) = \{56, 57\}$. We have

$$c_{56} = 230 < 306 = c_{57},$$

so we add 56 to E_T, 5 to S.

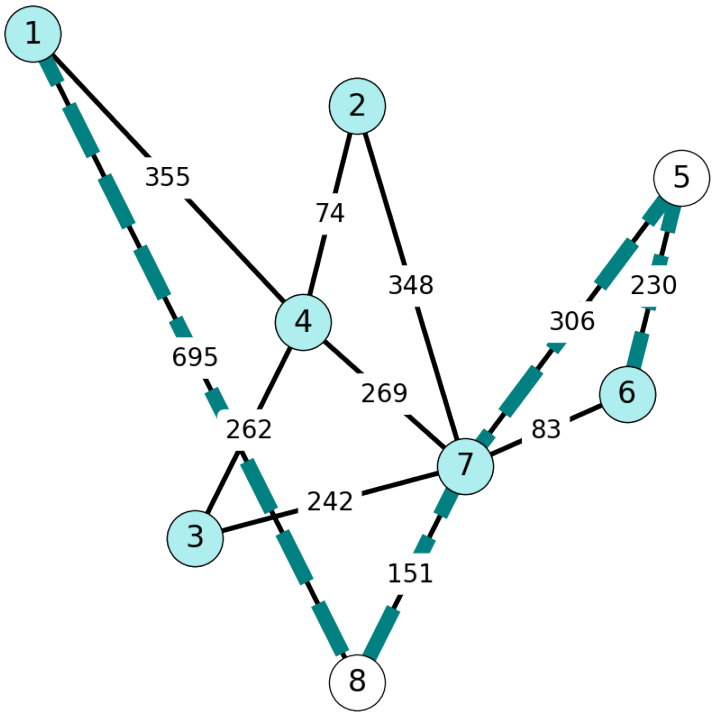


Figure 6.16: Step 6 of the JPD Algorithm

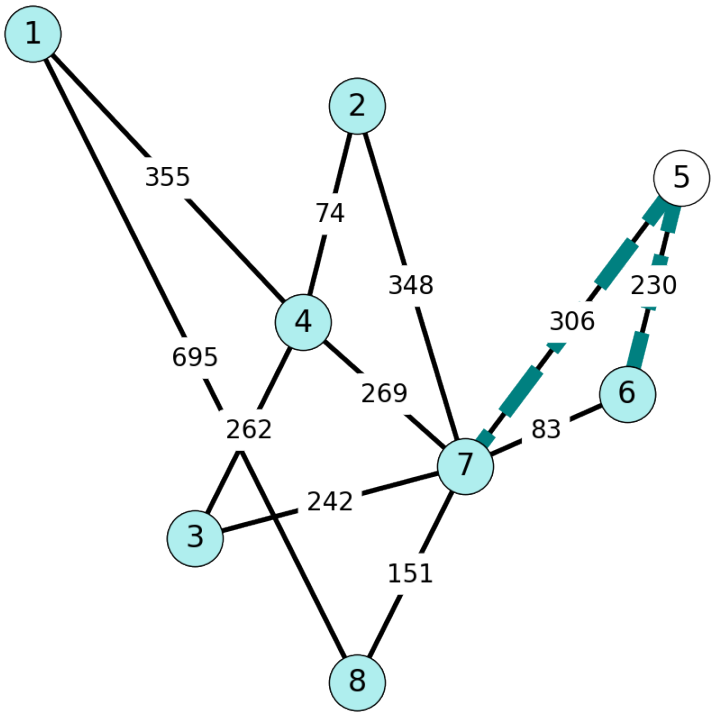


Figure 6.17: Step 7 of the JPD Algorithm

Termination

Note that $S = \{V\}$ and E_T contains $n - 1 = 7$ edges. This implies that we have constructed the minimum spanning tree T.

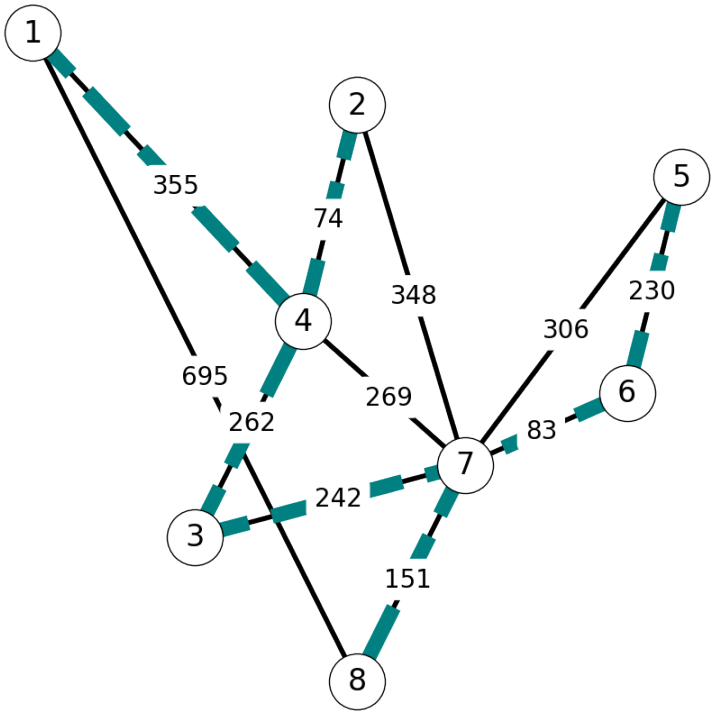


Figure 6.18: Minimum spanning tree identified by the JPD Algorithm

6.3 Improving the JPD Algorithm

6.3.1 Inefficiency of Searching over Edges

Problem: The algorithm scans many edges that will never be selected.

To see why this may be the the case, consider the set $S = \{1, 3, 5\}$ in the graph given in Figure 6.19.

For $S = \{1, 3, 5\}$, we have $\delta(S) = \{21, 23, 25, 41, 45\}$. The JPD Algorithm will add edge 45 with minimum cost to E_T, but will consider *every edge* in G. However, we know that the only candidates to be added to S are Nodes 2 and 4. Since 23 and 45 are the minimum cost

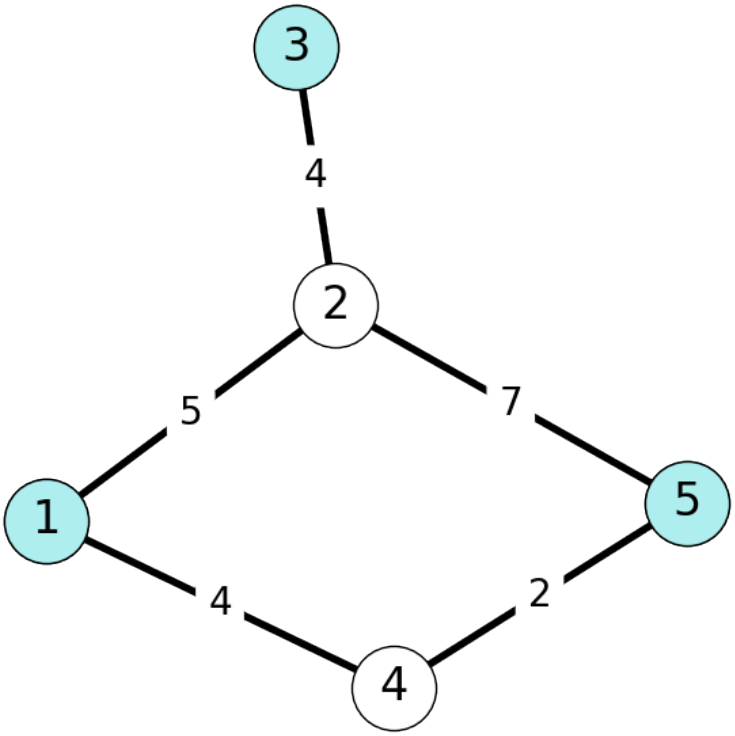


Figure 6.19: Graph G with $S = \{1, 3, 5\}$

edges incident with 2 and 4 respectively, we really only need to consider these two edges as candidates for E_T.

This example implies that if we knew which edge incident with each node has minimum cost, then we can search among nodes, not edges. Since we usually have far fewer nodes than edges, this can lead to substantial improvement in computational cost.

6.3.2 Node Labels

To facilitate this, let's introduce **two labels** for each node $j \in V \setminus S$:

- $C(j) := \min_{i \in S} c_{ij}$ (**minimum cost** of edge connecting S to j);
- $P(j) := \operatorname{argmin}_{i \in S} c_{ij}$ (endpoint in S of this minimum cost edge).

Example 6.4. For the graph given in Figure 6.19, we have

$$C(2) = 4, \quad C(4) = 2$$
$$P(2) = 3, \quad P(3) = 5.$$

If we have C and P, choosing the vertex w to add to **minimum spanning tree** should only require $O(n)$ EOs to compare the values of C for the vertices in $V \setminus S$.

Further, we must update C and P whenever S is updated. To do so, we search the **star** of w, which requires $O(n)$ EOs.

The total complexity is $O(n)$ per iteration and $O(n^2)$ total. This can be much smaller than $O(mn)$!!

6.3.3 Better Algorithm – Pseudocode

This suggests the following algorithm.

```
Let S = {1}, E_T = {}, C(1) = 0.

% Initialize costs.
For j in V - {1}
    Let C(j) = c_{1j} and P(j) = 1
    (assume c_{1j} = inf if 1j not in E).

While |E_T| < n-1

    Find w in V \ S minimizing C(w).
```

```
Let S = S + {w} and E_T = E_T + {P(w), w}.

% Update costs
For j in V \ S such that wj in delta(w)

    If c_{wj} < C(j):
        Update C(j) = c_{wj} and P(j) = w.
```

6.3.4 Another Example

To illustrate the application of the revised algorithm, let's consider the graph G given in Figure 6.20.

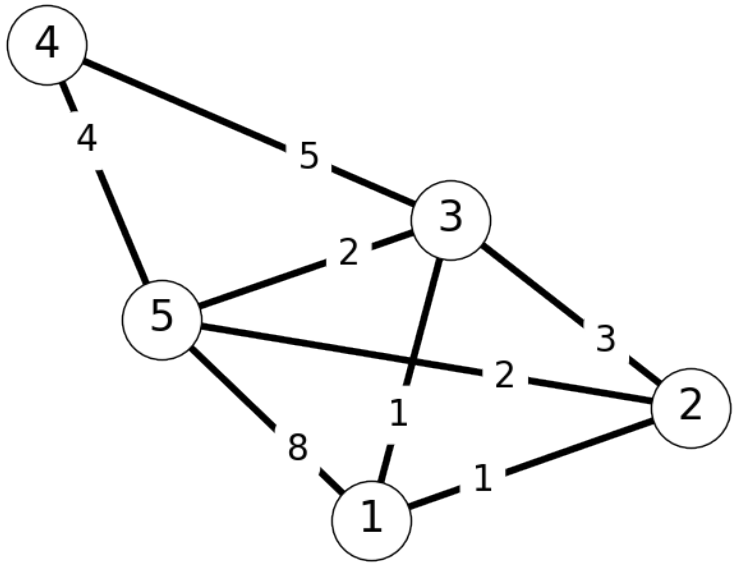


Figure 6.20: Graph G

Step 1

Initially, we have $S = \{1\}$. Note that

$$\begin{aligned} C(2) &= c_{12} = 1 \\ C(3) &= c_{13} = 1 \\ C(4) &= +\infty \\ C(5) &= c_{15} = 8. \end{aligned}$$

We choose $w = 2$. We add 2 to S and 12 to E_T. Note that we could also choose to add 3 to S and 13 to E_T since $c_{12} = c_{13} = 1$.

It remains to update C and P. Note that

$$C(3) = c_{13} = 1 < c_{23} = c_{w3}$$
$$c_{w5} = c_{25} = 2 < C(5).$$

Thus, we set $C(5) = 2$ and $P(5) = 2$ and leave $C(3)$, $P(3)$ unchanged.

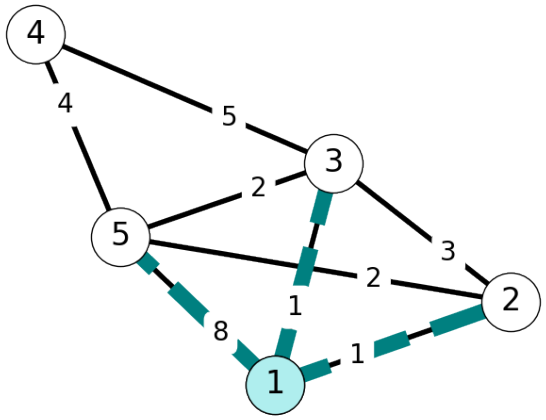


Figure 6.21: Step 1 of the revised JPD algorithm

Table 6.1: Predecessor and value of minimum cost $(1, v)$-path found so far.

v	C(v)	P(v)
2	1	1
3	1	1
5	2	2

Step 2

We have $S = \{1, 2\}$. Nodes 3 and 5 are adjacent to 1 and 2. Since

$$C(3) = 1 < 2 = C(5),$$

we add 3 to S; since $P(3) = 1$, we add 13 to E_T. Finally, we update

$$C(4) = 5, P(4) = 3$$

since 4 is adjacent to 3; $C(5)$ and $P(5)$ are unchanged.

Step 3

Since $C(5) < C(4)$, we add 5 to S and 25 to E_T. Note that

$$c_{45} = 4 < C(4) = 5,$$

we set $C(4) = 4$ and $P(4) = 5$.

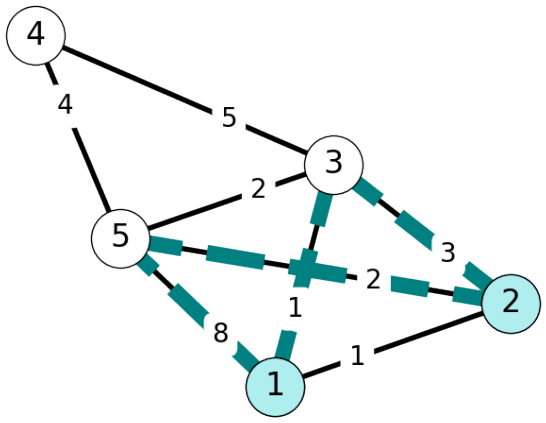


Figure 6.22: Step 2 of the revised JPD algorithm

Table 6.2: Predecessor and value of minimum cost $(1, v)$-path found after two steps of revised JPD algorithm.

v	C(v)	P(v)
2	1	1
3	1	1
4	5	3
5	2	2

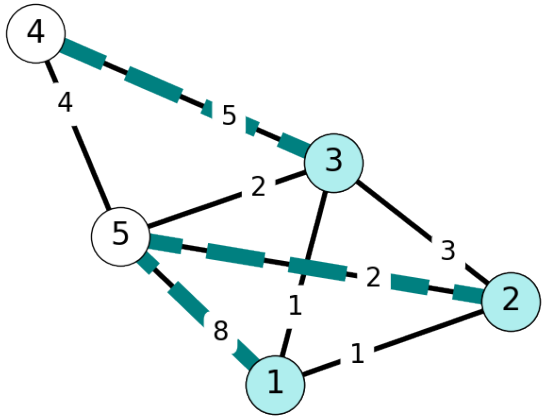


Figure 6.23: Step 3 of the revised JPD algorithm

Table 6.3: Predecessor and value of minimum cost $(1, v)$-path found after three steps of revised JPD algorithm.

v	C(v)	P(v)
2	1	1
3	1	1
4	5	3
5	2	2

Step 4

The only edge left to add E_T is 54.

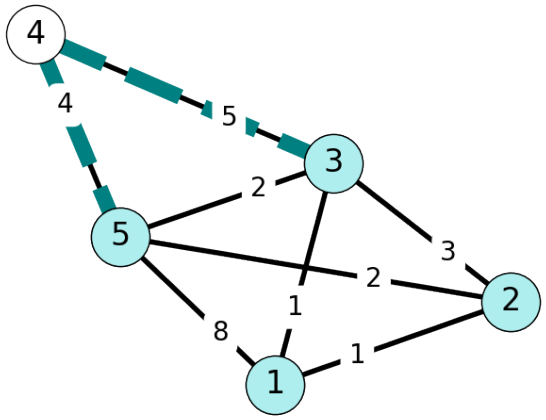


Figure 6.24: Step 4 of the revised JPD algorithm

Table 6.4: Predecessor and value of minimum cost $(1, v)$-path found after four steps of revised JPD algorithm.

v	C(v)	P(v)
2	1	1
3	1	1
4	4	5
5	2	2

This gives the minimum cost spanning tree T found in Figure 6.25.

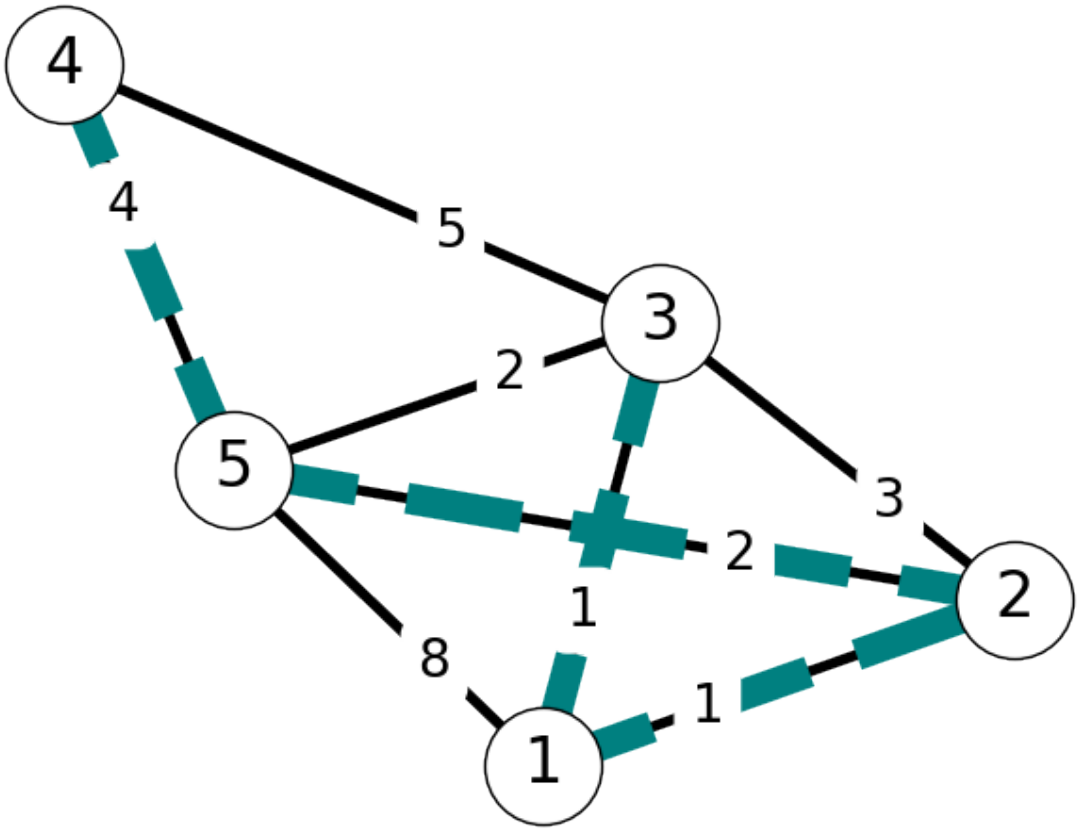


Figure 6.25: Mininum spanning tree in G

7 Network Flows

7.1 Model Formulation

7.1.1 Motivation

We want to figure out the maximum amount of some commodity can be transported from a certain node s to another point t of a network.

This can take the form of any of the following (among many other examples):

- Oil, water, really any fluid in pipeline network;
- Vehicle traffic in a roadway network;
- Information in a communication network;
- Spread of disease in pandemic models.

7.1.2 Standard Constraints

Regardless of physical application/interpretation of the network, we have the following standard collection of constraints.

7.1.2.1 Capacity constraints

Each pipe cannot carry more than its capacity. - Let x_{ij} denote the amount of commodity carried from i to j and let k_{ij} denote the capacity of the pipe from i to j. - Then we must have

$$0 \leq x_{ij} \leq k_{ij}.$$

7.1.2.2 Balance constraints

The commodity cannot enter or leave the network except at source s and sink t.

- Consider node h (not s or t).

- The flow of the commodity *into* h must be equal to the flow *out of* h:

$$\sum_{(i,h)\in\delta^-(h)} x_{ih} = \sum_{(h,j)\in\delta^+(h)} x_{hj}.$$

7.1.3 The Maximum Flow Problem

Given a **directed graph** $G = (V, A)$, **arc capacity function** $k : A \mapsto \mathbf{R}_+$, and nodes $s, t \in V$.

Definition 7.1 (Maximum Flow Problem)**.** Find a function $x : A \mapsto \mathbf{R}_+$ (called a **flow**) satisfying:

- **capacity constraints:**

$$0 \le x_{ij} \le k_{ij}$$

  for all $(i, j) \in A$;

- **balance constraints:**

$$\sum_{(i,h)\in\delta^-(h)} x_{ih} = \sum_{(h,j)\in\delta^+(h)} x_{hj}$$

  for all $h \in V \setminus \{s,t\}$,

- **maximizing the net flow value:**

$$\varphi(\{s\}) = \varphi(x) = \sum_{(s,j)\in\delta^+(s)} x_{sj} - \sum_{(i,s)\in\delta^-(s)} x_{is}.$$

7.1.4 Example

Example 7.1. Figure 7.1 gives an example of a flow x for the given graph G with capacity k.

The flow can be written as

$$\begin{aligned} x_{s1} &= 3, & x_{s2} &= 0 \\ x_{12} &= 2, & x_{1t} &= 1 \\ x_{21} &= 0, & x_{2t} &= 2. \end{aligned}$$

This flow x has value $\varphi(x) = 3$.

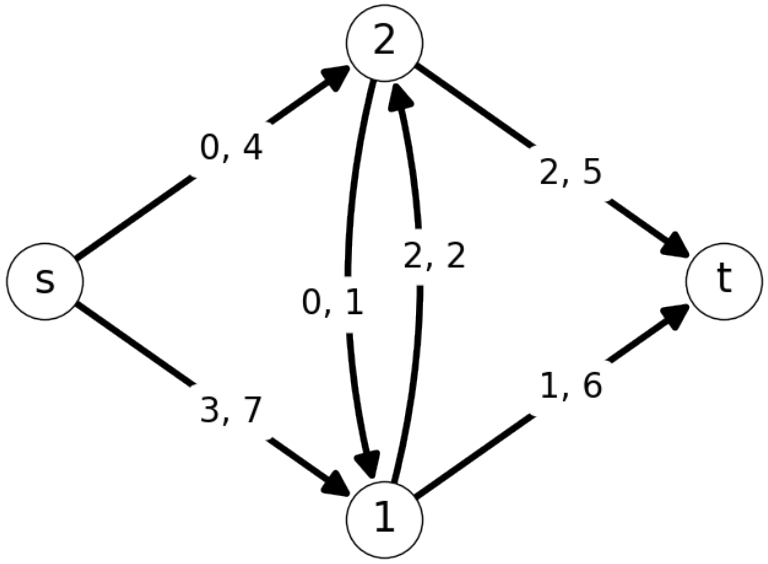


Figure 7.1: Graph G with capacity k and flow x. Arc ij has label x_{ij}, k_{ij}

7.1.5 Formulation as a Linear Program

We can write the **maximum flow problem** as the **linear program** or **linear optimization problem**[1]:

$$\begin{aligned} \max \quad & \varphi \\ \text{s.t.} \quad & \sum_{(h,j)\in\delta^+(h)} x_{hj} - \sum_{(i,h)\in\delta^-(h)} x_{ih} = \begin{cases} +\varphi, & \text{if } h = s, \\ -\varphi, & \text{if } h = t, \\ 0, & \text{otherwise,} \end{cases} \\ & 0 \le x_{ij} \le k_{ij} \text{ for all } (i,j) \in A, \\ & \varphi \ge 0. \end{aligned}$$

7.2 Cuts and Flow

7.2.1 st-cuts

Given a subset of nodes $S \subseteq V$ with $s \in S$ and $t \in V \setminus S$.

Definition 7.2. An **st-cut** is the set of arcs that **enter** or **leave** S:

$$\delta(S) = \delta^+(S) \cup \delta^-(S).$$

[1] https://en.wikipedia.org/wiki/Linear_programming

Definition 7.3. The **(forward) capacity** of the st-cut is

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

7.2.2 Net Flow across a Cut

Definition 7.4. The **net flow** across $\delta(S)$ is

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

7.2.3 Example

Example 7.2. Consider the graph given in Figure 7.2.

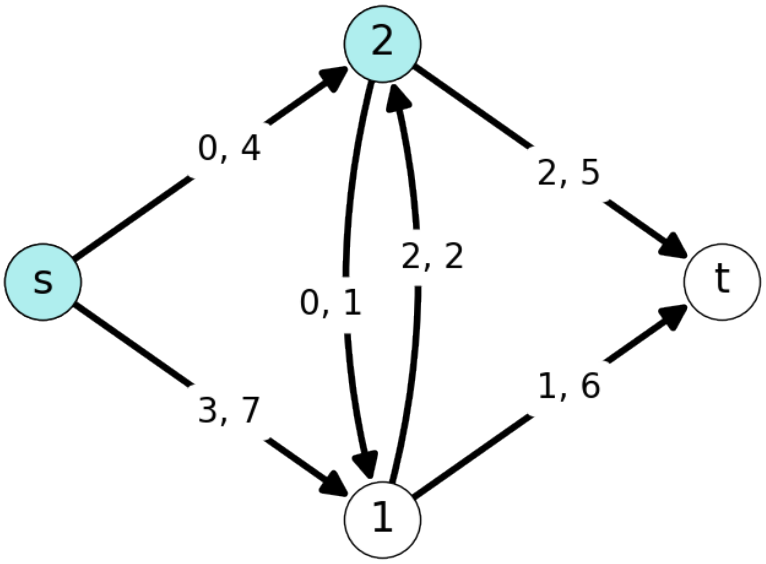


Figure 7.2: Directed graph with $S = \{s, 2\}$

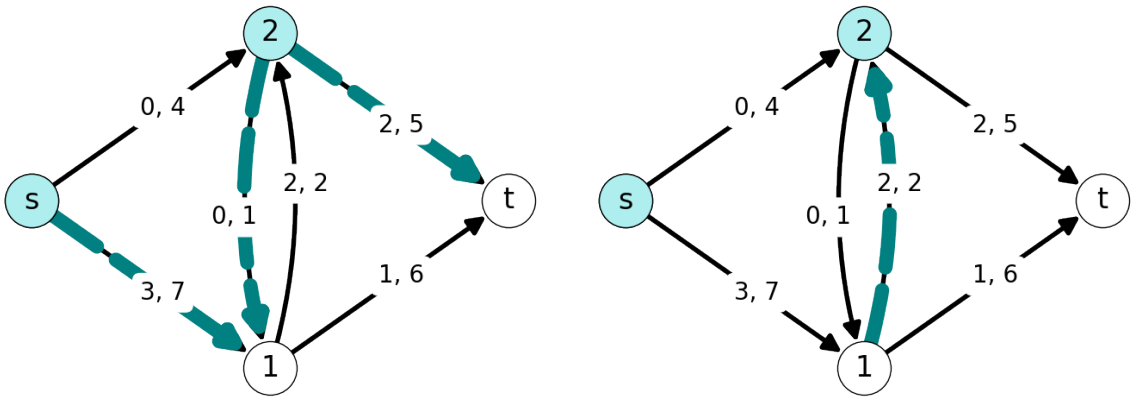
The forward and backward cuts induced by S are given in Figure 7.3.

The st-cut induced by S is

$$\delta(S) = \delta^+(S) \cup \delta^-(S) = \{(s,1),(2,1),(2,t),(1,2)\}.$$

The forward capacity of this cut is

$$k(\delta(S)) = 7 + 1 + 5 = 13.$$



(a) Forward cut $\delta^+(S) = \{(s,1),(2,1),(2,t)\}$ (b) Backward cut $\delta^-(S) = \{(1,2)\}$

Figure 7.3: Induced cuts in graph G given in Figure 7.2.

The value of this cut is

$$\varphi(\delta(S)) = 3 + 0 + 2 - 2 = 3.$$

7.2.4 Conservation of Flow Across Cuts

Lemma 7.1. *Given a flow x with net value φ, we have*

$$\varphi(\delta(S)) = \varphi$$

for every $S \subseteq V$ with $s \in S$, $t \in V \setminus S$.

Consider the balance constraint

$$\sum_{(h,j)\in\delta^+(h)} x_{hj} - \sum_{(i,h)\in\delta^-(h)} x_{ih} = \begin{cases} +\varphi, & \text{if } h = s, \\ -\varphi, & \text{if } h = t, \\ 0, & \text{otherwise.} \end{cases}$$

Let's take the sum of both sides over all $h \in S$; note that $s \in S$ and $t \notin S$. First, the sum of the right-hand side over $h \in S$ is equal to φ.

Next, each arc (i,j) with $i,j \in S$ appears twice in the sum on the left-hand side: - When $h = i$, we add $+x_{ij}$ to the total. - When $h = j$, we add $-x_{ij}$ to the total. These two values cancel when we take the sum over all h. On the other hand, if $i \in S$ and $j \in V \setminus S$, then $(i,j) \in \delta^+(S)$ and appears once in the sum of the left-hand side (when $h = i$). Finally, if $i \in V \setminus S$, $j \in S$ then $(i,j) \in \delta^-(S)$ and this edge contributes $-x_{ij}$ to the sum in the left-hand side. Putting everything together, we have

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

This completes the proof.

7.3 Duality

7.3.1 Weak Duality

The following theorem gives an upper bound on the value of *any flow* in terms of the capacity of *an arbitrary cut*. Specifically, the value of any flow is at most the capacity of any st-cut.

Theorem 7.1. *For each $S \subseteq V$ with $s \in S$ and $t \in V \setminus S$, we have*

$$\varphi(x) \leq k(\delta(S))$$

for any flow x.

Let x be a feasible flow and consider st-cut S. Then

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

since x_{ij} k_{ij}$ for any arc (i,j). On the other hand,

$$-\sum_{(i,j)\in\delta^-(S)} x_{ij} le 0,$$

since $x_{ij} \geq 0$ for any arc (i,j). Applying conservation of flow, Lemma 7.1, we have

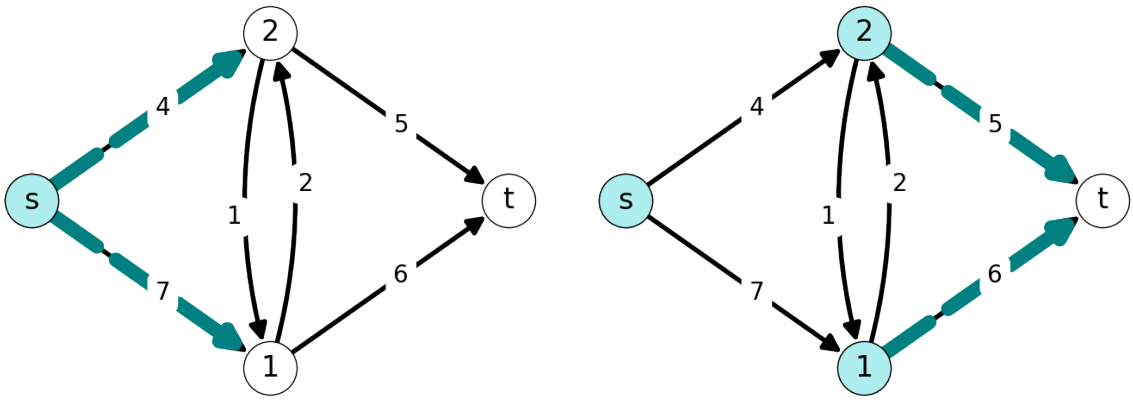
$$\varphi = \sum_{(i,j)\in\delta^+(S)} x_{ij} - \sum_{(i,j)\in\delta^-(S)} x_{ij} \leq \sum_{(i,j)\in\delta^+(S)} k_{ij} = k(\delta(S)).$$

7.3.2 The Minimum Cut Problem

Theorem 7.1 holds for every st-cut. In particular, it is true for the cut with minimum capacity. Thus, for any feasible flow x, we have

$$\varphi(x) \leq \Big\{ k(\delta(S)) : \delta(S) \text{if a } st\text{-cut} \Big\}.$$

::: {#def-7-min-cut-problem}



(a) Minimum capacity cut $\delta(\{s\})$ (b) Minimum capacity cut $\delta(\{s\})$

Figure 7.4: Two minimum capacity cuts in Figure 7.1.

7.3.2.1 Minimum Cut Problem

Given a directed graph $G = (V, A)$ with arc capacity function $k : A \mapsto \mathbf{R}_+$ and two nodes $s, t \in V$.

Example 7.3. Consider the graph given in Figure 7.1. Find an st-cut of **minimum capacity**.

7.3.3 Strong Duality: Max-Flow Min-Cut Theorem

The relationship

$$\begin{aligned} & \varphi \leq k(\delta(S)) \text{ for all } st\text{-cuts} \\ \Leftrightarrow \quad & \varphi \leq \min_S \Big\{ k(\delta(S)) : \delta(S) \text{ is a } st\text{-cut} \Big\} \end{aligned}$$

holds for **all net flow values** φ.

This implies that for any flow value φ we have

$$\varphi \leq \max \Big\{ \varphi(x) : x \text{ is a flow} \Big\} \leq \Big\{ k(\delta(S)) : \delta(S) \text{if a } st\text{-cut} \Big\}.$$

The maximum flow value and minimum cut capacity are actually guaranteed to agree for every flow network. We have the following two results.

Lemma 7.2. *Consider flow x of net value φ and st-cut $\delta(S)$ of capacity $k(\delta(S))$.*

*If $\varphi = k(\delta(S))$, then x is a **maximum flow** and $\delta(S)$ is a **minimum cut**.*

Theorem 7.2. *The value of a maximum flow is **always** equal to the capacity of a **minimum** st-cut.*

We'll construct an **algorithmic proof:** we'll present an algorithm that reveals and constructs the flow and st-cut while solving the problem.

7.4 The Ford-Fulkerson Algorithm – Idea

7.4.1 Improving Flows along Forward and Backward Arcs

Idea: Starting from given flow x, we will try to find an st-path along which φ can be increased.

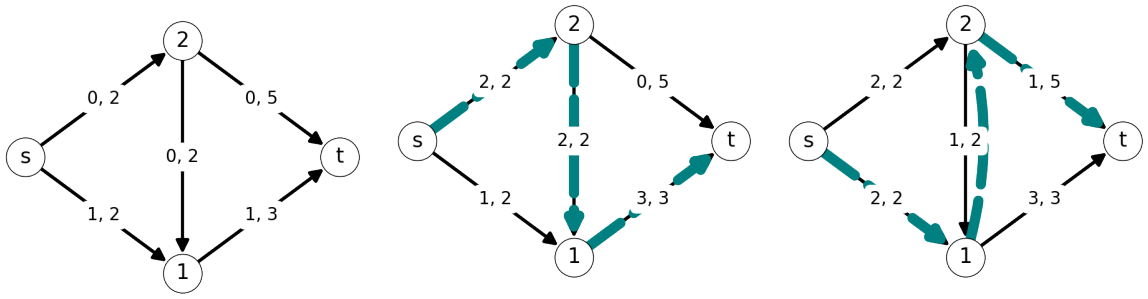
We can increase flow using

- the **original arcs** in the graph (**forward arcs**), ***and***
- arcs whose directions have been flipped (**backward arcs**).

Example 7.4. Consider the graph given in Figure 7.5a with flow x indicated.

We can increase the total flow by increasing by two units of flow along the *forward arcs* comprising the path $(s2, 21, 1t)$. This gives a new flow with value $\varphi = 3$.

Subsequently, we can move an additional 1 unit of flow by redirecting 1 unit of flow along arc 21 to flow along $2t$ and pushing an additional unit of flow along $s1$; see Figure 7.5b. This is equivalent to pushing 1 unit of flow along the path $(s, 1, 2, t)$, where $s1, 2t$ are forward arcs and 12 is a backward arc; see Figure 7.5c.



(a) Initial flow x

(b) Increased flow along path $(s, 2, 1, t)$

(c) Increased flow along path $(s, 1, 2, t)$

Figure 7.5: Increasing flow using forward and backward arcs.

7.4.2 Saturated and Empty Arcs

Definition 7.5. An arc (i, j) is **saturated** if $x_{ij} = k_{ij}$.

Definition 7.6. An arc (i, j) is **empty** if $x_{ij} = 0$.

Observation: Our algorithm for finding the maximum flow and minimum capacity cut hinges on the following:

- If (i, j) is **not saturated** $(x_{ij} < k_{ij})$ then x_{ij} can be **increased**.
- If (i, j) is **not empty** $(x_{ij} > 0)$ then x_{ij} can be **decreased**.

Example 7.5. Consider the graph G with flow x given in @#fig-7-saturated-ex-G.

- Arcs $s2$, 21, and $1t$ are saturated;
- Arc $2t$ is empty;
- Arc $s1$ is not empty or saturated.

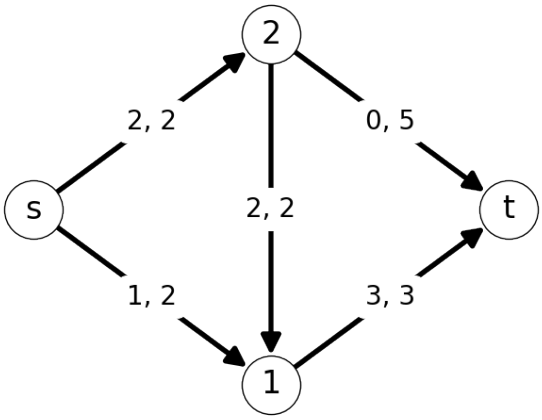
7.4.3 Augmenting Paths

Definition 7.7 (Augmenting Paths)**.** We call a st-path using a combination of forward and backward arcs an **augmenting path** if:

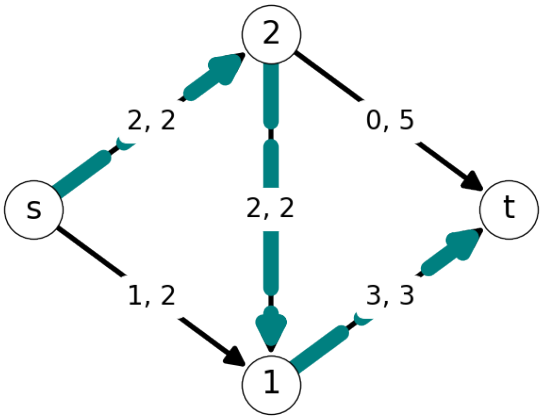
- All **forward arcs** are **not saturated** $(x_{ij} < k_{ij})$; and
- All **backward arcs** are **not empty** $(x_{ij} > 0)$.

We can improve net flow value by **increasing flow on forward arcs** and **decreasing flow on backward arcs**.

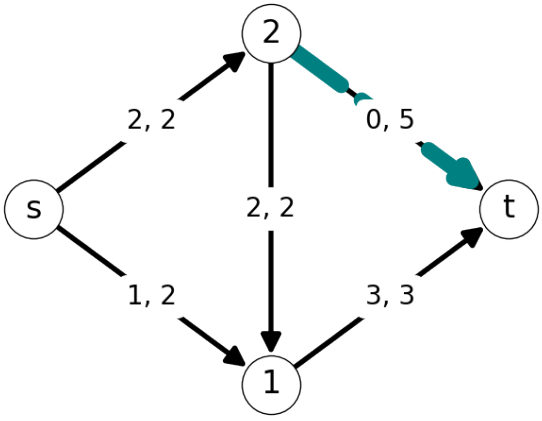
Example 7.6. The graph G and flow x indicated in Figure 7.7 has augmenting path $P = (s, 1, 2, t)$.



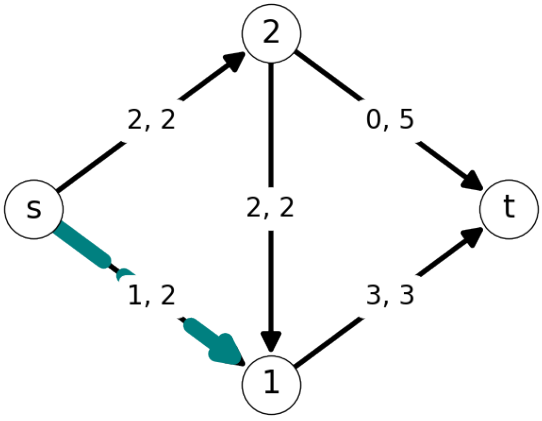
(a) Graph G with flow x



(b) Saturated arcs $s2$, 21, and \$1t



(c) Empty arc $2t$



(d) Arc $s1$ is neither.

Figure 7.6: Saturated and empty arcs for flow x in the given graph.

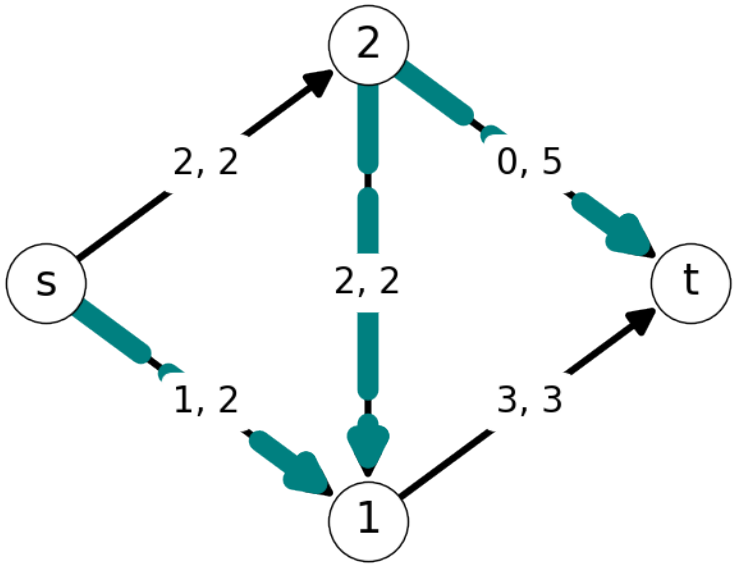


Figure 7.7: An augmenting path $P = (s, 1, 2, t)$

7.4.4 The Auxiliary Network

At each step of our algorithm, we will try to find an augmenting path. We will construct an additional flow network to facilitate finding such an augmenting path.

We can construct an **auxiliary network** $G' = (V, A')$ with **auxiliary capacity function** $k' : A \mapsto \mathbf{R}_+$ that accounts for all possible flow variations with respect to current flow x.

For all $(i, j) \in A$:

- If $x_{ij} < k_{ij}$ (not saturated), we add **forward arc** (i, j) to A' with $k'_{ij} = k_{ij} - x_{ij}$.
  - This arc models the **possibility of increasing the flow** on (i, j).
- If $x_{ij} > 0$ (not empty), we add **backward arc** (j, i) to A' with $k'_{ji} = x_{ij}$.
  - This models the **possibility of decreasing the flow** on (i, j).
- If $0 < x_{ij} < k_{ij}$ we add both a forward arc (i, j) and a backward arc (j, i) to A'.

7.4.5 The Augmentation Step

Start with flow x and corresponding auxiliary network G' with auxiliary capacities k'.

1. Identify st-path P in G'.

2. Let $\Delta = \min_{(i,j)\in P} k'_{ij}$.
3. Add Δ to all forward arcs.
4. Subtract Δ from all backward arcs.

7.5 Example

Example 7.7. Let's find the **maximum flow** for $s = 1$ and $t = 7$ for the graph Figure 7.12a. We start from $x = 0$ with net flow $\varphi = 0$. The initial auxiliary graph G' is equal to G.

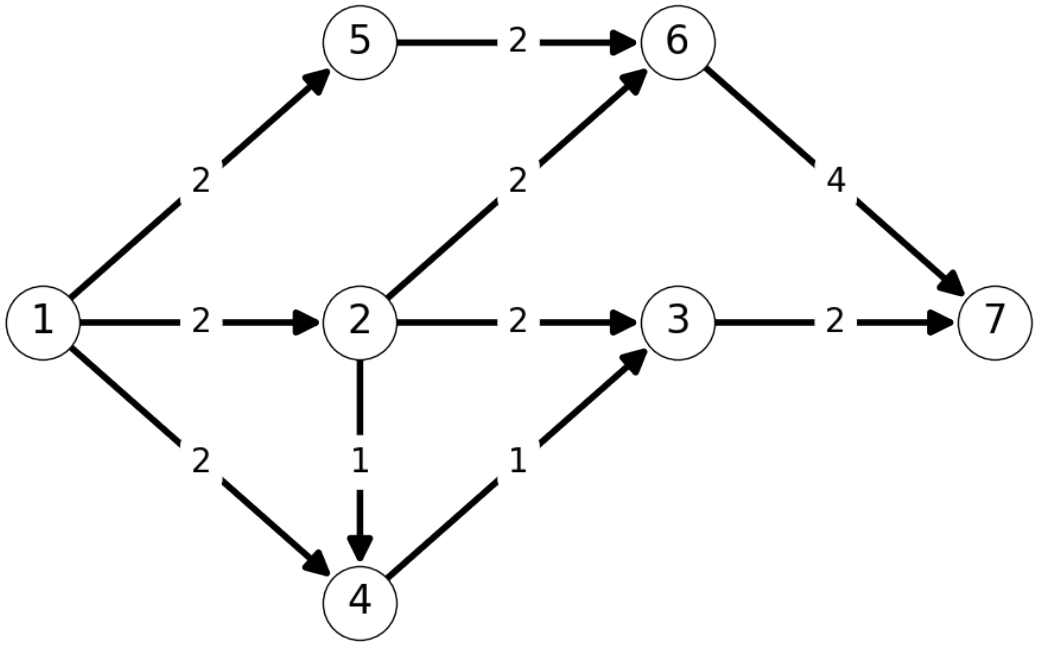
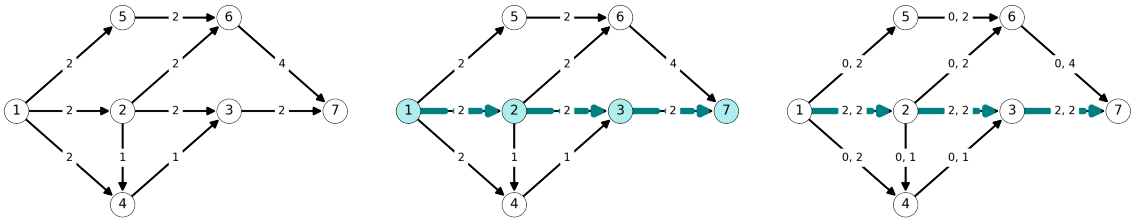


Figure 7.8: Graph G

Step 1

We have several augmenting paths to choose from. Consider $P = (1, 2, 3, 7)$ in G'. The minimum edge capacity in G' along this augmenting path is $\Delta = 2$. All arcs in this path are forward arcs, so we add Δ to x_{ij} for arcs $12, 23, 37$, and leave all flow on other edges equal to 0.

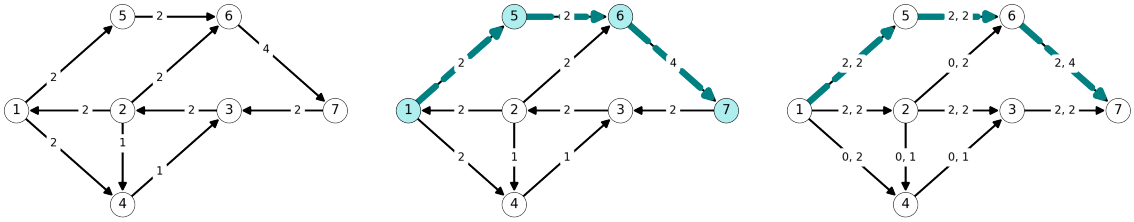


(a) Auxiliary Graph G' (b) Augmenting Path (c) Updated Flow

Figure 7.9: Details of Step 1.

Step 2

Arcs 12, 23, and 37 are now *saturated*. We can update the auxiliary graph by reversing each of these edges and setting the capacity equal to the flow value across each arc. This reveals the augmenting path $P = (1, 5, 6, 7)$ (among several possible augmenting paths). This path has minimum capacity arc with capacity $\Delta = 2$ and every arc is a *forward arc* on this path. Thus, we can increase flow across each of these arcs by $\Delta = 2$.



(a) Auxiliary Graph G' (b) Augmenting Path (c) Updated Flow

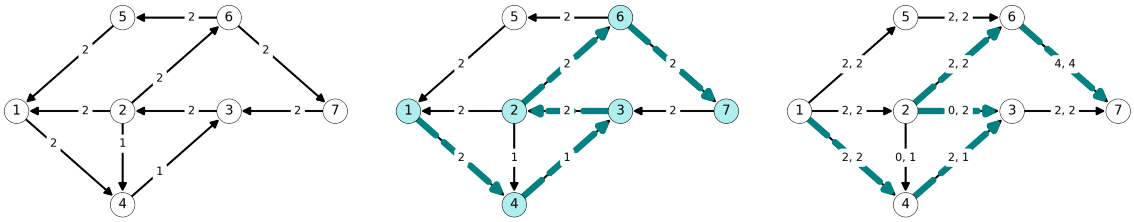
Figure 7.10: Details of Step 2.

Step 3

Next, we have augmenting path $P = (1, 4, 3, 2, 6, 7)$. Note that this path uses the *nonempty backward arc* 32. The minimum capacity arc in the auxiliary graph along this path is 43, with capacity $k'_{43} = 1 = \Delta$. We can update our flow by increasing the flow by $\Delta = 1$ along all forward arcs in this path and decreasing flow across backward arcs by $\Delta = 1$

Step 4 and Termination

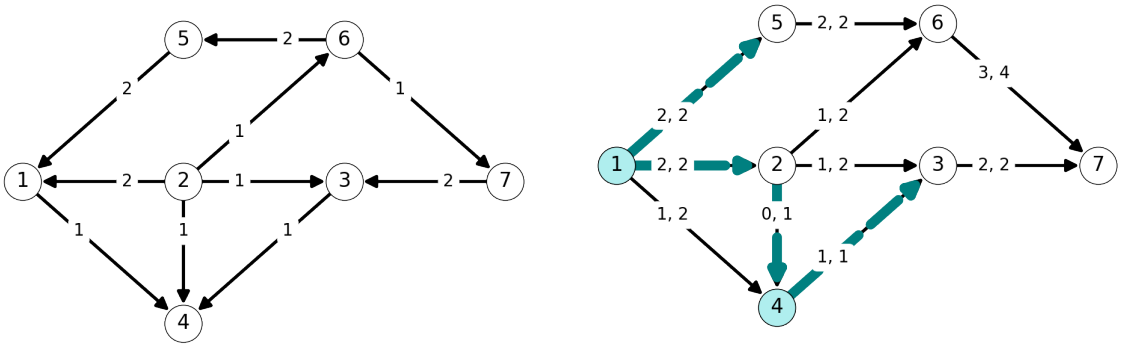
After updating the flow and auxiliary graph G', we can note that t is *not reachable* from s in G'. Indeed, the reachable set from s in G' is $S = \{s, 4\}$. Note that the cut $\delta(S)$ defined by S has capacity $k = (\delta(S)) = 5$. This is equal to the total value of the flow x ($\varphi = 5$). This



(a) Auxiliary Graph G' (b) Augmenting Path (c) Updated Flow

Figure 7.11: Details of Step 3.

implies that x is a maximimum value flow and $\delta(S)$ is a minimum capacity cut; we can stop applying the algorithm.



(a) Auxiliary Graph G'

(b) Reachable nodes S and minimum capacity cut $\delta(S)$

Figure 7.12: Details of Step 4

7.6 Termination and Correctness

7.6.1 Stopping Condition

Let S^* be the set of nodes **reachable** from $s \in G'$ when the algorithm stops.

We can conclude that $\delta^+_{G'}(S^*) = \emptyset$.

By the construction of G', this implies that:

- Every arc $(i, j) \in \delta^+_G(S^*)$ is **saturated!**
- Every arc $(i, j) \in \delta^-_G(S^*)$ is **empty!**

7.6.2 Correctness of the FF-Algorithm

When the algorithm terminates, **all arcs in $\delta_G^+(S^*)$ are saturated** and **all arcs in $\delta_G^-(S^*)$ are empty**:

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

However, we know that

$$\varphi \leq \text{max flow value} \leq \text{min cut value} \leq k(\delta(S^*)).$$

This shows that when the algorithm terminates, we have found a **flow equal in value** to the **capacity of an s,t cut**!.

This implies that:

- The **flow is optimal for the maximum flow problem**.
- The **cut is optimal for the minimum cut problem**

This establishes that the Ford-Fulkerson algorithm **always finds an optimal solution!**

This is an **algorithmic proof** of the **max-flow min-cut theorem**.

7.7 The Ford-Fulkerson Algorithm – Pseudocode

```
Initialize x = 0, phi = 0.

While x is not a max flow:
    Construct auxiliary graph G' = (V, A') with capacities k_{ij}'.

    Find an st-path P in G'.

    If P does not exist: % Terminate
        STOP! x is a max flow.
    Else % Update flow.
        Delta =  min {k_{ij}': (i,j) \in P }
        phi = phi + Delta.

        For (i,j) in P:
            If (i,j) is a forward arcL
                x_{ij} = x_{ij} + \Delta
            Else
                x_{ij} = x_{ij} - \Delta
```

7.8 Complexity of the Ford-Fulkerson Algorithm

7.8.1 A Rough Bound on Elementary Operations

We can count the number of elementary operations used by each iteration of the Ford-Fulkerson Algorithm:

- Constructing the auxiliary graph G' in Line 4 costs $O(m+n)$ elementary operations. Indeed, we need to construct a copy of G, change direction of arcs (if necessary), and update capacities. This requires a finite number of operations for each arc and node.
- We can find an st-path, if one exists, using a modification of the *graph reachability algorithm*. This requires $O(m+n)$ elementary operations.
- The remaining steps in the loop require $O(|P|)$ elementary operations for augmenting path P. Assuming that P is a simple path, we have $|P| \leq n-1$ edges.

Therefore, each iteration of the Ford-Fulkerson Algorithm requires $O(m)$ elementary operations (assuming $n = O(m)$). If the algorithm uses q iterations, then the total number of operations is

$$O(qm).$$

7.9 Worst-Case Analysis of the Number of Iterations

To bound the number of operations needed by the Ford-Fulkerson Algorith, we need to determine how many iterations are needed by the Ford-Fulkerson Algorithm in worst-case?

The following lemma will help determine an upper bound on the number of iterations needed.

Lemma 7.3. *If all capacities k_{ij} are* ***integer*** *then k'_{ij}, Δ, and x are* ***integer at each iteration****.*

Since Δ is integer, we could have **unit increment** $\Delta = 1$ each iteration (in worst case). More precisely, there are graphs where

$$\Delta = \min_{ij} k'_{ij} = 1$$

every iteration of the Ford-Fulkerson Algorithm. In this case, if we start with flow value $\varphi = 0$ then we need $q = \varphi^*$ iterations, where φ^* is the value of the maximum flow.

On the other hand, if Δ is integral and positive every iteration, then we increment φ by *at least* Δ each iteration. Therefore, $q \leq \varphi^*$ if the capacities are integral.

It follows that the worst-case complexity of the Ford-Fulkerson Algorithm is $O(\varphi^* m)$.

7.9.1 A Pathologically Bad Example

Example 7.8. Let's apply the Ford-Fulkerson Algorithm with the graph G given in Figure 7.13 and initial flow $x = 0$.

We increase flow by $\Delta = 1$ along the augmenting path $P = (s, 2, 1, t)$ (Figure 7.17a) This yields the new flow given in Figure 7.14b.

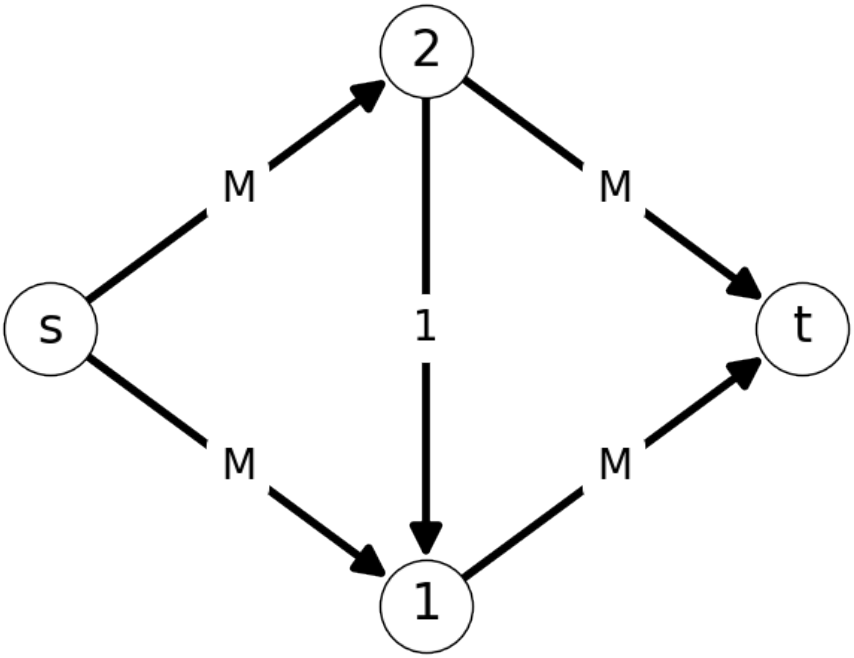


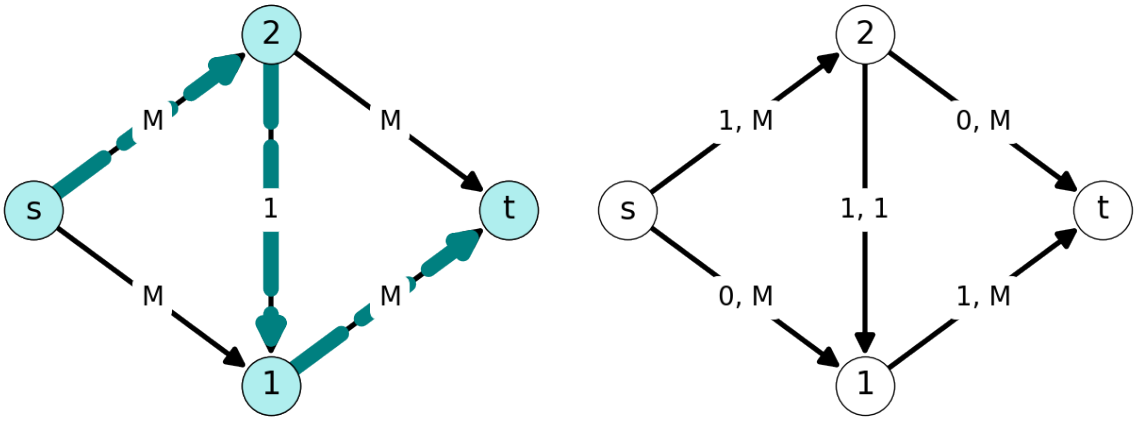
Figure 7.13: Graph G

After updating the flow, we need to update the auxiliary graph. This reveals the augmenting path $P = (s, 1, 2, t)$. We can increase the flow along this path by $\Delta = 1$. See Figure 7.15.

We can repeat this process. In the next step, we have augmenting path $P = (s, 2, 1, t)$; this is the same augmented path used in Step 1. We can increase flow by $\Delta = 1$. In the fourth step, we increase flow by 1 along the augmenting path $P = (s, 1, 2, t)$.

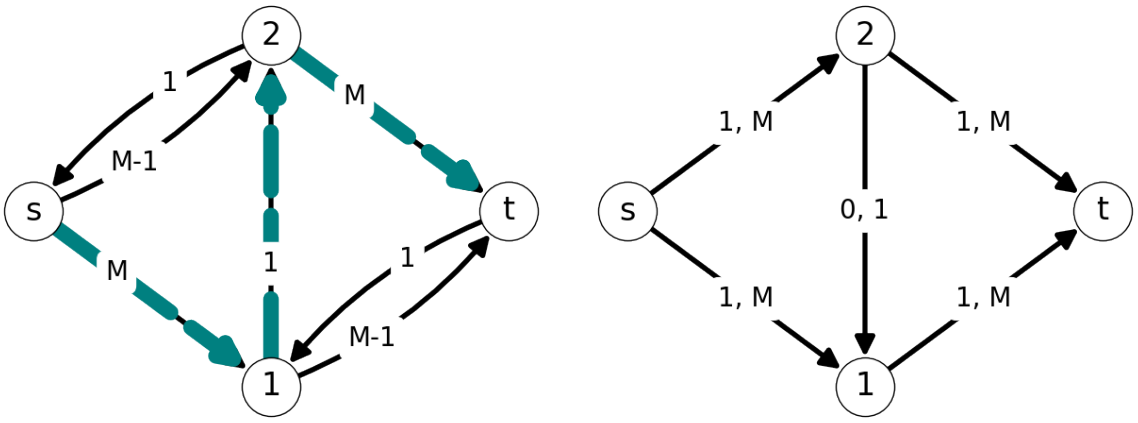
We can repeat this process a total of M times ($2M$ iterations). This gives optimal flow given in

Applying the Ford-Fulkerson Algorithm with this graph requires exactly $q = 2M = \varphi^*$ iterations.



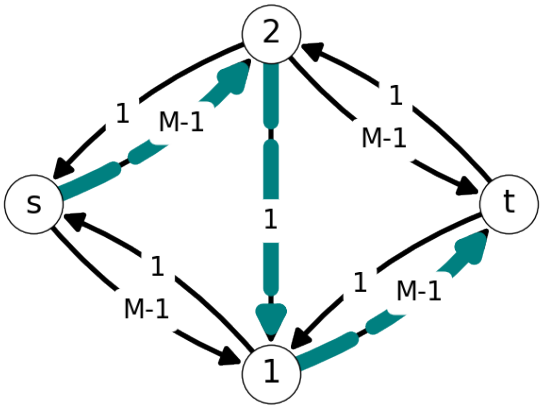
(a) Augmenting path in auxiliary graph (b) Updated flow

Figure 7.14: Details for first step of the Ford-Fulkerson Algorithm.

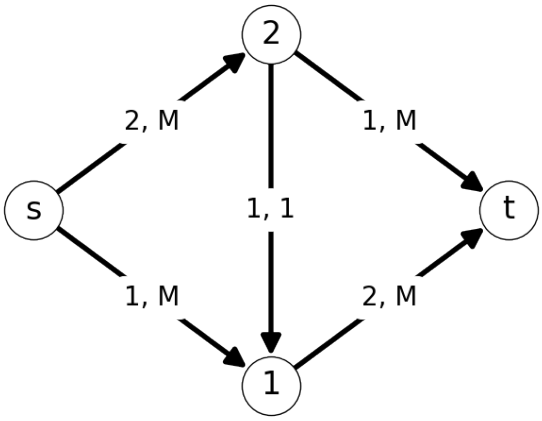


(a) Augmenting path in auxiliary graph (b) Updated flow

Figure 7.15: Details for second step of the Ford-Fulkerson Algorithm.

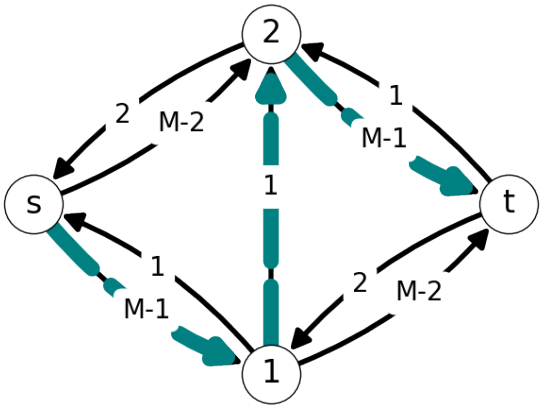


(a) Augmenting path in auxiliary graph

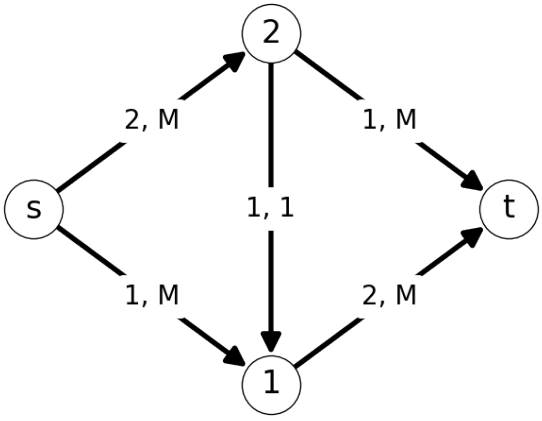


(b) Updated flow

Figure 7.16: Details for third step of the Ford-Fulkerson Algorithm.



(a) Augmenting path in auxiliary graph



(b) Updated flow

Figure 7.17: Details for fourth step of the Ford-Fulkerson Algorithm.

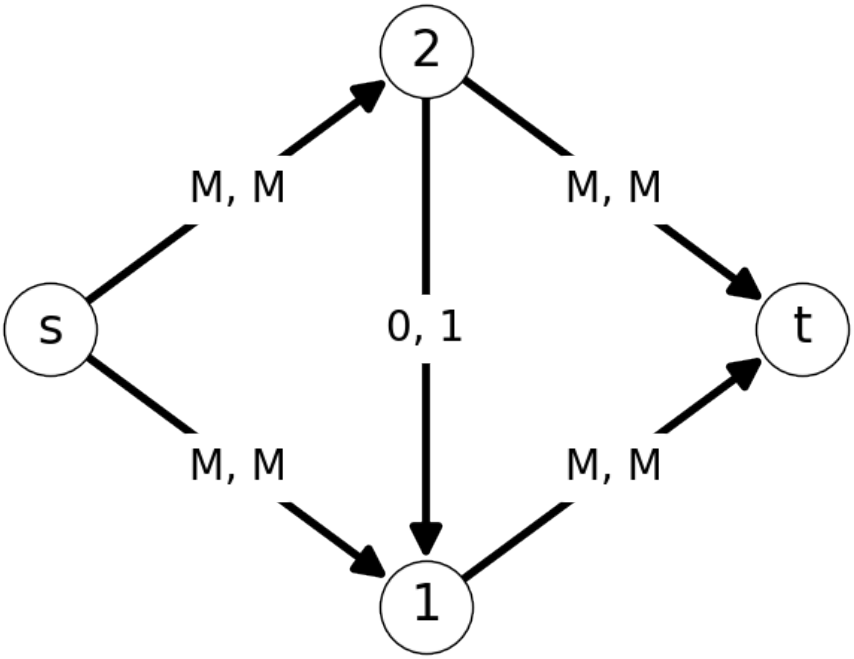


Figure 7.18: Optimal flow in graph G given in Figure 7.13

7.9.2 Does the FF Algorithm Run in Polynomial Time?

We have deduced that the **Ford-Fulkerson algorithm** has complexity of $O(\varphi^* m)$.

It is unclear whether $O(\varphi^* m)$ a **polynomial of the instance size**.

Unfortunately, we can't bound φ^* as a polynomial of n and/or m.

Question: how should we represent the size of φ^*?

7.9.3 Aside: Binary Encoding

We'll use **binary encoding** of numbers in our model of complexity.

Binary encoding is a **positional notation system** with base 2:

- Uses **two symbols:** 0 and 1.
- It is **positional:** same symbol represents different orders of magnitude based on relative position.
- Digit/symbol is called a **bit.**

Example 7.9. Let's convert the **base-2** or **binary encoding** 100110 to **base-10**:

$$\begin{aligned}(100110)_{10} &= 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 \\ &= 2 + 4 + 32 = 38.\end{aligned}$$

Theorem 7.3. *Given a* ***base-10*** *number n, we need*

$$\lfloor \log n \rfloor + 1 = O(\log n)$$

bits to encode it in base-2.

Example 7.10. For example,

$$(38)_{10} = (100110)_2$$

requires

$$\lfloor \log_2(38) \rfloor \approx \lfloor 5.25 \rfloor + 1 = 6$$

bits of storage.

7.9.4 Storage Needs of the Maximum Flow Problem

For an instance of the maximum flow problem, we need to store the **graph topology**, i.e., nodes and adjacencies encoded by arcs, and the **capacity value per arc**.

We can extend our notion of an **adjacency list** to keep track of indices and arc capacities. We will store the capacity of an arc using tuples in a list for each node. This is illustrated in the following example.

Example 7.11. Consider the graph given in Figure 7.19. We can encode the edges within the adjacency list. For example, nodes 2 and t are both adjacent 1, by arcs with capacities 2 and 6 respectively. We can encode this using a modified adjacency list

$$L(1) = \{(2, 2), (t, 6)\}.$$

We can construct adjacency lists for each remaining node using an identical process:

$$\begin{aligned}L(s) &= \{(1, 7), (2, 4)\}, \\ L(2) &= \{(1, 1), (t, 5)\}, \\ L(t) &= \emptyset.\end{aligned}$$

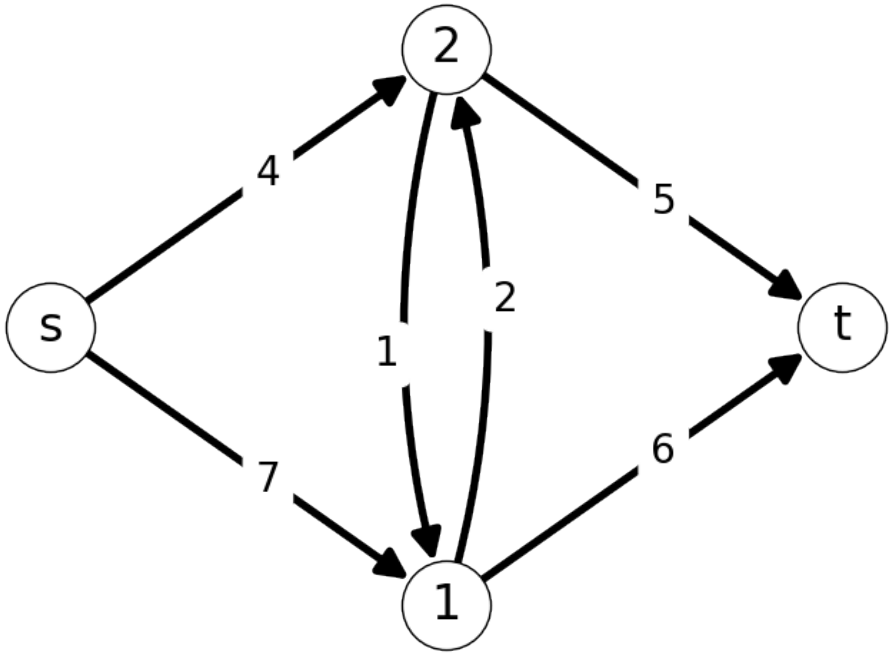


Figure 7.19: Graph G

7.9.5 Instance Size of Maximum Flow

Arc (i, j) corresponds to an entry in **adjacency sublist** $L(i)$.

Using a **binary encoding**, each entry contains:

- The **index** $j \in \{1, \dots, n\}$ encoded using

$$\lfloor \log j \rfloor + 1 = O(\log j) = O(\log n) \text{ bits},$$

  since $\log j = O(\log n)$.

- The **capacity** k_{ij} of arc (i, j) encoded using

$$\lfloor \log k_{ij} \rfloor + 1 = O(\log k_{\max}) \text{ bits},$$

  where $k_{\max} := \max\{k_{ij} : (i, j) \in A\}$.

This **totals** $O\Big(m(\log n + \log k_{\max})\Big)$ bits.

Example 7.12. Consider the graph G given in Figure 7.13. Recall, that solving the maximum flow problem using the Ford-Fulkerson Algorithm has complexity $O(\varphi^*)$. We'd like to know how this complexity $O(\varphi^*)$ compares to instance size $O(m(\log n + \log k_{\max}))$.

Recall that $\varphi^* = 2M$ for this example. Moreover, $k_{\max} = M$. Unfortunately, the instance size depends *logarithmically* on $k_{\max}$, *not linearly*. Indeed,

$$k_{\max} = 2^{\log_2(k_{\max})}.$$

This implies that

$$O(\varphi^* m = O(k_{\max} m) = O\left(2^{\log k_{\max}} \cdot m\right),$$

which grows exponentially in $\log(k_{\max})$.

On the other hand, the instance size $O(m(\log n + \log k_{\max}))$ grows linearly in $\log(k_{\max})$.

Therefore, $O(\varphi^* m)$ is ***not a polynomial function*** of the instance size. Therefore, the Ford-Fulkerson Algorithm is ***not polynomial*** in complexity.

7.9.6 The General Case

This example establishes that the **Ford-Fulkerson Algorithm** is **not polynomial**.

We can generalize the complexity of the FF algorithm **without** relying on φ^*. To avoid using φ^*, which is usually not known, we can substitute an upper bound on φ^*.

Example 7.13. As a simple example, consider a graph with maximum capacity $k_{\max}$ and m arcs. Then

$$\varphi^* \leq m k_{\max}$$

since each arc cannot carry more flow than $k_{\max}$ and there are m arcs. This gives the upper bound on complexity

$$O(\varphi^* m) \leq O\left((k_{\max} m) \cdot m\right) = O\left(k_{\max} m^2\right).$$

Unfortunately, this upper bound is *tight*: there are graphs G for which the Ford-Fulkerson Algorithm uses $\varphi^* = k_{\max} m$ iterations.

7.9.7 Pseudopolynomiality

The complexity $O(k_{\max} m^2)$ is **not polynomial** if we assume a **binary encoding** because $k_{\max} = O(2^{\log k_{\max}})$.

However, the instance size is **polynomial** with respect to instance size if we use a **unary encoding** (encode integers using only 1s):

- For example, 7 is encoded in unary as 1111111.

Using a unary encoding, we need:

- $O(mn)$ symbols to store the graph topology, and
- $O(mk_{\max})$ symbols to store the arc capacities.

We call such an algorithm **pseudopolynomial**.

7.9.8 Polynomial-Time Algorithms for Maximum Flow

Strongly polynomial algorithms do exist for the maximum flow problem. The **Edmonds/Karp algorithm**[2][3] is the most widely used example, but is beyond the scope of this course.

[2] https://cp-algorithms.com/graph/edmonds_karp.html
[3] https://en.wikipedia.org/wiki/Edmonds-Karp_algorithm]

8 Matching Problems

8.1 The Unweighted Matching Problem

8.1.1 Motivation

Given resources and locations, as well as compatibility constraints, we we want to match/assign resources to locations to create as many pairs as possible.

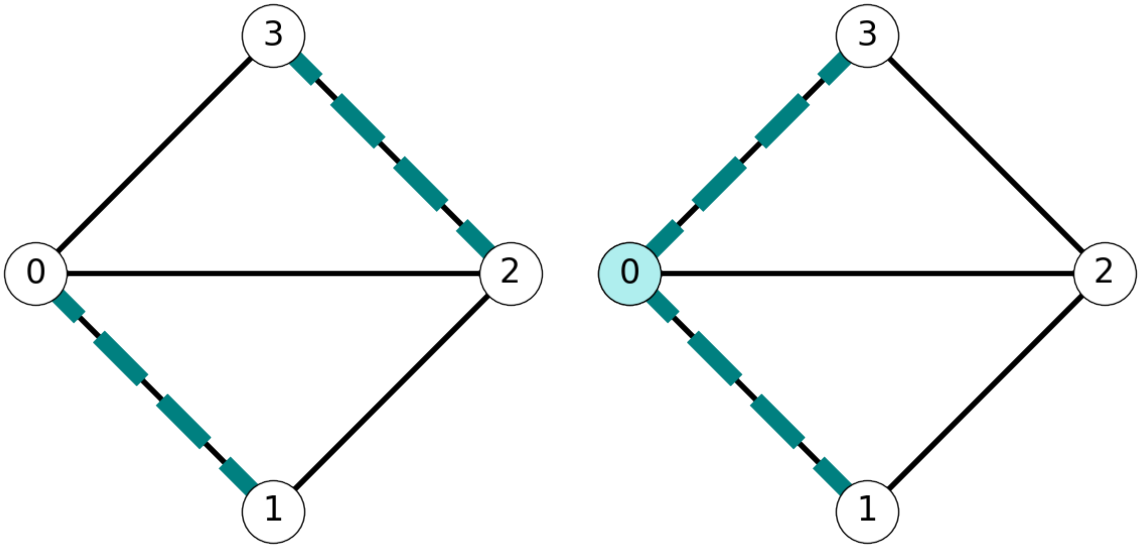
- Assignment of workers to jobs, students to internships, etc;
- Assign flights to gates;
- Assignment of physicians to hospitals, and so on.

8.1.2 Matchings

Definition 8.1 (Matchings)**.** Given an undirected graph $G = (V, E)$ with n nodes and m edges, we call a subset of edges $M \subseteq E$ a **matching** if it satisfies **one** of the following **equivalent properties**:

1. No two edges in M are **incident with the same vertex**.
2. $H = (V, M)$ has node degrees with values 0 or 1.

Example 8.1. Consider the graph G given in Figure 8.1. The set $M = \{01, 23\}$ is a matching. On the other hand $\tilde{M} = \{01, 03\}$ is *not* a matching since it includes two edges incident with 0.



(a) A matching in graph G

(b) Not a matching in graph G

Figure 8.1: Subsets in graph G: $M = \{01, 23\}$ is a matching, but $\tilde{M} = \{01, 03\}$ is not.

8.1.3 Bipartite Graphs

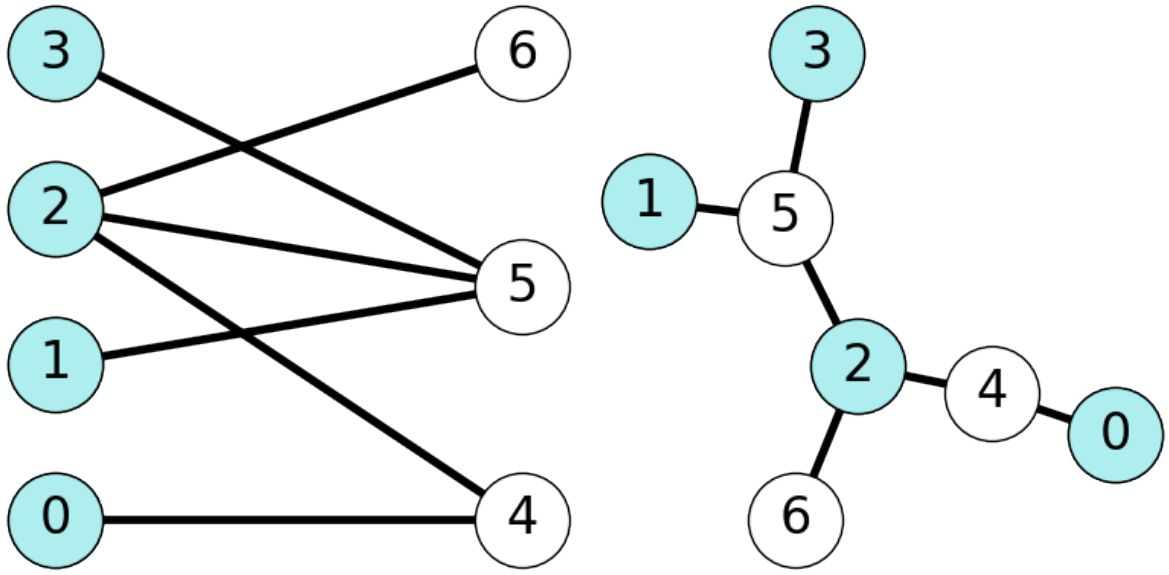
We will focus on the task of finding matchings in special class of graphs, namely, bipartite graphs.

Definition 8.2 (Bipartite Graphs)**.** An undirected graph $G = (V, E)$ is **bipartite** if V can be **partitioned** into two **disjoint sets** A and B such that for each $(i, j) \in E$:

1. Either $i \in A$ and $j \in B$,
2. Or $j \in A$ and $i \in B$.

We call A and B **shores** or **color classes** or **independent sets**.

Example 8.2. Figure 8.2 gives two representations of the same bipartite graph G. The vertices of G consist of two shores $A = \{0, 1, 2, 3\}$ and $B = \{4, 5, 6\}$.



(a) A bipartite graph G (b) Another representation of G

Figure 8.2: Two visualizations of a bipartite graph G.

8.1.4 The Unweighted Matching Problem

Given a bipartite graph G, we want to find the matching of G containing the maximum number of edges.

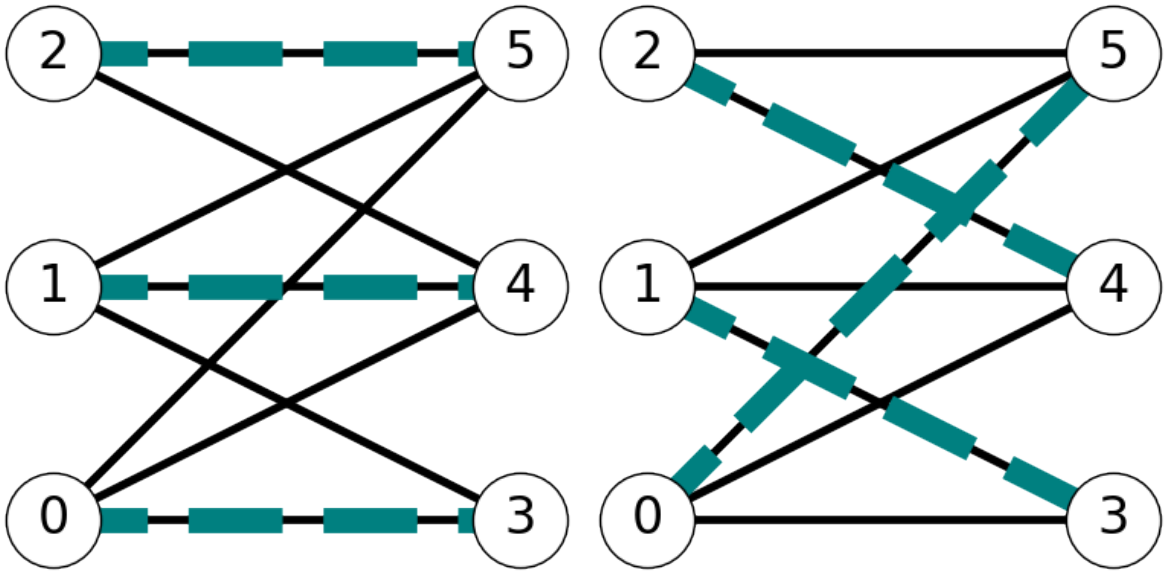
Definition 8.3 (The Unweighted Matching Problem)**.** Given an undirected graph G, find a **matching** $M \subseteq E$ with **maximum cardinality**.

Definition 8.4 (The Matching Number)**.** We call the **cardinality** of a **maximum matching** the **matching number** $\nu(G)$.

Example 8.3. The graph given in Figure 8.3 has matching number $\nu(G) = 3$. Indeed, G contains maximum cardinality matchings $M_1 = \{03, 14, 25\}$ and $M_2 = \{05, 13, 24\}$

8.1.5 The Augmenting Path Algorithm

We will construct an algorithm for solving the maximum matching problem, based on construction of augmenting paths. We start by defining a set of nodes that are not incident with matching edges.



(a) Maximum cardinality matching in G (b) Another maximum cardinality matching in G

Figure 8.3: Maximum matchings in a given graph.

8.1.5.1 Exposed Nodes

Definition 8.5 (Exposed nodes)**.** Given a matching M, we call any unmatched vertex **exposed**.

Example 8.4. Consider the matching $M = \{05, 24\}$ for graph G given in Figure 8.4. The nodes 1 and 3 are *exposed* for this matching.

8.1.5.2 Alternating Paths and Cycles

Definition 8.6 (Alternating Paths)**.** Given a matching M, a path P is an **alternating path (cycle)** if the edges of the path (cycle) **alternate** between M and $E \setminus M$.

Example 8.5. Consider the graph G with matching $M = \{05, 24\}$. Figure 8.5 highlights an *alternating path* $P = (0, 5, 2, 4)$ and an *alternating cycle* $C = (0, 5, 2, 4, 0)$. We use dashed edges to indicate edges in both M and P/C and dash-dotted edges for edges not in M.

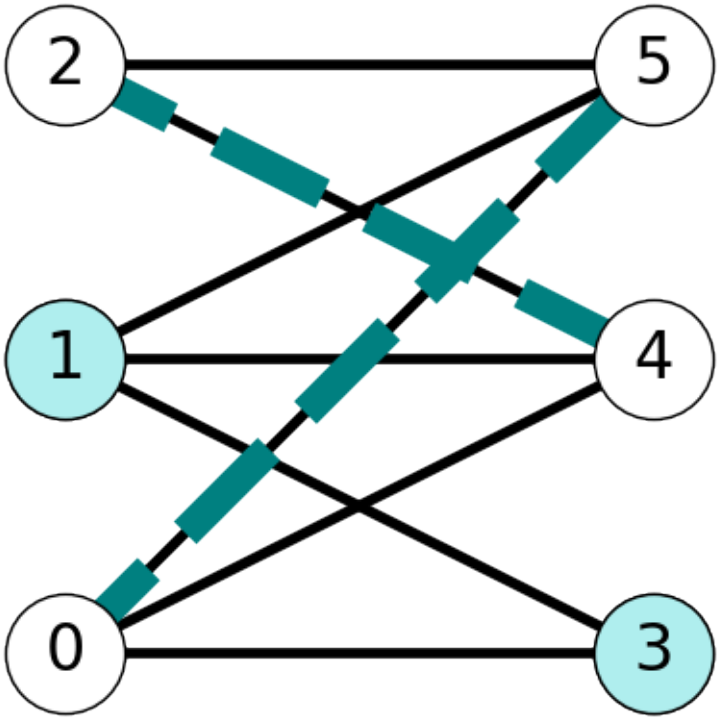
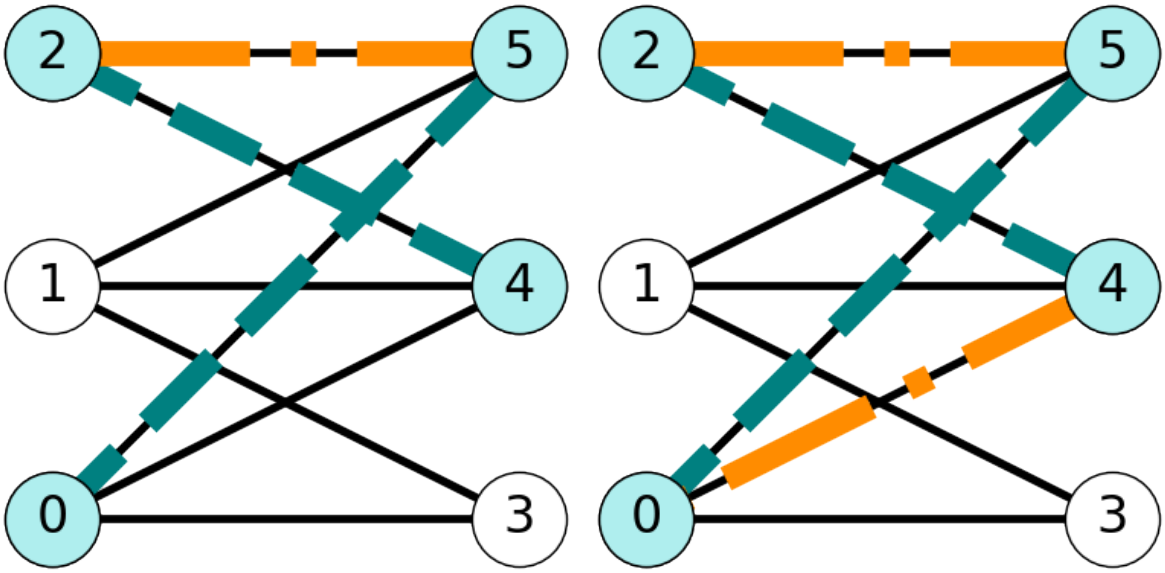


Figure 8.4: Exposed nodes 1 and 3



(a) An alternating path

(b) An alternating cycle

Figure 8.5: Alternating path and alternating cycle in G.

8.1.5.3 Augmenting Paths

We will iteratively identify alternating paths along which to increase the cardinality of a known matching. To do so, we use the following definition of an augmenting path.

Definition 8.7 (Augmenting Path)**.** An **alternating path** is an **augmenting path** if it **starts** and **ends** with an **exposed node**.

Example 8.6. Consider G with matching $M = \{05, 24\}$ again. Recall that the nodes 1 and 3 are exposed. The path $P = (1, 5, 0, 3)$ is an *augmenting (1,3)-path* Indeed, the edges of this path alternate between edges not in M and those in M, and the path starts and ends with exposed nodes.

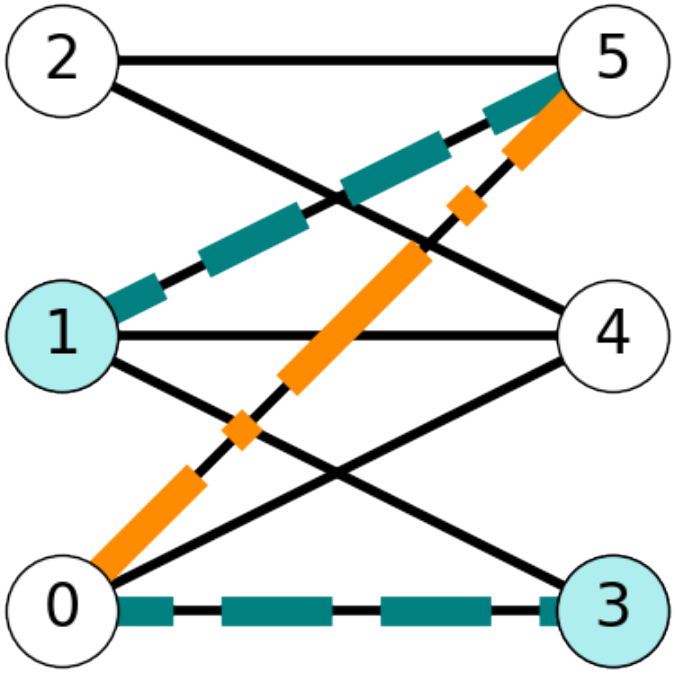


Figure 8.6: An augmenting path $P = (1, 5, 0, 3)$

8.1.5.4 Properties of Augmenting Paths

Lemma 8.1. *Augmenting paths contain an* ***even number of nodes*** *and* ***odd number of edges****.*

Proof. Suppose that P is an augmenting path for matching M. Since P is augmenting, it starts and ends with an exposed node.

This implies that the first and last edges of P are not in M. Consequently, if P contains k edges in M, then P must contain $k+1$ edges in $E \setminus M$ since P is an alternating path. Therefore, P contains $2k+1$ edges total, which is odd for every nonnegative integer k. □

Theorem 8.1 (The Augmenting Path Theorem)**.** *A matching M has **maximum cardinality** if and only if it admits no augmenting paths.*

Proof. We start by proving *necessity*. Suppose that M is a maximum cardinality matching; we want to show that M has not augmenting paths.

We'll do so by contradiction. Suppose that M has augmenting path P. Let's consider the *symmetric difference* of M and P:

$$\begin{aligned} M' = M \Delta P &= (M \cup P) \setminus (M \cap P) \\ &= (M \setminus P) \cup (P \setminus M). \end{aligned}$$

Let $P = ((v_1, v_2), (v_2, v_3), \ldots, (v_{2k+1}, v_{2k+2}))$; we can assume that P contains an even number of nodes by Lemma 8.1.

We'll use this fact to construct a matching with more edges than M:

- We know that $(v_1, v_2) \notin M$ since v_1 is exposed. This implies that $(v_1, v_2) \in M'$.
- Next, (v_2, v_3) is in $M \cap P$ because P is an alternating path. Therefore, $(v_2, v_3) \notin M'$.

We can continue this pattern to establish that M' is a matching. If P contains k edges in M and $k+1$ edges in $E \setminus M$, then M' must consist of the $k+1$ edges in $E \setminus M$. This implies that M is not a maximum matching; a contradiction. Therefore, if M is a maximum matching then no augmenting path exists.

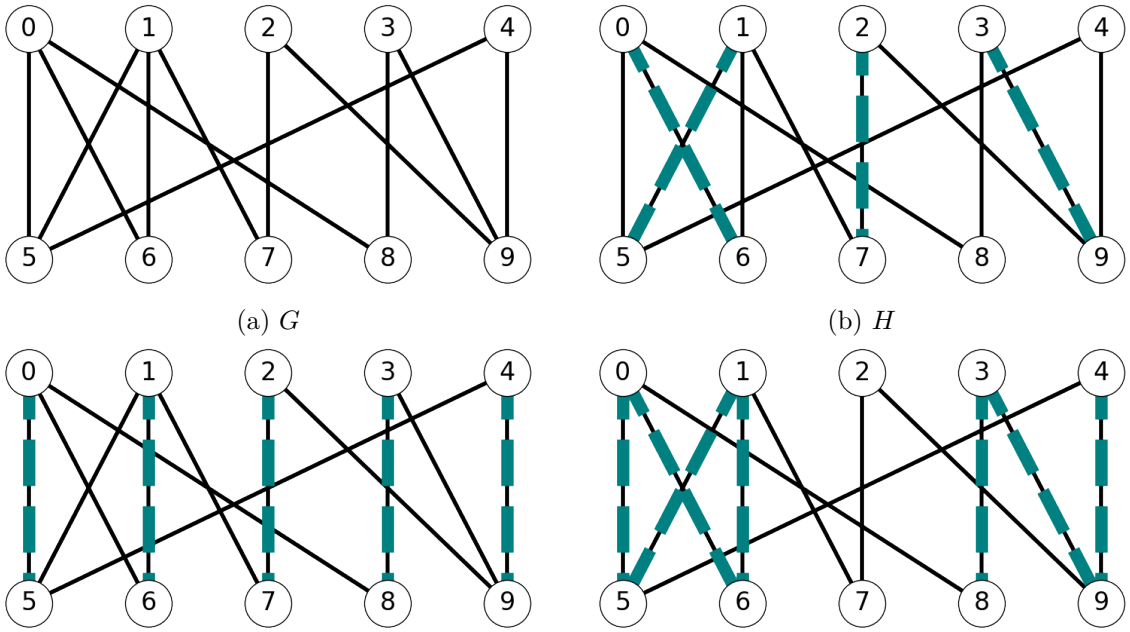
Let's now prove *sufficiency*. Suppose that there is no augmenting path. Let's assume that M is *not* a maximum matching and find a contradiction.

By assumption, there is some matching M' with $|M'| > |M|$. Let's consider the symmetric difference

$$M'' = M \Delta M' = (M \setminus M') \cup (M' \setminus M).$$

Let's consider three subgraphs of G:

1. $H = (V, M)$;
2. $H' = (V, M')$; and
3. $H'' = (V, M'')$.



(a) G (b) H

(c) H' (d) H''

Figure 8.7: Example of bipartite graph G, with subgraphs H, H', and H''.

Figure 8.7 gives an example illustrating the construction of these three subgraphs.

Let's consider *maximally connected subgraphs* of H''; these are subgraphs that adding an node creates a disconnected subgraph of G. Note that:

- Nodes in H and H' have either degree 0 or 1, since H and H' are both induced by matchings.
- This implies that the nodes in H'' have degrees in $\{0, 1, 2\}$.

Therefore, each maximally connected component of H'' is one of:

- an isolated node;
- an alternating path; or
- an alternating cycle.

Since $|M'| > |M|$ there is some maximally connected subgraph C with more edges from M' than edges from M. Note that C is not an isolated node. Moreover, C is not an alternating cycle, because an alternating cycle would contain the same number of edges from M and M'. Therefore, C is an alternating path. Further, C is an *augmenting path*. Indeed, Its end points are exposed with respect to M, since it is alternating and contains more edges from M' than M. □

8.1.6 The Augmenting Path Algorithm

While augmenting path P exists:

- Find an augmenting path P
- If $P \neq \emptyset$:
  - $M \leftarrow (M \cup P) \setminus (M \cap P)$.
- Else:
  - **STOP** (exit loop).

8.1.7 Finding an Augmenting Path

We need an algorithmic way to identify an augmenting path. To do so, we will work with an auxiliary graph.

8.1.7.1 The Auxiliary Graph

We build **an auxiliary directed bipartite graph** $G' = (V, E')$ where, for each $\{i, j\} \in E$, $i \in A$, $j \in B$:

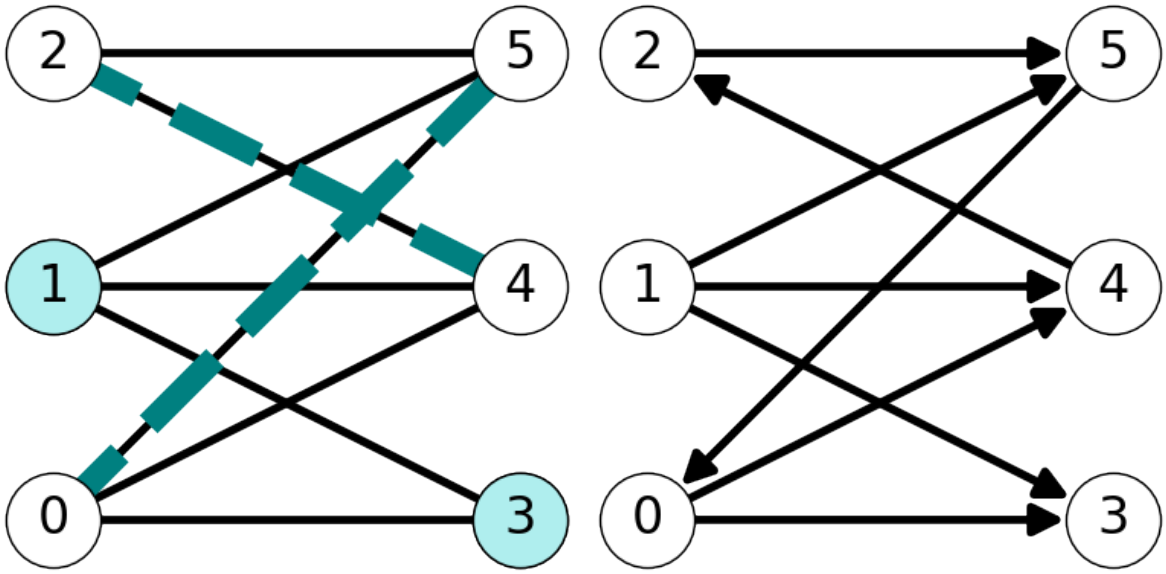
- Add (i, j) to E' if $\{i, j\} \notin M$.
- Add (j, i) to E' if $\{i, j\} \in M$.

The vertex set A can be left only via arcs corresponding to edges **not in** M.

The vertex set B can only be left via arcs corresponding to edges **in** M.

This implies that paths in G' are **alternating**.

- Paths between **exposed nodes** $i \in A$ and $j \in B$ are **augmenting**.



(a) $M = \{05, 24\}$ (b) Auxiliary graph

Figure 8.8: Construction of the auxiliary graph for graph G and matching $M = \{05, 24\}$.

8.1.7.2 A Practical Algorithm for Finding an Augmenting Path

To find an **augmenting path** using G':

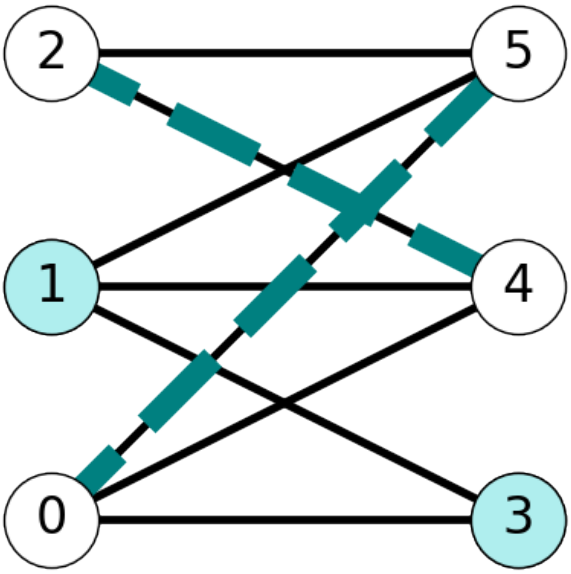
1. Introduce two extra nodes s, t.
2. Connect s to all the exposed nodes in A.
3. Connect t to each exposed node in G.
4. Look for an st-path in the **extended version** of G'.

Example 8.7. Figure **??** illustrates the construction of the extended auxiliary graph with graph G containing matching $M = \{05, 24\}$.

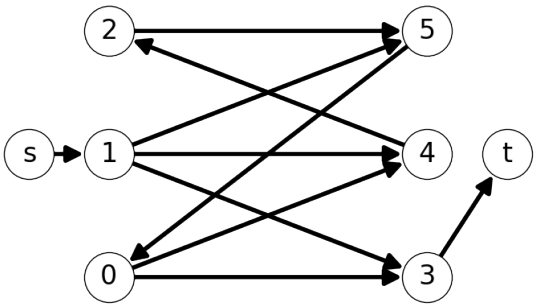
8.1.8 Complexity of the Augmenting Path Algorithm

8.1.8.1 Operations per Iteration

To find an augmenting path P costs $O(m + n)$. Indeed, we need to construct the auxiliary graph and then apply a variant of graph reachability to find an st-path in the auxiliary; both steps cost $O(m + n)$ elementary operations.



(a) Graph G with matching $M = \{05, 24\}$



(b) Extended auxiliary graph

On the other hand, updating M as $M\Delta P$ costs $O(n)$ operations.

Therefore, the total complexity is

$$\text{maximum \# of iterations } \times O(m+n).$$

8.1.8.2 Total Number of Iterations

Notat that a matching M has at most $n/2$ edges in a bipartite graph. Indeed,

$$\nu(G) \leq \min\{|A|, |B|\} \leq \frac{n}{2}.$$

The augmenting path algorithm increases the cardinality of a matching by 1 each iteration. Therefore, the augmenting path algorithm has total complexity bounded above by

$$O\Big(\nu(G) \times (m+n)\Big) = O(n^2 + mn).$$

If G is connected, we have $n \leq m+1 = O(m)$. In this case, we have total complexity $O(mn)$.

8.1.9 Example

Example 8.8. Consider the graph G given in Figure 8.10 with initial matching

$$M_0 = \{16, 27, 39\}.$$

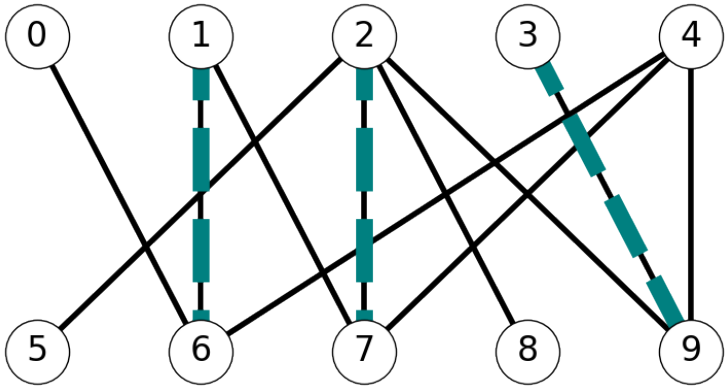


Figure 8.10: Graph G for Example 8.8 with matching M_0.

8.1.9.1 Step 1

We start by constructing the auxiliary graph G'. After doing so, we can observe that there is an st-path $P' = (s, 4, 7, 2, 5, t)$ in G'. This gives the augmenting path $P = (4, 7, 2, 5)$.

We update the matching

$$M_1 = M_0 \Delta P = (M_0 \cup P) \setminus (M_0 \cap P) = \{16, 39, 25, 47\}.$$

8.1.9.2 Step 2

As before, we construct the auxiliary graph G' (Figure 8.12a). Note that t is *not reachable* from s in G'; Figure 8.12b gives the reachable set from s in G'.

This implies that there are no augmenting paths for matching M_1. Therefore, M_1 is a maximum matching.

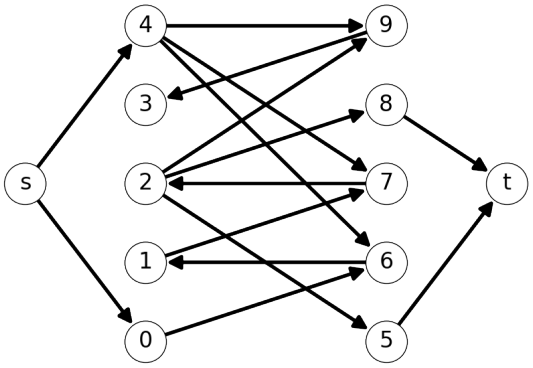
8.2 The Weighted Matching Problem

8.2.1 Problem Definition

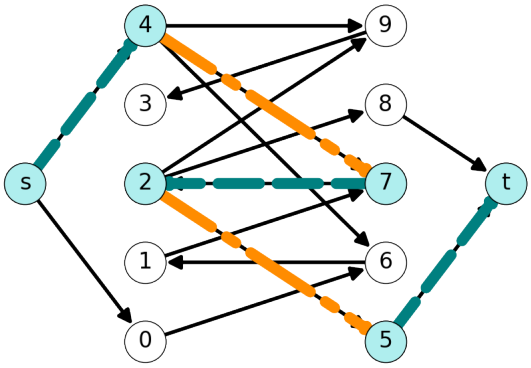
Definition 8.8 (Weighted Matching Problem)**.** Given an undirected graph $G = (V, E)$ and a **weight function** $w : E \mapsto \mathbf{R}$, find a **matching** $M \subseteq E$ of **maximum weight**:

$$w(M) = \sum_{\{i,j\} \in M} w_{ij}.$$

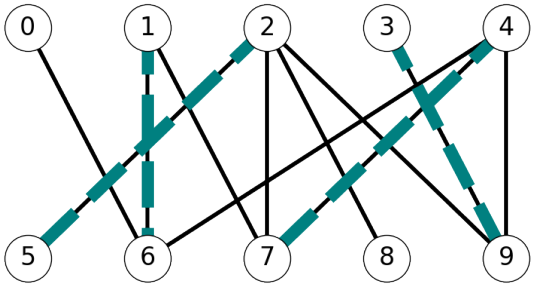
Definition 8.9 (Weighted Matching Number)**.** We call the **weight of a maximum-weight matching** of G the **weighted matching number** and denote it $\nu_w(G)$.



(a) Auxiliary graph G'

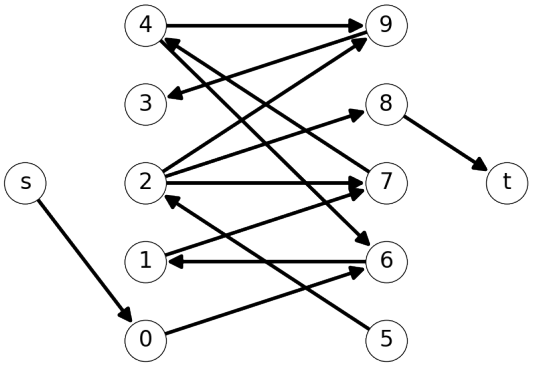


(b) Augmenting path P'

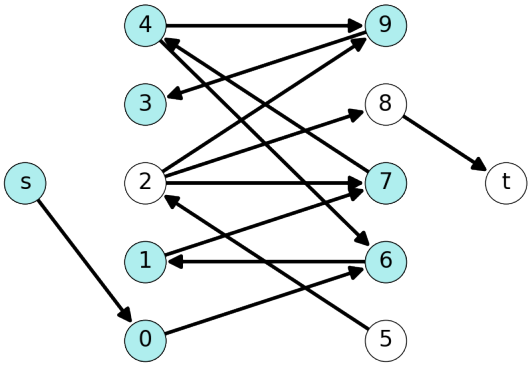


(c) New matching M_1

Figure 8.11: Details of the first step of the augmenting path algorithm.



(a) Auxiliary graph G'



(b) Reachable nodes from s

Figure 8.12: Details of the second and final step of the augmenting path algorithm.

8.2.2 Extremality

Definition 8.10 (Extreme Matchings)**.** We call a matching **extreme** if it has **maximum weight among all matchings of the same cardinality** $|M|$.

Question: We know how to construct another matching of cardinality one unit larger using the **unweighted matching algorithm**.

- Can we construct this matching so that it is **extreme?**
- If so, we can build an extreme matching M_1 from an empty matching M_0. From M_1, we can build an extreme matching M_2 of size 2, and so on.

8.2.2.1 An Observation

Theorem 8.2. *If $M_1, M_2, \dots, M_{\nu(G)}$ is a sequence of **extreme matchings of increasing size**, then*

$$M = arg\ max\Big\{w(M_i) : i = 1, 2, \dots, \nu(G)\Big\}$$

is a maximum-weight matching.

Here, the argmax operator returns a extreme matching with maximum weight:

- if a maximum weight matching has cardinality 1, then M_1 is a maximum weight matching;
- if a maximum weight matching has cardinality 2, then M_2 is a maximum weight matching;
etc.

8.2.3 Augmentation and Weights of Matchings

Let M be a matching and let be P an **augmenting path**.

Let

$$M' := M \Delta P = (M \cup P) \setminus (M \cap P) = (M \setminus P) \cup (P \setminus M).$$

The weight of M' is

$$w(M') = \sum_{ij \in M'} w_{ij} = w(M) + \underbrace{\sum_{e \in P \setminus M} w_i - \sum_{e \in P \cap M} w_e}_{=:(*)}.$$

The summand $(*)$ is a linear combination of weights of edges in P with

- coefficient $+1$ if $e \notin M$;

- coefficient -1 if $e \in M$.

Let's introduce new weights and rewrite the identity:

$$\ell_{ij} = \begin{cases} -w_{ij} & \text{if } ij \notin M, \\ +w_{ij} & \text{if } ij \in M. \end{cases}$$

Using these lengths, we have

$$w(M') = w(M) - \sum_{ij \in P} \ell_{ij}.$$

This implies that the increase in weight is **maximized** if the length of P is **minimized**.

8.2.4 The Shortest Augmenting Path Theorem

Question: Suppose that we augment an **extreme matching** M by a **shortest augmenting path**.

- Does this yield an **extreme matching** M' (with cardinality $|M'| = |M| + 1$)?

We have the following theorem.

Theorem 8.3. *Given an **extreme matching** M and an **augmenting path** P of **minimum length**.*

*The matching $M' := M \Delta P$ is an **extreme matching** of cardinality $|M'| = |M| + 1$.*

Proof. We'll use *contradiction*. Let's assume that M' is *not* extreme. Then there is N' with $|N'| = |M'|$ such that

$$w(N') > w(M').$$

We know that $|N'| = |M'| > |M|$. By an identical argument to the proof in the unweighted case, the subgraph

$$G_{M \Delta N'} = (V, M \Delta N')$$

has a maximal connected component C with maximal connected component C with more edges from N' than M, i.e., an alternating path. This alternating path augments M. (Notes C may not be P.)

Earlier, in the unweighted case, we used C to extend M to a larger matching. Here, we'll do the opposite: we'll shrink N' using C:

$$N = N' \Delta C.$$

Note that

$$|N| = |N'| - 1 = |M|.$$

We have

- four matchings M, N, M', N';
- two augmenting paths P and C.

By assumption $\ell(P) \leq \ell(C)$. Moreover, $w(M) \geq w(N)$. It follows that

$$w(M') < w(N') = w(N) - \ell(C) \leq w(M) - \ell(P) = w(M').$$

This is a contradiction. Therefore, M' is an extreme matching. □

8.2.5 The Iterative Shortest Path Algorithm

8.2.5.1 The Algorithm

Initialize $M_0 = \emptyset$.

For $i = 0, 1, 2, \ldots, \nu(G)$:

- Find a **shortest M_i-augmenting path** P.
- If $P \neq \emptyset$:
  - $M_{i+1} \leftarrow (M_i \cup P) \setminus (M_i \cap P)$.
  - $w(M_{i+1}) \leftarrow w(M_i) - \ell(P)$
- Else:
  - **STOP** (exit loop).

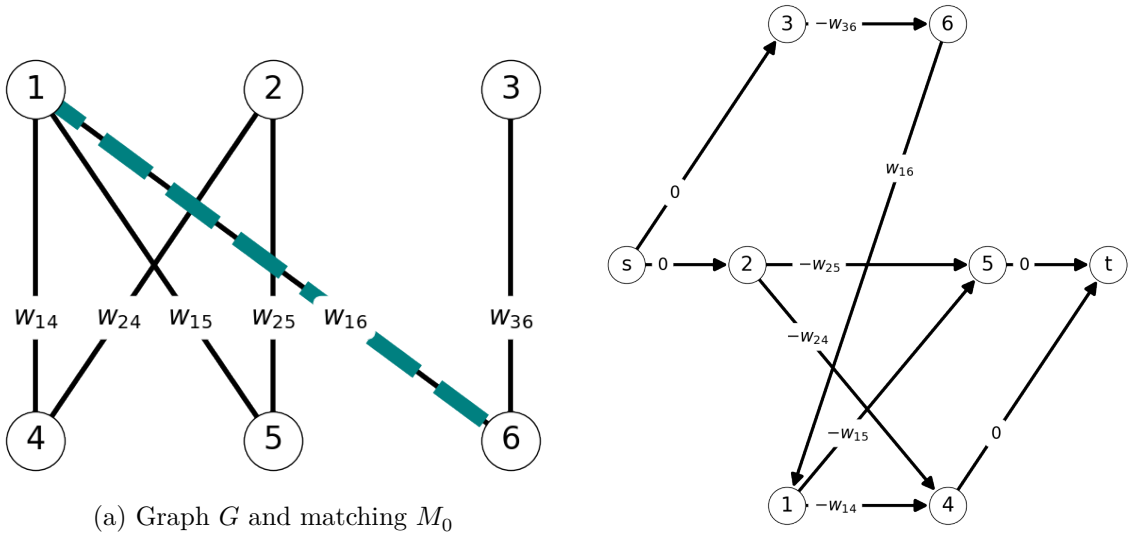
Return $M = \arg\max\{w(M_i)\}$.

8.2.5.2 Finding a Shortest Augmenting Path

We need to find augmenting paths. To do so, we construct an auxiliary graph and identify the shortest path, which will yield the desired augmenting path.

- Build an **auxiliary directed graph** $G' = (V \cup \{s, t\}, A')$ for matching M_i:
- For each $\{i, j\} \in E$:
  - Add (i, j) to A' if $\{i, j\} \notin M_i$ with length $\ell_{ij} = -w_{ij}$.
  - Add (j, i) to A' if $\{i, j\} \in M_i$ with length $\ell_{ij} = w_{ij}$
- Connect s and t to/from the exposed nodes in the two shores with **zero length arcs**.
- Finally, find a **shortest st-path** in G'.

Example 8.9. Figure 8.13 illustrates the construction of the auxiliary graph for a given matching M_0.



(a) Graph G and matching M_0

(b) G'

Figure 8.13: Auxiliary graph G' for graph G and matching $M_0 = \{16\}$.

8.2.5.3 Cycles in Bipartite Graphs

Question: The auxiliary graph G' has edges with **negative lengths**.

- Is it possible to have cycle with negative length?
- In this case, the shortest-path problem is ill-posed.

We will use the following lemma to address this possible issue.

Lemma 8.2. *Every **directed bipartite graph** G **does not contain odd cycles***

Proof. We use contradiction. Let's assume that G has a cycle C with odd length. Specifically, assume C has $2k+1$ nodes for some positive integer k; label these nodes as $v_1, v_2, \ldots, v_{2k+1}$.

Since G is bipartite, we change shore with each edge. That is, v_i, v_{i+1} are in different shores. The last edge in the cycle is (v_{2k+1}, v_1). Since C has odd length, v_1 and v_{2k+1} are in the same shore of G. Thus, this edge cannot exist; a contradiction.

Therefore, any cycle in G has even length. □

The following theorem confirms that the auxiliary graph does not have negative-length cycles if the matching M is extreme.

Theorem 8.4. *If M is an **extreme matching** then G' **does not contain** negative-length cycles.*

Proof. To obtain a contradiction, suppose C is a cycle with negative length:

$$\ell(C) = \sum_{e \in C} \ell_e < 0.$$

Since G' is bipartite, C has even length by Lemma 8.2. Moreover, C is alternating by the orientation of arcs of G'.

Now, consider $M' = M \Delta C$. Note that

$$|M'| = |M|.$$

Indeed, C is even so half of the edges in $M \cup C$ are in $M \cap C$. Thus,

$$|C| = |M| + |M'|$$

is even and $|M| = |M'|$..

Now consider the weight of M':

$$w(M') = w(M) - \ell(C) > w(M)$$

since $\ell(C) < 0$. Therefore, M is not an extreme matching. □

8.2.6 Complexity

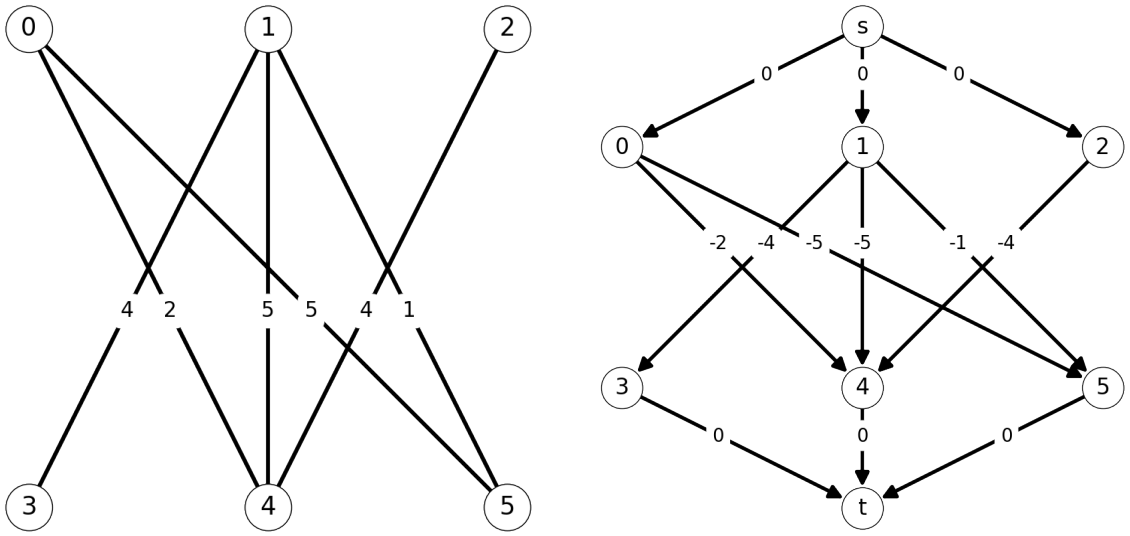
For each $i = 0, 1, 2, \ldots, \nu(G)$, we need $O(n + m)$ operations to form the auxiliary graph G' and $O(nm)$ operations to find the shortest st-path using the Bellman-Ford Algorithm. After identifying a shortest M_i-augmenting path, we need a further $O(n)$ operations to compute M_{i+1}. Thus, each iteration has complexity $O(mn)$.

We repeat this process $\nu(G) = O(n)$. Thus, the total complexity is

$$O(\nu(G)mn) = O(n^2 m).$$

8.2.7 Example

Example 8.10. Consider the graph G given in Figure 8.14.



(a) G (b) G'

Figure 8.14: Graph G with initial matching $M_0 = \emptyset$ and corresponding auxiliary graph G'.

Step 1

We start by finding the shortest st-path in G'. Here, $(s, 1, 4, t)$ and $(s, 0, 5, t)$ are both shortest st-paths (with length -5). We can use either as an augmenting path. Let's choose $(s, 1, 4, t)$ to use as an augmenting path and add 14 to M_0 to get M_1 :

$$M_1 = M_0 \Delta P_0 = \emptyset \Delta \{14\} = \{14\}; \quad w(M_1) = 5.$$

Step 2

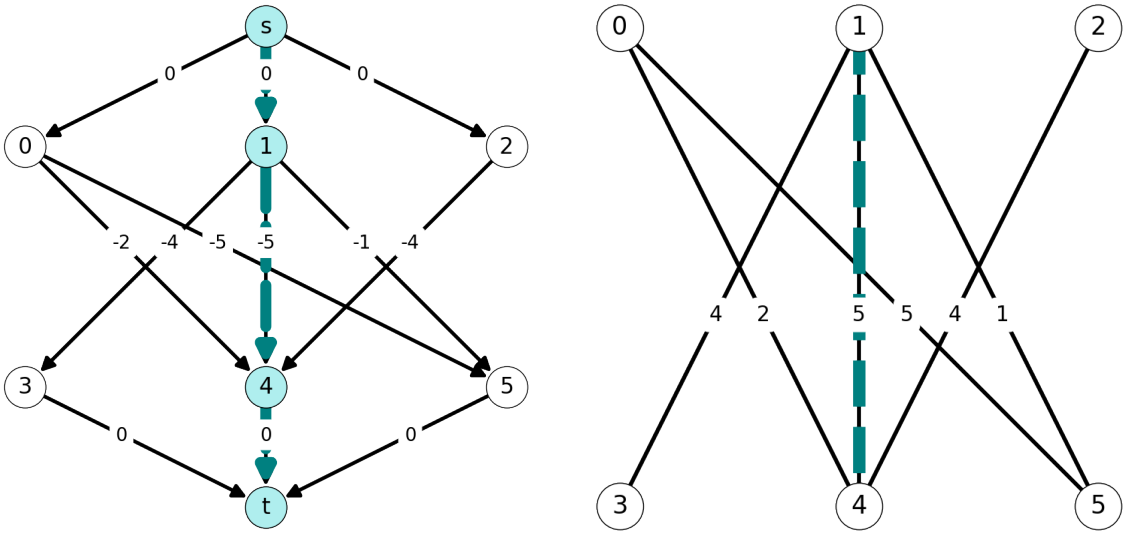
The shortest st-path in the auxiliary graph G' is $(s, 0, 5, t)$. We use the path $P_1 = \{05\}$ to update the matching:

$$M_2 = M_1 \Delta P_1 = \{14, 05\}; \quad w(M_2) = 10.$$

Step 3

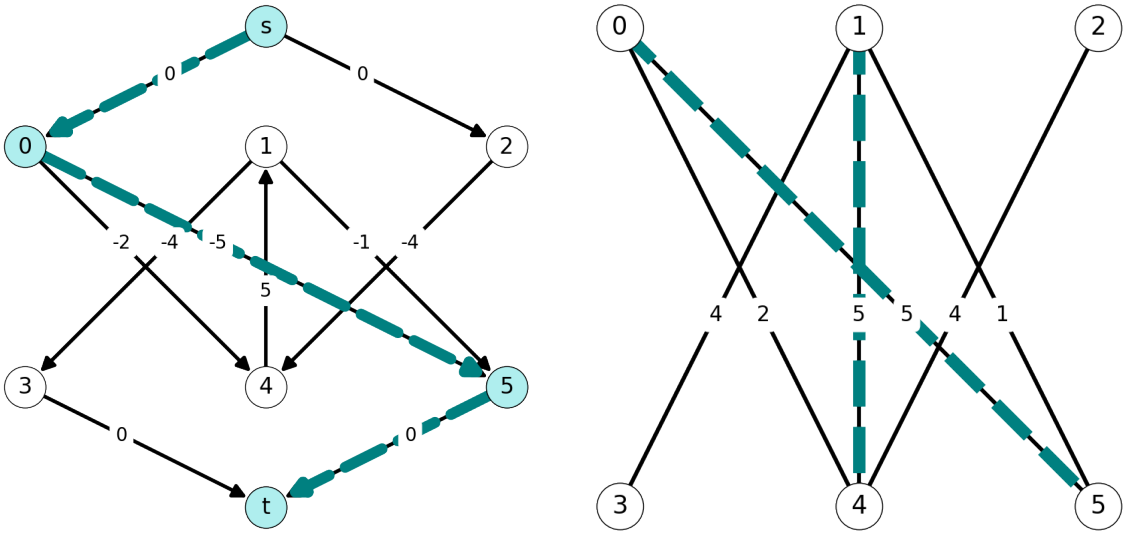
Next, the shortest st-path is $(s, 2, 4, 1, 3, t)$. We compute M_3 using $P_2 = (2, 4, 1, 3)$:

$$M_3 = M_2 \Delta P_2 = \{05, 24, 13\}; \quad w(M_3) = 13.$$



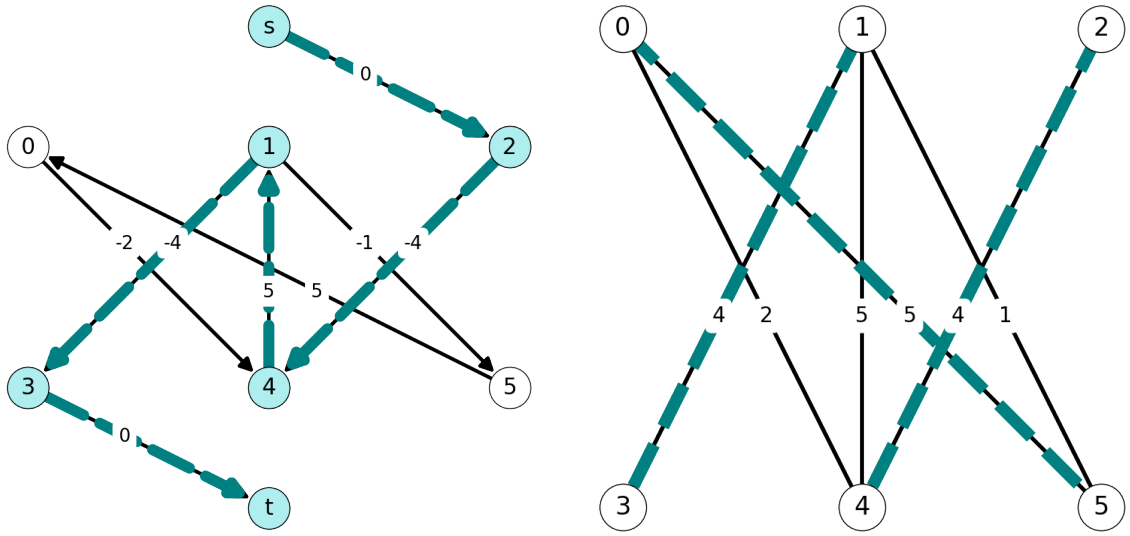
(a) Shortest st-path in G' (b) M_1

Figure 8.15: Matching M_1 following one iteration of augmenting path algorithm.



(a) Shortest st-path in G' (b) M_2

Figure 8.16: Details of Step 2 of the augmenting path algorithm.



(a) Shortest st-path in G' (b) M_3

Figure 8.17: Details of Step 3 of the augmenting path algorithm.

Termination

Each shore has three nodes and we have found a matching with three edges. This implies that the unweighted matching number is $\nu(G) = 3$. We can stop the algorithm.

Note that

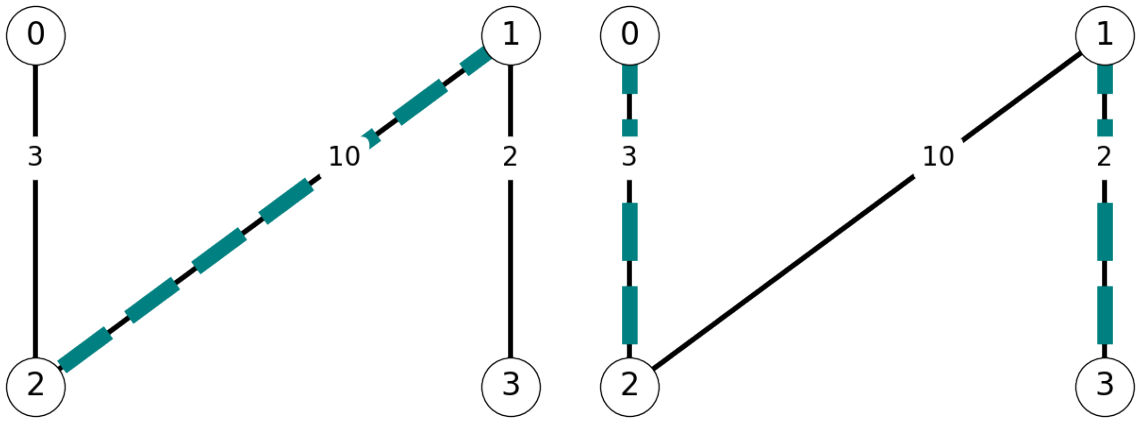
$$w(M_3) = 13 = \max\{w(M_0), w(M_1), w(M_2), w(M_3)\}.$$

Therefore, M_3 is the maximum weight matching.

8.2.8 A Final Remark

The extreme matching of maximum cardinality $M_\nu(G)$ is ***not necessarily the maximum weight matching!***

Example 8.11. Consider the graph given in Figure 8.18. This graph has *maximum weight* matching $M_1 = \{12\}$ and *maximum cardinality* matching $M_2 = \{02, 13\}$.



(a) $M_1 = \{12\}$ (b) $M_2 = \{02, 13\}$

Figure 8.18: A graph where the maximum cardinality matching is ***not*** the maximum weight matching.

8.3 The Assignment Problem

8.3.1 Preliminaries

Definition 8.11 (The Assignment Problem)**.** Given two groups with k items each and a ***pairwise cost*** a_{ij} for each $i, j = 1, 2, \ldots, k$, the **assigment problem** seeks an assignment of the objects from the first group to the second such that:

1. Each object is assigned from group 1 to exactly one item in group 2.
2. The total assignment cost is minimized.

8.3.1.1 Permutations

A **permutation** is a bijective function π mapping $\{1, 2, \ldots, k\}$ onto itself.

The **assignment problem** can be expressed in terms of permutations:

- Given a $k \times k$ matrix A, find a ***permutation*** π of $\{1, 2, \ldots, k\}$ maximizing

$$\sum_{i=1}^{k} a_{i,\pi(i)}.$$

Example 8.12. Suppose $k = 3$ and consider the **permutation** π and **pairwise costs** A given by

$$\pi(1) = 3,\ \pi(2) = 1,\ \pi(3) = 2, \qquad A = \begin{pmatrix} -5 & 2 & 3 \\ 4 & 1 & 3 \\ 7 & 2 & -4 \end{pmatrix}.$$

Then the **cost of the assignment according to** π is

$$a_{1,\pi(1)} + a_{2,\pi(2)} + a_{3,\pi(3)} = a_{13} + a_{21} + a_{32} = 3 + 4 + 2 = 9.$$

8.3.2 Connection to Maximum Matchings

We can cast an instance of the **assignment problem** as an instance of **maximum weighted matching problem**:

- Construct ***complete bipartite graph*** $G = (U \cup V, E)$ with
$$U = \{1, \dots, k\}, \qquad V = \{1', \dots, k'\}.$$
- Reverse signs of costs $w_{ij} = -a_{ij}$ to obtain a ***maximization*** problem.
- Impose constraint $|M| = v(G) = n/2 = k$ to ensure assignment of all objects.

It suffices to apply the **augmenting path algorithm** for the maximum weighted matching problem with ***one change***:

- The minimum cost assignment is the **last** matching $M_{\nu(G)}$.
- This ensures that the assignment constraint is met.

8.3.3 Complexity of the Assignment Problem

We can solve the weighted matching problem using $O(n^2 m)$ EOs using the augmenting path algorithm.

Question: How does this translate to an instance size of the assignment problem?

- Let n is the number of vertices G in the bipartite representation of the assignment problem with k nodes in each shore U and V:
$$n = |U| + |V| = k + k = 2.$$
- G is complete bipartite, so it has
$$k^2 = |U| \times |V|$$
edges.

This implies that we need

$$O(n^2m) = O(k^4)$$

elementary operations to solve the assignment problem. This is quadratic in the number, k^2, of pairwise assignment costs. Thus, the modified augmenting path algorithm solves the assignment problem using at most a polynomial of its instance size operations.

8.3.4 Example

Example 8.13. Let's solve the instance of the assignment problem given by cost matrix

$$A = \begin{pmatrix} -5 & 2 & 3 \\ 4 & 1 & 3 \\ 7 & 2 & -4 \end{pmatrix}.$$

8.3.4.0.1 * Initialization

We start by constructing the weight matrix W and constructing the corresponding instance of the maximum weighted matching problem:

$$W = -A = \begin{pmatrix} +5 & -2 & -3 \\ -4 & -1 & -3 \\ -7 & -2 & +4 \end{pmatrix}.$$

Associating rows in A and W with nodes $0, 1, 2$ and columns $3, 4, 5$ yields an instance of the maximum weight matching problem corresponding to the graph G in Figure 8.19.

8.3.4.0.2 * Step 1

We construct the auxiliary graph G' corresponding to the initial matching M_0 with 0 edges. The shortest st-path in G' is $(s, 0, 3, t)$. We use the subpath $P_0 = (0, 3)$ to compute M_1:

$$M_1 = M_0 \Delta P_0 = \{03\}; \quad w(M_1) = 5.$$

8.3.4.0.3 * Step 2

After updating the auxiliary graph, we see that shortest st-path in G' is $(s, 2, 5, t)$. We let $P_1 = (2, 5)$ and set

$$M_2 = M_1 \Delta P_1 = \{03, 25\}; \quad w(M_2) = 9.$$

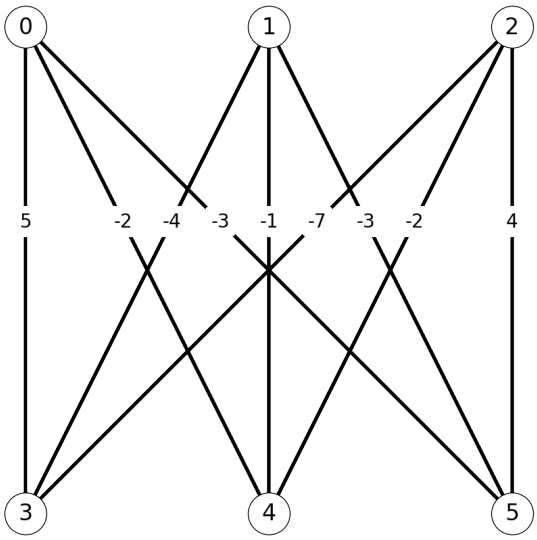
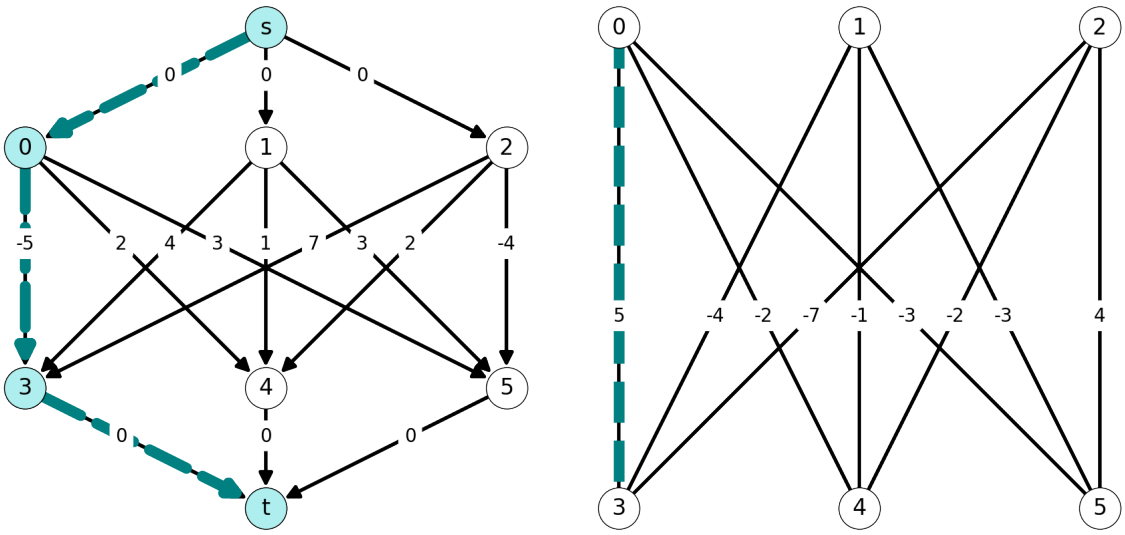


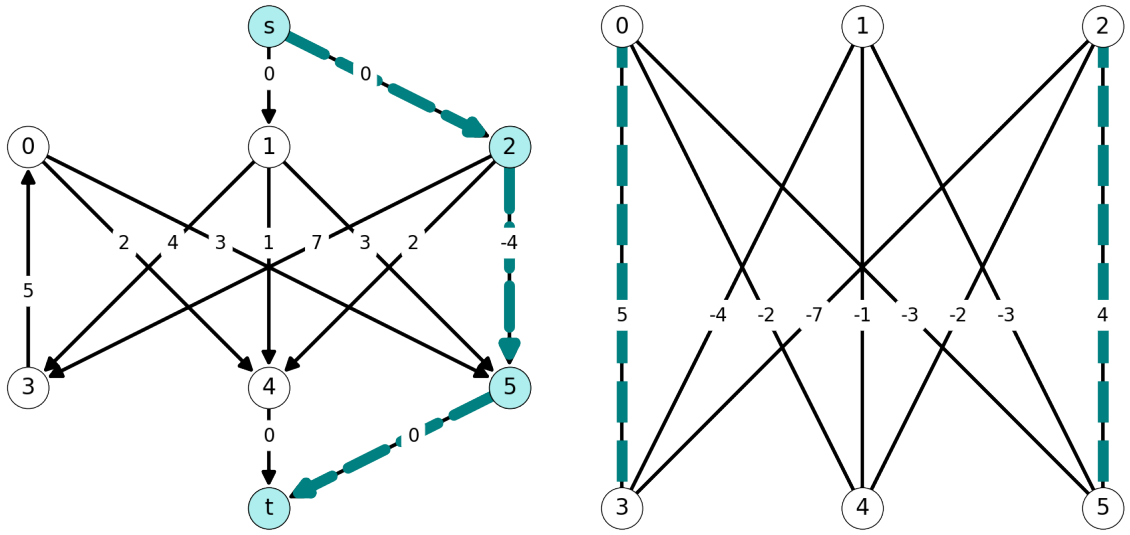
Figure 8.19: Graph G corresponding to the assignment problem with costs A.



(a) Shortest st-path in G'

(b) Matching M_1

Figure 8.20: Details of the first iteration of the augmenting path algorithm.



(a) Shortest st-path in G'

(b) Matching M_2

Figure 8.21: Details of the second iteration of the augmenting path algorithm.

8.3.4.0.4 * Step 3

The shortest st-path in the auxiliary graph G' with respect to M_2 is $(s, 1, 4, t)$. We let $P_2 = (1, 4)$ and obtain the matching

$$M_3 = M_2 \Delta P_2 = \{03, 25, 14\}$$

with $w(M_3) = 9 - 1 = 8$.

8.3.4.0.5 * Termination

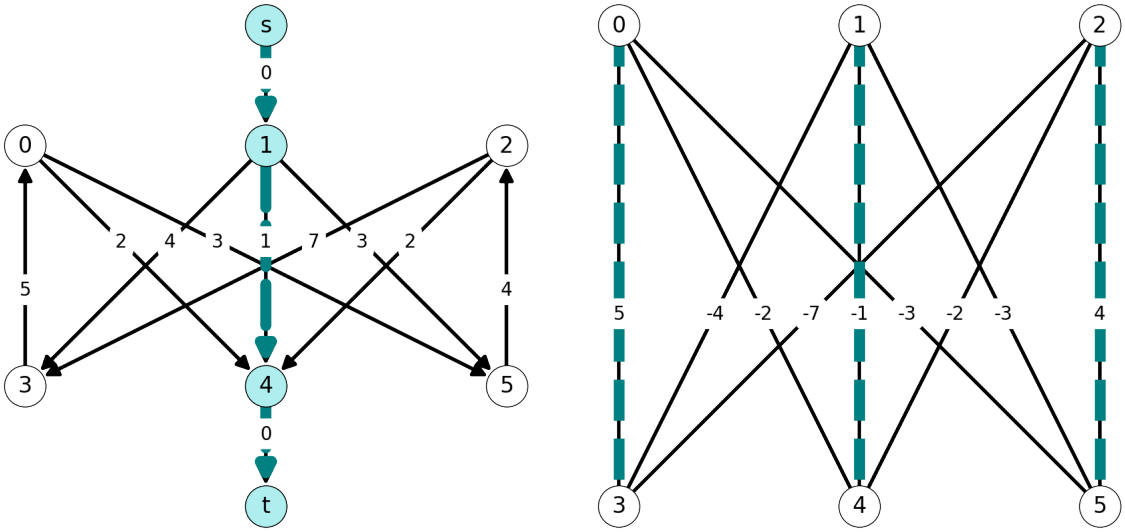
Since $\nu(G) = 3$, we stop the algorithm.

- The maximum weight matching is M_2 with $w(M_2) = 9$.
- The minimum cost assignment is given by M_3:

  $$\pi(0) = 0, \quad \pi(1) = 1, \quad \pi(2) = 2$$

  with minimum cost

  $$\sum_{i=0}^{2} a_{i,\pi(i)} = a_{11} + a_{22} + a_{33} = -w(M_3) = -8.$$



(a) Shortest st-path in G'

(b) Matching M_3

Figure 8.22: Details of the third iteration of the augmenting path algorithm.

9 Complexity of Problems

9.1 Complexity Classes

9.1.1 Problem Complexity

Recall the **Cobham-Edmonds thesis**[1]: P is **tractable** or **well-solved** if

1. There is an algorithm A which solves it.
2. A has ***computational complexity*** which is a ***polynomial function of the instance size*** of P.

We have seen several problems which are *tractable*:

- minimum spanning tree
- shortest path
- max flow / min cut problem
- matching (unweighted, weighted, and assignment).

Indeed, we have developed and analysed polynomial time algorithms for each of these problems earlier in the module.

A foundational question in complexity theory is whether there are **inherently difficult** problems which **cannot be solved in polynomial time**?

- There are many problems that nobody knows how to solve in polynomial time.
- This doesn't mean that it is impossible to solve them in polynomial time.
- It just means we haven't found a polynomial time algorithm ***yet***.

[1] https://en.wikipedia.org/wiki/Cobham's_thesis

9.1.2 The PRIME Problem

PRIME: given an integer k, is k ***prime***?

It was thought that PRIME did not have a polynomial time algorithm for a long time. However, Agrawak, Kayal, and Saxena[2] found one in 2002 that requires $O\Big((\log k)^{12}(\log(\log k)^{12})^{c}\Big)$ elementary operations, for some $c \geq 1$.

9.1.3 Goals of Complexity Theory

We want to ***build a theory*** that allows us to conclude that a problem **does not have a polynomial time algorithm**.

The best we can do ***today*** is develop theory that let's us conclude that a problem is **very unlikely** to have a polynomial time algorithm.

9.1.3.1 The Traveling Salesperson Problem (TSP)

We will repeatedly reference the ***traveling salesperson problem*** as an illustrative example.

Definition 9.1. A **Hamiltonian circuit** or **cycle** or **tour** of direted graph $G = (V, A)$ is a cycle that ***visits each node exactly once***

Definition 9.2. Traveling Salesperson Problem (TSP):[3]

- Given graph G with arc costs $c_{ij} \in \mathbf{Z}_+$.
- Find a Hamiltonian cycle of G with ***minimum cost***.

The TSP has many applications, especially within logistics/transportation, circuit design, and scheduling.

[2] https://en.wikipedia.org/wiki/AKS_primality_test,https://people.engr.tamu.edu/andreas-klappenecker/629/aks.pdf

[3] http://www.math.uwaterloo.ca/~bico/

9.1.4 Types of Problems

Given problem instance I defined by set of feasible solutions F and cost function $c : F \mapsto \mathbf{R}$.

Definition 9.3 (Decision Problem)**.**

- Given $L \in \mathbf{R}$, is there $X \in F$ such that $c(X) \leq L$?
- Algorithm A takes input I, L and returns **YES** or **NO**.
- **Search version:** A returns $X \in F$ with $c(X) \leq L$ if one exists.

Definition 9.4 (Optimization Problems in Search Version)**.**

- Algorithm A finds $X \in F$ minimizing $c(X)$.

Example 9.1 (Versions of TSP)**.** We can formally define ***two versions*** of the TSP:

- TSP as an Optimization Problem:
  - Given directed graph $G = (V, A)$ with costs $c_{ij} \in \mathbf{Z}_+$ for each arc $(i, j) \in A$:
  - ***Find a Hamiltonian cycle with minimum total cost.***
- TSP as a Decision Problem
  - Given directed graph G, arc costs c_{ij}, and **integer** L:
  - ***Does G contain a Hamiltonian cycle of total cost at most L?***

9.1.5 Comparison of Complexity of Decision and Optimization Problems

This prompts a reasonable question:

- Which is inherently more difficult?
- Optimization or decision problems?

The following theorem answers this fundamental question.

Theorem 9.1. *Any algorithm A for an* **optimization problem** *P solves the* ***decision version*** *P_d.*

Theorem 9.1 has a natural corollary, which establishes that tractable optimization problems correspond to tractable decision problems.

Corollary 9.1. *If the optimization problem P is* **polynomial time solvable**, *then the decision problem P_d is also* **polynomial time solvable**.

Proof. We have the following algorithm for solving the decision problem P_d given an algorithm A for solving optimization problem P:

1. Solve P using A. Let X^* be the minimizer of P with minimum value $c(X^*)$.
2. Compare the optimal value to L:
   - Return **YES** if $c(X^*) \leq L$;
   - Otherwise, return **NO**.

If A is polynomial time then this algorithm is also polynomial time. □

We aim to categorize ***decision problems*** based on their inherent difficulty.

We will start by comparing two complexity classes, **P** and **NP**.

9.1.6 The Class P

Definition 9.5. We denote by **P** the class of all decision problems which can be solved in ***polynomial time*** for **every instance** I[4].

We have seen several examples of problems belonging to **P**. Indeed, the decision versions of the ***minimum spanning tree, shortest path, max flow, matching,*** and ***assignment problems*** are all in **P**.

9.1.7 The Class NP

Definition 9.6. We say a problem **NP** belongs to the **complexity class P** or **nondeterministic polynomial**[5] if:

- Every instance I of $P \in$ **NP** with answer ***YES*** (called a **YES-instance**), has a **certificate** which verifies that the instance has answer **YES** in ***polynomial time***.

We are not interested in ***how to build the certificate***; just whether we can verify in polynomial time.

[4] https://en.wikipedia.org/wiki/P_(complexity)
[5] https://en.wikipedia.org/wiki/NP_(complexity)

Example 9.2 (Complexity Class of TSP)**.** The ***Traveling Salesperson Problem*** is in **NP**.

To see that TSP is in **NP**, we need to identify a certificate for an arbitrary YES-instance.

Consider the following proposed certificate for instance of TSP with $G = (V, A)$ and threshold L:

- Let $S = (v_1, v_2, \dots, v_n)$ be a sequence of nodes such that
  1. S contains all nodes of G ***exactly*** once.
  2. Pairs of consecutive nodes share an arc: $(v_i, v_{i+1}) \in A$. Moreover, $(v_n, v_1) \in A$.
  3. Total cost of these arcs is
  $$\sum_{i=1}^{n} c((v_i, v_{i+1})) \leq L;$$
  here, we use $v_{n+1} = v_1$.

This is a ***certificate***! If given S, we can that check that (1), (2), and (3) are satisfied in polynomial time.

- Indeed, S and (v_n, v_1) define the desired Hamiltonian cycle in this case.

9.1.8 Differences between P and NP

The class **P** contains **polynomial time solvable** problems:

- **P** contains decision problems to which a YES/NO answer can be given for any instance in polynomial time.

Other other hand, **NP** contains **polynomial time certifiable** problems:

- YES-instances admit a certificate which can be verified in polynomial time.

9.1.9 Turing Machines

Definition 9.7. A **turing machine (TM)**[6] is a mathematical model of computer as an **abstract machine** that:

- manipulates symbols from an infinite tape (can read and write),
- according to table of rules implementing a ***finite-state automaton***[7].

Everything a computer can do can be modeled by a TM.

[6] https://en.wikipedia.org/wiki/Turing_machine
[7] https://en.wikipedia.org/ewiki/Finite-state_machine

- **P** is the set of problems that can be **solved in polynomial time** by a TM.
- **NP** is the class of problems that can be **solved** (not just certified) in polynomial time by a **nondeterministic TM**[8].

9.1.10 The Relationship Between P and NP

It is known that **P** is a ***subset*** of **NP**. That is, every problem in **P** also belongs to **NP**.

We have the following theorem.

Theorem 9.2. $P \subseteq NP$

Proof. Suppose that problem Q in **P**, i.e., $Q \in$ **P**. This implies that there is a polynomial time algorithm A for Q.

The steps of A act as a certificate:

1. If A solves search version of Q, then the solution returned by A satisfies the conditions for **YES** response.
2. If A only returns **YES** or **NO** applying A and obtaining response **YES** is the certificate.

□

9.1.10.1 Does P = NP or P ≠ NP?

Conjecture 9.1. *It is widely conjectured that there is at least one problem in* **NP but not in P**, *i.e.,* **P ≠ NP**.

- Falsifying Conjecture 9.1 would prove that **P = NP**.
- Determinining if **P = NP** or **P = NP** is one of the seven ***Millenium Prize Problems***[9].

[8] https://en.wikipedia.org/wiki/Nondeterministic_Turing_machine
[9] https://www.claymath.org/millennium-problems/