Algorithmic Discrete Mathematics

Lecture Notes, Summer Term 2020

Andreas Paffenholz

January 26, 2021

TECHNISCHE UNIVERSITÄT DARMSTADT

Contents

1 Introduction

Contents

These lecture notes grew out of the course *Algorithmic Discrete Mathematics* at TU Darmstadt in 2011 and 2013 and were prepared by Andreas Paffenholz and Silke Horn, with several contributions by Marc Pfetsch. They were substantially rewritten in summer 2020. The course is part of the BSc curriculum at TU Darmstadt in the fourth semester.

The course is taught in English language. In these lecture notes we will give the *German translation [deutsche Übersetzung]* of the important notions we introduce.

1.1 What is it?

Algorithmic Discrete Mathematics is a field at the intersection of two mathematical fields with strong connection to computer science: *Discrete Mathematics* and *Algorithm Theory*.

1.1.1 Discrete Mathematics

Discrete Mathematics deals with mathematical structures that are fundamentally *discrete*, meant as a contrast to *continuous*. You have met the latter in your Calculus class, where you looked at the reals and where values can vary *smoothly*. You have exploited *smoothness* at various places, *e.g.*, in the *mean value theorem*. Discrete Mathematics, however, studies objects that are inherently finite or countable (*i.e.,* having then same cardinality as the integers), such as the integers, (relations on) finite sets, partially ordered sets, or graphs.

As a mathematical field in its own right and with its own name, *Discrete Mathematics* emerged only since about the 1950s with the development of computers. By their very

version of January 26, 2021

nature, computers can only store and process a finite amount of discrete data, which is stored as bits, in discrete steps. However, many questions now considered to be in this field appear in basically all other fields of mathematics and mostly have been around for much longer. You have certainly encountered already many such questions, *e.g.*, when counting permutations or combinations of an n-set within an m-set.

Indeed, many of the results now considered to be part of Discrete Mathematics originated in other fields like Algebra, Algebraic Geometry, Analysis, Number Theory, Group Theory, or Optimization, when people realized that some of the questions they studied had a natural discrete structure. Abstracting the discrete structure became useful in particular with the observation that similar structures appear in other places. Thus, analyzing the discrete problem immediately supports applications in otherwise quite different fields.

Nowadays discrete mathematics is a rather large field that again splits into various topics:

▷ *Combinatorics* studies configurations in which discrete structures can be arranged, and tries to describe, count or enumerate them. For instance, we can ask how many different k-element subsets we can obtain from a set of n distinct elements, where maybe the order of the elements matters.

▷ Studying the structure of finite or countable sets with a binary relation on its elements is the topic of *graph theory*. Graphs and its generalizations are one of the most common structures in mathematics and the natural world surrounding us. Examples you have probably met before are

- street networks, where our set are street crossings, and two crossings are related (connected), if there is a direct street between them,
- social networks, where the set is a group of people and two persons are related if they know each other,
- antenna interference, where the set are mobile or radio antennas, and we consider two such as related if their transmission interferes if they use the same frequency.

▷ *Coding* or *Information Theory* studies the the quantification of information and ways to store or transmit information efficiently. A commonly known example are bar codes or QR codes found on store items that allow a unique identification even if some amount of errors occur during transmission.

▷ Also some topics of *Number Theory*, the study of properties of the integers, are a topic of Discrete Mathematics, where one can study prime numbers, the partially ordered set of divisibility of numbers, or cryptography.

▷ *Polyhedral Theory* studies the structure of intersections of a finite set of half spaces (the set of solutions of a system of linear inequalities). *Linear Optimization* is about maximizing linear functionals on such sets.

In this course we will mostly be concerned with graph theory.

1.1.2 Algorithms

The theory of *algorithms* is a topic at the interplay of mathematics with computer science. It deals with the question of efficient computability of classes of objects or properties thereof, and the complexity of these computations. Similar to Discrete Mathematics, algorithmic questions became important with the rise of computers and their conquest of every-day life and scientific applications.

Essentially, an algorithm is a finite and deterministic list of instructions that explicitly solves a problem on a given set of instances. Algorithms are in the background of many processes we interact with every day, *e.g.*

▷ the sat nav in cars, which compute a shortest route between your current position and your destination,

▷ elevator controls, that decide which elevator should serve the requests in some order.

▷ processing of weather data to produce a forecast for the next days.

For all such problems we want to find good algorithms, where we need to make our interpretation of *good* precise. Clearly, algorithms should always return the *correct* results (which again needs to be mad more precise). But we could also be interested in *fast* or *efficient* algorithms, or algorithms that can deal with errors in the input or can adapt to online changes of the data. Thus we need to develop methods to measure the *complexity* or *running time* of algorithms, and to compare algorithms that solve the same problem.

1.2 What is in this course?

In a class with only two hours per week we can clearly only give you a small glimpse of what Discrete Mathematics and Algorithm Theory is about. But we hope to get you interested, so that you will continue with these topics in one or more of the many advanced courses in these fields at TU Darmstadt. In this semester, we will cover the following topics.

▷ We start with a short introduction to graph theory, introduce directed and undirected graphs, trees, and prove some properties.

▷ We discuss algorithms and all necessary tools to measure and compare them.

▷ As a first and fundamental application we look at the problem of searching and sorting data and look at some algorithms for this task.

▷ Our first algorithms on graphs will be concerned with the problem of traversing such a structure efficiently and construct so called spanning trees in graphs.

▷ We look at the problem of finding shortest paths in graphs, a problem you all know from your sat nav devices.

▷ Next will be flows in networks, which model for instance the throughput in gas or water pipeline networks, or can be used to optimize traffic flow in cities.

▷ Finally, we will also look at matchings or assignments, which *e.g.*, is the mathematical task underlying common scheduling problems like service schedules or the optimal use of machine capacity.

In particular for the last topics we can only discuss basic methods. These problems will be picked up in much more detail in some advanced courses.

1.3 What next?

In this course we can give you only a small introduction to both these fields. If you got interested and want to continue working in this areas there will be various courses at TU Darmstadt that pick up specific topics presented here. Here is a selection of courses regularly offered.

- ▷ *Discrete Mathematics* (Diskrete Mathematik) will concentrate on (non-algorithmic) topics from Enumerative Combinatorics, Arrangements and Generating Functions,
- ▷ *Introduction to Optimization* (Einführung in die Optimierung) is, besides some topics from continuous optimization, about linear programming and polyhedral theory,
- ▷ *Discrete Optimization* (Diskrete Optimierung) continues with polyhedral theory and discusses (mixed) integer linear programming and various algorithmic approaches for this,
- ▷ *Geometric Combinatorics* (Geometrische Kombinatorik) is about the geometric background of integer programming and discusses integer points in polyhedra, triangulations, and some connections to algebra,
- ▷ *Combinatorial Optimization* (Kombinatorische Optimierung) continues with graph algorithms.

But even if you intend to specialize in a completely different field during your studies you will notice that methods from discrete mathematics and algorithm theory appear throughout mathematics and computer science, and that it often saves time if one knows about these connections and can identify combinatorial structures in the problems one is interested in.

2 Graph Theory

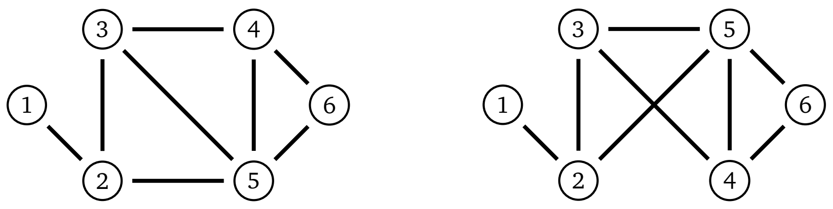
Contents

Graphs are the mathematical tool to model and study relations, usually *binary relations*, *i.e.,* relations between pairs of elements, on a finite or countable, not necessarily homogeneous, ground set. Sometimes the relations are also not assumed to be symmetric. This is a very basic notion that appears in many different settings, sometimes maybe a bit hidden. Among them are

(i) acquaintance between members of a social network,
(ii) hierarchies in society or companies,
(iii) phylogenetic trees modelling the evolution of species,
(iv) public transportation networks with stations as the set and direct connections between stations,
(v) assignments between students and classes they attend, or workers and the tasks they should work on.

2.1 Graphs

Here is a precise definition of a *graph* that we will use throughout this course. We will see variations of this definition, and we will always clearly state this if we use one. Note however, that textbooks sometimes choose one of the variations as their *basic* definition, and take ours as a variation.



(a) A planar drawing (b) A non-planar drawing

Figure 2.1: Drawings of the graph of Example 2.2

Definition 2.1. A graph $G = (V, E)$ is a pair of

- ▷ a finite set $V = V(G)$ of *nodes [Knoten]* or *vertices [Ecken]* and
- ▷ a set $E = E(G) \subseteq \binom{V}{2}$ of *edges [Kanten]*.

For a graph G we use $V(G)$ to refer to its nodes, and $E(G)$ for its edges. The terms *node* and *vertex* are used interchangeably. For a number $n \geq 1$ we will denote the set of the first n numbers by $[n] := \{1, \ldots, n\}$. Note that $\binom{V}{2}$ denotes the set of all two element subsets of the set V. *E.g.* we have $\binom{[3]}{2} = \{\{1,2\}, \{1,3\}, \{2,3\}\}$. We will sometimes use the following shorter notation for edges:

$$e = \{u, v\} = uv \ (= vu) .$$

An intuitive way to represent a graph is via a *drawing*. To *draw* a graph, we place a point into the plane $\mathbb{R}^2$ for each element of V (and maybe label it with the name of the element in V), and we connect two of these points with a line if the two nodes form an edge in the graph.

Example 2.2. Consider the following graph $G = (V, E)$ given by $V = [6] = \{1, 2, \ldots, 6\}$ and

$$E = \{\{1,2\}, \{2,3\}, \{2,5\}, \{3,4\}, \{3,5\}, \{4,5\}, \{4,6\}, \{5,6\}\} .$$

A drawing is shown in Figure 2.1(a). A drawing of a graph is obviously not unique. See Figure 2.1(b) for another drawing of the same graph. Note that lines representing edges in the graph may cross, this does not introduce a new node in the graph. It is usually not possible to draw a graph without crossings among the edges. Graphs that can be drawn such that no edges cross are called *planar [planar/eben]*. Most graphs do not have a planar drawing (*e.g.*, the graph K_5 of Figure 2.6 below)).

For larger graphs it is usually difficult to decide whether two drawings represent the same graph. You may want to check this for Figure 2.2.

The labels of the nodes given by the set V may or may not have a specific meaning. Often, labelling nodes in a graph with elements from $\mathbb{Z}_{\geq 0}$ is necessary to specify the

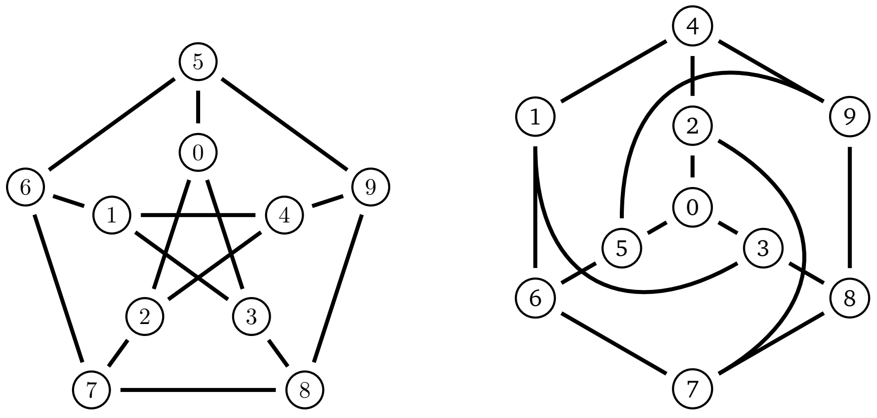


Figure 2.2: Two drawings of the same graph.

edge set, but the particular order of the nodes imposed by this is not important, and we consider graphs as *equal* if they only differ in this labelling. However, depending on the application, some nodes (or subsets of nodes) may have a specific interpretation in the given setting (*e.g.*, for the assignments in example (v) of the introduction above), and not all relabelings of the nodes give the same graph anymore.

We often identify a graph G with its set of nodes and edges. This is convenient for set operations involving (sets of) nodes and edges, where it allows us to write

$$G + v := (V \cup \{v\}, E) \qquad G - v := (V \setminus \{v\}, E \setminus \{e \mid |e \cap v| = 1\})$$
$$G + e := (V, E \cup \{e\}) \qquad G - e := (V, E \setminus \{e\}) .$$

Of course, the operations must yield valid graphs, *e.g.*, if we add an edge its two nodes should exist in the graph. This easily extends to sets of nodes and edges. See Figure 2.3 for examples.

We have seen above, that structures with binary relations appear in various places. Not all fit precisely into our definition of a graph, but share important properties. We want to use our tools to study also those. Of course, we always have to make sure that our results on graphs remain valid also with this variation. Let us introduce some common variants in an example.

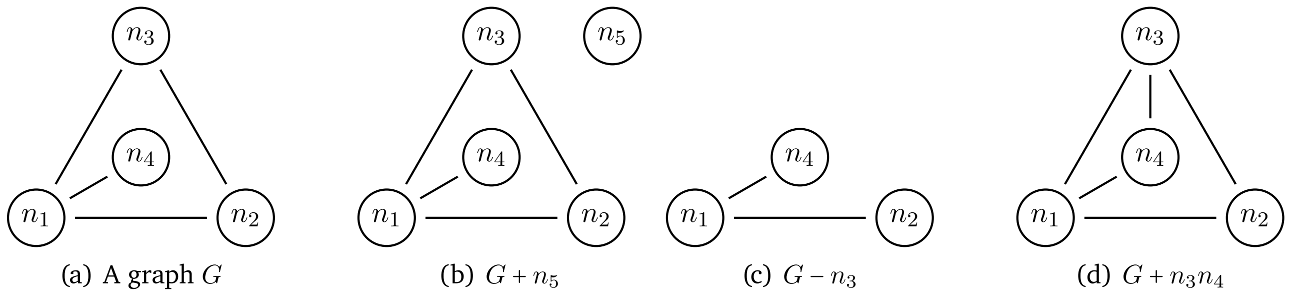


Figure 2.3: Adding and deleting nodes and edges

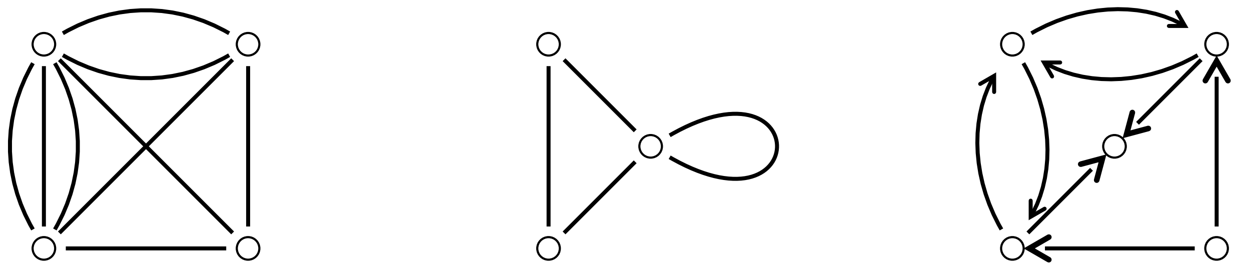

(a) An loopless graph with multiple edges (b) An undirected simple graph with a loop (c) A directed simple loopless graph

Figure 2.4: Examples of graphs

Example 2.3. *Street Maps* are a classical and widely known example that can be modelled with a graph. Here, the nodes are the street crossings, and the edges are the streets directly connecting two crossings. This example also shows the usefulness of many of the common variations of our definition of a graph one might consider:

(i) Sometimes streets have both ends at the same crossing, *i.e.,* they make a *loop*. We can include this in our definition if we let E be a subset of $\binom{V}{2} \cup V$. This leads to *graphs with loops*, while our graphs are *loopless*.

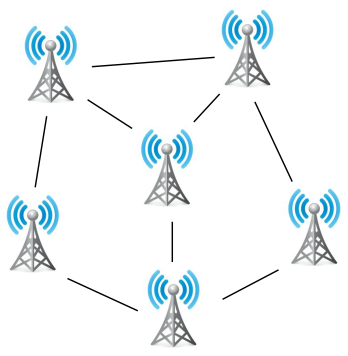
(ii) Larger streets sometimes have a small byway connecting the same two crossings. We can capture this if E is allowed to contain copies of the same edge (so E is a multiset). This leads to *multigraphs*. To stress that each edge can only appear at most once one sometimes calls our graphs *simple*.

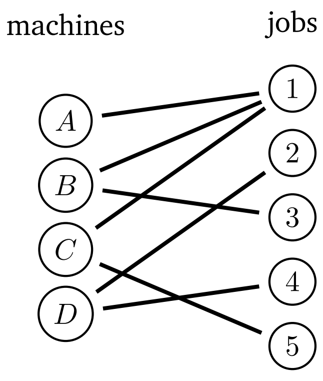
(iii) Streets have different lengths, or different travel times if we use them to get from one crossing to the other. We can put *weights* on the edges to keep track of this, that is, we additionally define a function $w : E \to \mathbb{R}$. Such graphs are called *weighted graphs*.

(iv) Some streets can only be travelled in one direction. Hence, our binary relation is not symmetric anymore, *i.e.,* the edges are not sets of nodes, but ordered pairs. This is such an important variation that we will define these *directed graphs* in a separate definition below (see Definition 2.4). In contrast, we have defined an *undirected* graph above.

There are many algorithmic questions one might be interested in. The most commonly known task is probably the *routing problem* that asks for a fast route from your position to some destination. In terms of graph theory, we would look for a *shortest path*in a weighted graph, which we discuss in more detail in Chapter 7. We could also ask how much traffic we can actually route through our street network, which, in mathematical terms asks for a *maximum flow*. We deal with this in Chapter 8.

Another interesting algorithmic problem on street networks is the *Chinese postman problem*. This ask for a route through the graph using all edges at least once and returning to the starting point. A route that uses each edge exactly once and returns to its start point is an *Eulerian Tour* in the graph. Such a tour need not exist, and it is a nice exercise to characterize graphs that do have such a tour. Graphs with this property


(a) Cell towers with interference



(b) An assignment problem

Figure 2.5: Graph problems

are called *Eulerian graphs*. The more famous (and considerably more difficult) *traveling salesperson problem* asks for a shortest route in a weighted graph, in which all nodes are connected by an edge (a *complete* graph), that visits each node exactly once.

To distinguish the graphs from our definition from those variations one may denote them as *simple, loopless, undirected graphs*. In this course we silently assume those specifications when talking about graphs and in contrast explicitly mention if we want to allow loops, multiple edges, or weights. See Figure 2.4 for two examples.

In later chapters we will mostly talk about directed graphs, so let us give a separate formal definition for those.

Definition 2.4. A *directed graph [gerichteter Graph]* (*digraph* for short) $G = (V, E)$ is a pair of

- ▷ a finite node set V and
- ▷ an a set $A \subseteq (V \times V) \setminus \{(v, v) \mid v \in V\}$ of *edges [Kanten]* or *arcs [Bögen]*.

Again, we might add *simple* and *loopless* to distinguish from other graphs. Since a digraph considers ordered pairs, arcs have a direction. We can draw them with arrows pointing from the first to the second node to indicate which node comes first in the pair. If (u, v) is such an arc, then u is its *tail* and v its *head*. Note that in directed graphs we may have two anti-parallel arcs between a pair u, v of nodes: One heading from u to v and one heading from v to u. See Figure 2.4(c) for an example.

Example 2.5. Here are some more examples where we can find graph structures.

(i) In **data or transportation networks** we have hubs or servers as nodes, connected by transportation links or cables, which are the edges in the graph. As with maps, we may have weights or directions on the edges in such a network.

A common question one would like to answer for such networks is again the shortest path problem. But also the question, whether the network is actually

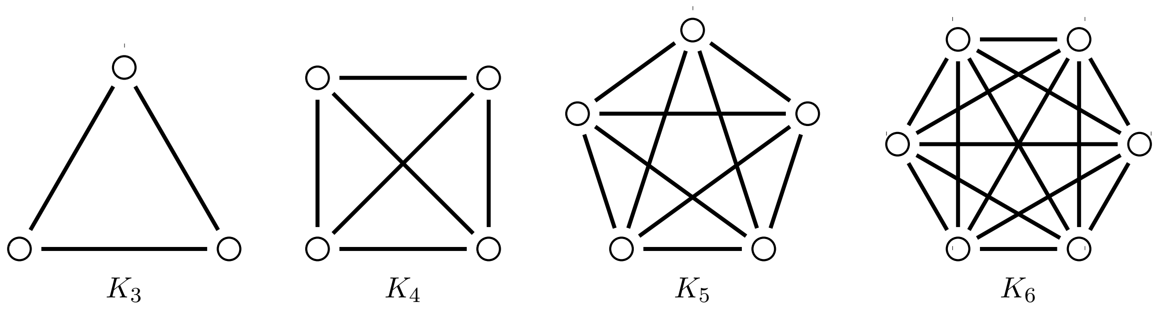


Figure 2.6: Complete graphs.

connected, *i.e.,* whether one can get from each node to any other node is important, or whether the network is robust, *i.e.,* how many links can fail before the network becomes disconnected. We will discuss the former already in the next section, while some ideas for the latter will have to wait until Chapter 8.

(ii) In **assignment problems** we usually have two types of objects for the nodes, *e.g.* jobs and machines, that are the nodes of the graph, and we connect a job and a machine with an (usually weighted) edge, if we could assign the job to the machine. The weight might be the cost, or the execution time of the job if done on this machine. See Figure 2.5(b).

Given such weights, the obvious question is the one for a cost-efficient assignment. In mathematical terms we ask for a min cost perfect (or maximum) matching in the graph. We solve this in Chapter 9.

(iii) Interference of signals of **mobile phone cell towers** can be modeled with the cell towers as nodes, and an edge between two cell towers if their signals interfere. See Figure 2.5(a).

We might be interested in finding an assignment of frequencies to the towers such that interfering towers broadcast on different frequencies. Mathematically, this can be modeled by a *coloring* of the graph. Other questions might be for a *node cover*, or we just try to minimize interference provided we have some kind of weight on the edges.

(iv) Prominent **data structures** are modelled with a graph-like structure, *e.g.* are *AVL trees*, *hash tables*, and *linked lists*.

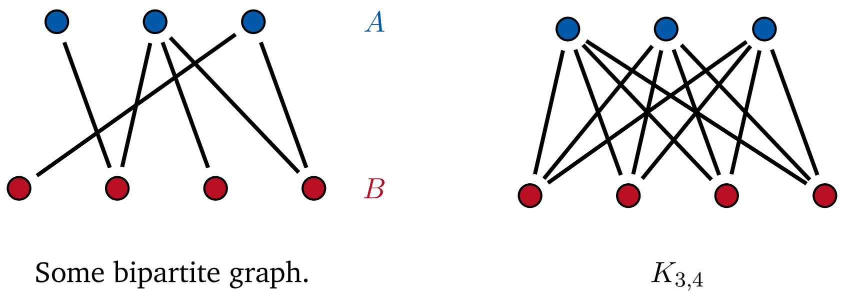
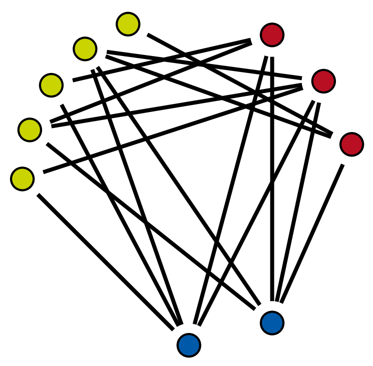


Figure 2.7: Bipartite graphs.


A 3-partite graph.

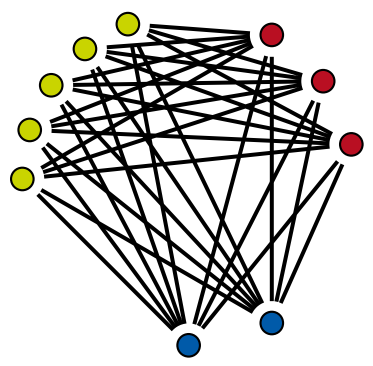

The complete 3-partite graph with color classes of size 2, 3, and 5.

Figure 2.8: 3-partite graphs

In many applications the graphs that may appear in the problem have additional common properties. Many of these subfamilies of graphs are so common that we have names for these classes. We introduce some of them now, and will see more later and in the exercises.

(i) A graph $G = (V, E)$ is a *complete graph [vollständiger Graph]* if $E = \binom{V}{2}$. We write K_n for this graph if $V = [n]$. See Figure 2.6 for some examples.

(ii) A graph is *bipartite [bipartit]* if there exists a partition of the nodes into two sets A and B such that $|e \cap A| = |e \cap B| = 1$ for all $e \in E$. Recall that A, $B \subseteq V$ form a partition of V if $A \cap B = \varnothing$ and $A \cup B = V$. The sets A, B are called the *color classes [Farbklassen]* of G. See Figure 2.7 for examples.

Note that A or B may be empty. We need this for graphs with one node being bipartite.

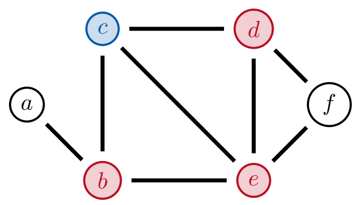
We say that G is *complete bipartite [vollständig bipartit]* if $E = \{\{u, v\} \mid u \in A, v \in B\}$. In this case we write $K_{n,m}$ if $A = [n], B = [m]$.

(iii) More generally, a graph $G = (V, E)$ is *k-partite [k-partit]* for some $k \geq 2$ is there is a partition of the nodes V into k subsets $V_1, \ldots, V_k$ such that no edge has both its endpoints in the same set of this partition.

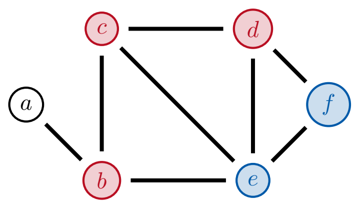
We say that G is *complete k-partite [vollständig k-partit]* if the edge set is the union over all $V_i \times V_j$ for all $i \neq j$. See Figure 2.8 for an example.

2.2 Neighborhood and Degree

For the rest of this chapter we only consider (simple, loopless) undirected graphs and the following definitions only work in this setting. We consider directed graphs in Chapter 7, and will modify the definitions to adapt to this case. However, as a first test that you really understand the next definitions and properties, it can be useful to check which of them need to be modified for directed graphs, or graphs with loops or multiple edges, and to come up with a sensible definition in this case.



(a) The neighborhood of c is $\{b, d, e\}$.



(b) The neighborhood of $\{e, f\}$ is $\{b, c, d\}$.

Figure 2.9: Neighborhoods.

Definition 2.6. Let $G = (V, E)$ be a graph, $u, v \in V$ and $e, f \in E$.

(i) The *neighborhood [Nachbarschaft]* of u is the set $\mathrm{N}(u) := \{v \mid \{u, v\} \in E\}$. The *degree [Grad]* of u is $\deg(u) := |\mathrm{N}(u)|$, and any node $v \in \mathrm{N}(u)$ is *adjacent [adjazent, benachbart]* to u.

(ii) The *(open) neighborhood [(offene) Nachbarschaft]* of $S \subseteq V$ is the set $\mathrm{N}(S) := \bigcup_{u \in S} \mathrm{N}(u) \setminus S$.

(iii) u is *incident [inzident]* to e if $u \in e$, and e is *incident [inzident]* to f if $e \cap f \neq \varnothing$.

Example 2.7. In the graph of Example 2.2 the neighborhood of c is $\mathrm{N}(c) = \{b, d, e\}$ (see Figure 2.9(a)) and the neighborhood of $S = \{e, f\}$ is $\mathrm{N}(S) = \{v, c, d\}$ (see Figure 2.9(b)). Further, the node e is *incident* to the edges $\{b, e\}$, $\{c, e\}$, $\{d, e\}$, and $\{e, f\}$, and the node f is *adjacent* to d and e.

The following theorem (or its Corollary 2.9) is also known by the name *handshake lemma*.

Theorem 2.8. *Let $G = (V, E)$ be a graph. Then*

$$\sum_{v \in V} \deg(v) = 2|E| .$$

Proof. The proof uses the simple, yet for proofs in enumerative combinatorics pretty common idea of *double counting*. For our proof we count the *number of incidences* between a vertex and an edge in the graph in two different ways.

On the one hand, the degree of a vertex v of the graph is exactly the number of such incidences that involve v. So the total number of incidences is the left hand side of the equation in the theorem.

On the other hand, each edge has precisely two vertices it is incident with. So the total number is also twice the number of edges, with is the right hand side of the equation. □

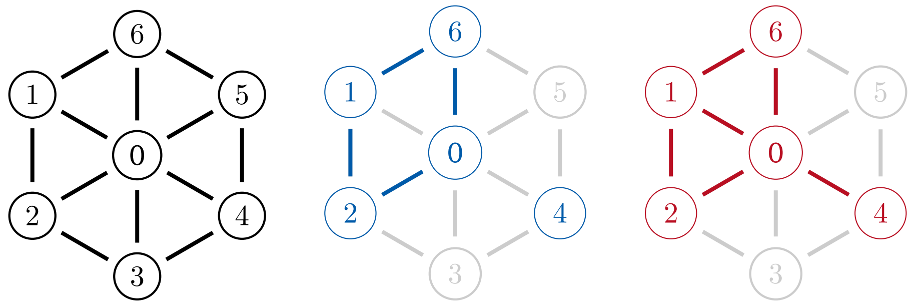


Figure 2.10: The graph in the middle is a subgraph of the left graph, but it is *not induced* since it does not contain the edges $\{0,1\},\{0,4\}$. The graph on the right is the induced subgraph on the vertex set $\{0,1,2,4,6\}$.

Corollary 2.9. *Let $G=(V,E)$ be a graph. The number $|\{v \mid \deg(v)$ is odd$\}|$ of nodes of odd degree is even.*

Proof. In the equation of Theorem 2.8, the right hand side is even. Hence, also the left hand side is even. □

2.3 Walks, Paths, Cycles, and Connectivity

We introduce *substructures* in a graph. This is an important notion also for many algorithmic tasks. To reduce the complexity of a problem, one often tries to find substructures in a given graph-like structure that still capture the properties one is interested in.

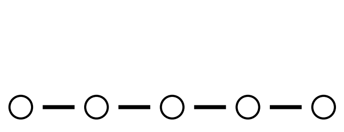
Definition 2.10. A graph $H=(W,F)$ is a *subgraph [Untergraph]* of $G=(V,E)$ if $W \subseteq V, F \subseteq E$. It is *induced [induziert]* if $F = E \cap \binom{W}{2}$.

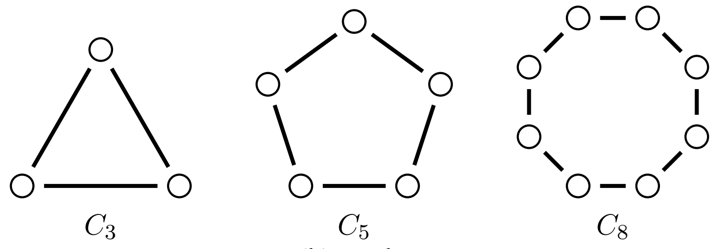
See Figure 2.10 for an example. Here are two more special classes of graphs that we will meet many times. Mostly, one uses them as (usually not induced) substructures in a graph, which we will make precise with Definition 2.11 below.

(i) A graph $P_n=(V,E)$ is a *path [Pfad]* of length $n \geq 1$, if $V=\{v_0,v_1,\ldots,v_{n-1},v_n\}$ and $E=\big\{\{v_{i-1},v_i\} \mid i=1,\ldots,n\big\}$. The two nodes v_0 and v_n have degree 1 and are called *terminal nodes [Endknoten]* or *endpoints*. All all other nodes have degree 2. It has n edges. See Figure 2.11(a).

(ii) A graph $C_n=(V,E)$ is a *cycle [Kreis]* of length $n \geq 3$, if $V=\{v_0,v_1,\ldots,v_{n-2},v_{n-1}\}$ and $E=\big\{\{v_{i-1},v_i\} \mid i=1,\ldots,n-1\big\} \cup \big\{\{v_{n-1},v_0\}\big\}$. It has n edges. See Figure 2.11(b) for examples. We say that C_n is *odd [ungerade]* if n is odd; C_n is *even [gerade]* if n is even.

Now let us rediscover those graphs as substructures of a general graph.


(a) The path P_4.



(b) Cycles.

Figure 2.11: Path and Cycles

Definition 2.11. Let $G = (V, E)$ be a graph. A *walk [Weg]* W in G is a sequence

$$v_0\, e_1\, v_1\, e_2\, \dots\, v_{k-1}\, e_k\, v_k$$

of nodes $v_i \in V$ and edges $e_j \in E$ such that $e_j = \{v_{j-1}, v_j\}$ for all $j = 1, \dots, k$. The nodes v_0, v_k are the *terminal nodes [Endknoten]* of W. The *length [Länge]* of a walk is the number of edges.

There are various important specializations of a walk:

(i) A *closed walk [geschlossener Weg]* is a walk such that $v_0 = v_k$.

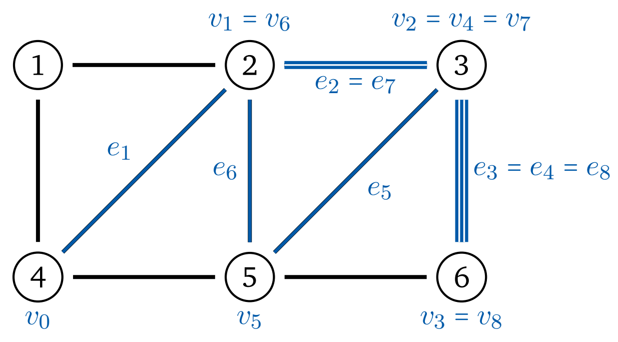
(ii) A *path [Pfad]* in G is a walk such that all v_i are distinct.

(iii) A *trail [kantendisjunkter Weg]* in G is a walk such that all e_j are distinct.

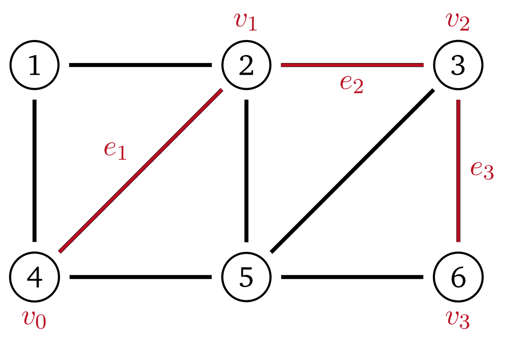
(iv) A *cycle [Kreis]* is a closed walk such that $k \geq 3$ and $v_i \neq v_j$ for all $i \neq j$ with $\{i, j\} \neq \{0, k\}$. A cycle is *odd/even [ungerade/gerade]* if its length is.

We usually omit the edges in the notation of a walk, as they are uniquely defined by the nodes. See Figure 2.12 for an illustration of the difference between a path and a walk.

The next theorem below will show that walks in a graph define an equivalence relation



(a) A *walk* from 4 to 6 in a simple graph. Some edges and nodes are used several times.



(b) A *path* from 4 to 6. All nodes v_i are distinct.

Figure 2.12: Path vs. walk.

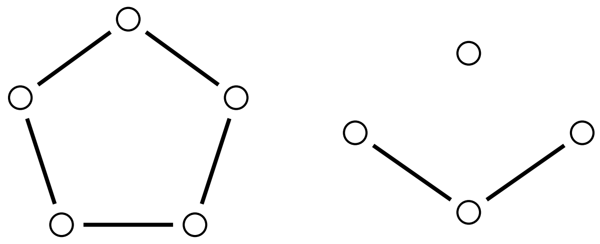


Figure 2.13: A graph with three connected components.

on the nodes of the graph. This will lead us to one of the first algorithmic tasks on graphs that we have identified above, the *connectivity* of a graph. But first, let us recall the definition of an equivalence relation.

Reminder. Let X be a set and $R \subseteq X \times X$. Then R is an *equivalence relation [Äquivalenzrelation]* on X if it satisfies

(i) *symmetry*: $(x, y) \in R \implies (y, x) \in R$ for all $x, y \in X$,

(ii) *reflexivity*: $(x, x) \in R$ for all $x \in X$, and

(iii) *transitivity*: $(x, y), (y, z) \in R \implies (x, z) \in R$ for all $x, y, z \in X$.

The *equivalence class [Äquivalenzklasse]* of $x \in X$ is $\{y \mid (x, y) \in R\}$. The equivalence classes form a partition of X.

Theorem 2.12. *Let $G = (V, E)$ be a graph. The relation*

$$x \sim y \ :\iff \quad \text{there is a walk from } x \text{ to } y$$

is an equivalence relation on V.

Proof. Given a walk in an undirected graph we can also traverse it in the opposite direction, so the relation is symmetric. A walk can just consist of the single vertex x, so it is reflexive. Finally, adding edges to the end of a walk results in a walk, so the relation is also transitive. ☐

Definition 2.13. Let $G = (V, E)$ be a graph. The equivalence classes in the relation of Theorem 2.12 above are the *connected components [Zusammenhangskomponenten]* of G. We say that G is *connected [zusammenhängend]* if it has only one component. We define $c(G)$ as the number of connected components of G.

See Figure 2.13 for an example. Any walk of length 0, *i.e.,* containing only a single vertex and no edge, is also a path. The following theorem shows more generally, that in any walk of length at least one with distinct terminal nodes we can pick a subsequence of the nodes that is a path between the two terminal nodes. Hence, we can replace *walk* by *path* in the definition of connectedness.

Proposition 2.14. *Let $G = (V, E)$ be a graph.*

(i) Any walk with distinct terminal nodes v, w contains a path between v and w.

(ii) Any closed walk in which at least one edge occurs an odd number of times contains a cycle.

Proof. (i) Let W be a walk between v and w. If W is a path, we are done. Otherwise there is some vertex x that occurs twice along W. We delete everything between the two occurrences and one of the x to get a new walk W'. We then repeat this procedure until we obtain a path.

(ii) Let $W = v_0 v_1 \ldots v_n$ be a closed walk and let i be the smallest index such that $\{v_i, v_{i+1}\}$ is an edge that is contained an odd number of times in W. We set $u := v_i$ and $v := v_{i+1}$. If $\{u, v\}$ is contained exactly once, delete $\{u, v\}$ from W to obtain a walk W' from v to u. By (i) W' contains a path p from v to u. Adding $\{u, v\}$ to p gives a cycle.

Now assume that $\{u, v\}$ is contained in W more than once, say $k \geq 3$ times. We consider the subwalk W' of W that starts with the two nodes u, v and ends with the next occurrence of u in W. Then W' contains $\{u, v\}$ either exactly once or exactly twice. In the first case, we found a closed walk W' that contains $\{u, v\}$ exactly once and can proceed as above. In the latter case, we remove W' from W and obtain a closed walk W'' that contains the edge $k - 2$ times. We can iterate this until we get a closed walk that contains $\{u, v\}$ exactly once. □

Note that Part (ii) of the previous proposition is wrong without the existence of an edge that occurs an odd number of times: Consider the graph $G = (V, E)$ with $V = \{v_0, v_1\}$, $E = \{e_1 = \{v_0, v_1\}\}$, and the closed walk $v_0\, e_1\, v_1\, e_1\, v_0$, which does not contain a cycle.

Proposition 2.15. *Any graph that contains an odd closed walk contains an odd cycle.*

Proof. Assume there is a graph G contradicting the statement. It must contain an odd closed walk. Let $W : v_0 v_1 \ldots v_k v_0$ be an odd closed walk in G of minimal length. We claim that W is already a cycle. Otherwise there are $0 \leq i < j \leq k$ such that $v_i = v_j$. We obtain two new closed walks $W_1 : v_0 v_1 \ldots v_{i-1} v_i v_{j+1} \ldots v_k v_0$ and $W_2 : v_i v_{i+1} \ldots v_{j-1} v_i$. The sum of their lengths is the length of W, so one has odd length. But both are shorter than W, contradicting our choice of an odd closed walk. □

We can use cycles in graphs for a first and important new characterization of bipartite graphs. We will see later in Chapter 5 that we can use this to efficiently detect such graphs.

Theorem 2.16. *Let $G = (V, E)$ be a graph. Then G is bipartite if and only if every cycle in G is even.*

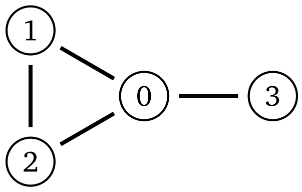


Figure 2.14: The graph of Example 2.18.

Proof. If G is bipartite with color classes A and B, then any path in G alternates between A and B. Hence every cycle has even length.

Conversely, let G be a graph that does not contain an odd cycle. We may without loss of generality assume that G is connected. Now fix some vertex $v \in V$. Define $A \subset V$ to be the set of all vertices w such that there is a path of odd length in G from v to w. Moreover, we define $B = V \setminus A$. (Then B contains all vertices with even distance from v.) It now suffices to show that $\binom{A}{2} \cap E, \binom{B}{2} \cap E = \varnothing$, *i.e.,* there are no edges between any vertices of A, respectively B.

Assume that there is an edge $\{a, a'\}$ with $a, a' \in A$. Then there are a $[v, a]$-path p_1 and a $[v, a']$-path p_2 of odd length. We can then construct a closed walk $C = (v, p_1 a, a', p_2, v)$ of odd length, which contains some odd cycle. This contradicts our assumption.

The proof that there is no edge $\{b, b'\}$ with $b, b' \in B$ is similar. □

2.4 Graph Isomorphism

We have already observed in the very beginning, that the labels we assign to vertices are in many cases only important to identify them in the edge list or in a drawing. Yet, in many cases this *labeling* of the graph is, though necessary for the purpose of of explicit computations, somewhat arbitrary. Also, two graphs may originate from different applications that induce some natural labeling, but they are structurally equivalent. We can identify such graphs via an *isomorphism*.

Definition 2.17. Let $G = (V, E), H = (W, F)$ be graphs. Then G is *isomorphic [isomorph]* to H if there is an *isomorphism [Isomorphismus]* between the two graphs, *i.e.,* a bijective map $\phi : V \to W$ such that

$$\{u, v\} \in E \quad \Leftrightarrow \quad \{\phi(u), \phi(v)\} \in F.$$

An isomorphism $\psi : V \to V$ is called an *automorphism [Automorphismus]* of G. The set of all automorphisms of a graph G forms a group, the *automorphism group [Automorphismengruppe]* $\mathrm{Aut}(G)$ of G.

Example 2.18. The function $\phi : V(G) \to V(G)$ that sends

$$0 \longmapsto 0 \qquad 1 \longmapsto 2 \qquad 2 \longmapsto 1 \qquad 3 \longmapsto 3$$

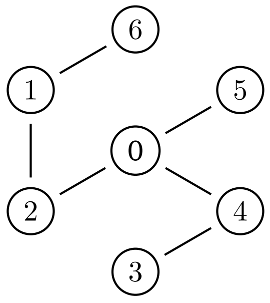


Figure 2.15: The graph $T = (\{0,\ldots,6\},\{\{0,2\}\{0,4\},\{0,5\},\{1,2\},\{1,6\},\{3,4\}\})$ is a tree. Its leaves are $3,5$ and 6.

is an automorphism of the graph in Figure 2.14. In fact, it is the only non-trivial automorphism of G and the automorphism group of G is $\mathrm{Aut}(G) = \{\mathrm{id}_G, \phi\} \cong \mathbb{Z}/2\mathbb{Z}$.

However, observe that in some cases the labelling of the nodes is important, and nodes with different labels may play different roles in our application. Think, *e.g.*, of our assignment problems, where we have bipartite graphs, and one class of nodes may represent machines, the other jobs that need to be assigned. We may reorder jobs and machines, but we cannot exchange them. Still, there might be a subgroup of the automorphism group acting on the nodes, that only permutes the labels of jobs and machines among themselves. You may want to work out a proper definition.

2.5 Trees

In this section we introduce another subfamily of graphs, the so called *trees*. These appear as an important substructure of (connected) graphs, that we will encounter in most of the following chapters. Here, we prove some of their basic properties that we need. Trees are also the basis of many hierarchical data structures widely used in computer science.

2.5.1 Forests and Trees

In plain words, *trees* are connected graphs without cycles. Equivalently, trees are graphs in which any two vertices are connected by *exactly* one path. We will see several further equivalent definitions.

Definition 2.19. A graph F is *acyclic [kreisfrei]* if it does not contain a cycle. In this case F is called a *forest [Wald]*. A *tree [Baum]* T is a connected forest. Nodes of degree 1 are called *leaves [Blätter]*.

See Figure 2.15 for an example.

Proposition 2.20. *A tree with at least two nodes has at least two leaves.*

Proof. Let T be a tree. Choose a path $v_1 \dots v_k$ in T of maximal length. Assume $\deg(v_1) > 1$. Then there is $w \neq v_2$ adjacent to v_1. Since T is acyclic, $w \notin \{v_1, \dots, v_k\}$. Hence, $wv_1 \dots v_k$ is a longer path – a contradiction. Thus, $\deg(v_1) = 1$. Similarly, we show that $\deg(v_k) = 1$. □

Remark 2.21. Let $T = (V, E)$ be a tree with $|V| \geq 2$ nodes. Then we can choose a node v of T such that $T - v$ is still a tree (and $T - v$ is a tree if and only if v is a leaf).

Theorem 2.22. *Let $G = (V, E)$ be a graph with $n \geq 1$ nodes. Then the following are equivalent:*

(i) *G is a tree.*

(ii) *G is connected and has $n - 1$ edges.*

(iii) *G is acyclic and has $n - 1$ edges.*

(iv) *G is* minimally connected *[minimal zusammenhängend], i.e., G is connected and $G - e$ is disconnected for all $e \in E$.*

(v) *G is* maximally acyclic *[maximal kreisfrei], i.e., G is acyclic and $G + e$ contains a cycle for all $e \in \binom{V}{2} \setminus E$.*

(vi) *For all $u, v \in V$ there is a unique path from u to v.*

Proof. We show $(i) \Rightarrow (ii) \Rightarrow (iii) \Rightarrow (iv) \Rightarrow (v) \Rightarrow (vi) \Rightarrow (i)$:

$(i) \Rightarrow (ii)$: We show that G has $n - 1$ edges by induction over n. If $n = 1$ then G has one vertex and 0 edges. Now let G be a tree with $n > 1$ vertices and let v be a leaf of G. We remove v and its incident edge to obtain a tree G' with $n - 1$ vertices. By induction G' has $n - 2$ edges and hence G has $n - 1$ edges.

$(ii) \Rightarrow (iii)$: Again, we show by induction over n that G has no cycle. If $n = 1$ then G has no edges and only one vertex. This graph is obviously acyclic. Now assume that G has $n > 1$ vertices and $n - 1$ edges and is connected. By connectedness every vertex has degree at least one. If $\deg(v) \geq 2$ for every vertex v then $|E| = \frac{1}{2} \sum \deg(v) \geq n$. Hence G has a leaf v. If we remove v and its incident edge, we obtain a connected graph G' with $n - 1$ vertices and $n - 2$ edges. By induction G' does not contain any cycle. Thus, G is also acyclic.

$(iii) \Rightarrow (iv)$: We first show (by induction over n) that G is connected. If $n = 1$, then G is obviously connected. Now let $n > 1$. Since G has no cycle, there is a longest path in G, whose endpoints are leaves. By removing a leaf v and its incident edge, we obtain a connected graph G' with $n - 1$ vertices and $n - 2$ edges. By induction G' is connected and by construction G is so, too.

If there is no cycle, then there is at most one path between any two vertices. So removing any edge will make the graph disconnected. Hence G is minimally connected.

$(iv) \Rightarrow (v)$: We first show that G is acyclic. Suppose on the contrary that there is a cycle. Then we can remove any edge from this cycle without affecting connectedness. Hence G is acyclic.

Now fix an edge $e = uv \notin E$. Since G is connected there is a path p from u to v in G. Hence $G + e$ contains a cycle and thus G is maximally acyclic.

$(v) \Rightarrow (vi)$: Let $u, v \in V$. Since G is maximally acyclic there is a $[u, v]$-path. Otherwise we could add the edge uv without creating a cycle. On the other hand if there were two $[u, v]$-paths p_1, p_2 then the concatenation of p_1 and the reverse of p_2 forms a closed walk with start point u. By Proposition 2.14(ii) this contains a cycle. Hence there is a unique $[u, v]$-path.

$(vi) \Rightarrow (i)$: G is connected since for any $u, v \in V$ there is a $[u, v]$-path in G. Moreover, G is acyclic since this path is unique. (The line of argument is as in the previous part.) □

The following is a simple consequence from the previous theorem (and the proof is left as an exercise).

Corollary 2.23.

(i) A forest F on n vertices has $m := n - c(F)$ edges.

(ii) A graph G on n vertices contains a cycle if and only if $m > n - c(G)$.

2.5.2 Spanning Trees

Trees and forests very prominently appear as a substructure in graphs, that already allows to determine many global properties of the graph. This is important, as these trees have usually less edges and a simpler structure, thereby reducing the algorithmic complexity of computations.

Definition 2.24. Let G be a connected graph and T a subgraph of G. Then T is a *spanning tree [aufspannender Baum]* of G if T is a tree and $V(T) = V(G)$.

See Figure 2.16 for an example. With this definition we can obtain a first important result where the tree substructure already tells us a global property of the whole graph.

Proposition 2.25. *A graph is connected if and only if it contains a spanning tree.*

Proof. By Theorem 2.22, any graph with a spanning tree must be connected.

We prove the converse by induction on the number of edges. If G has no edge, then it has only one node (as it is connected), and thus is already a tree. Now assume that G has at least one edge. If G is not a tree, then it contains a cycle C (by Theorem 2.22 (ii) and (iii)). Pick any edge e in C. Then $G' := G - e$ is connected and has one edge less

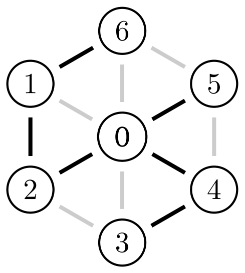


Figure 2.16: A spanning tree of the graph in Figure 2.10.

than G. By induction, G' has a spanning tree T. But G and G' have the same nodes, so T is also spanning in G. $\square$

3 Algorithms and Data Structures

Contents

This chapter introduces *algorithms*, their design, and their analysis. We will also look at options to encode graphs in a data structure and the algorithmic implications of these choices. We pick up *Complexity* of algorithms and their usual classification in complexity classes like P and NP later in Chapter 10.

Essentially, an *algorithm* is the formalization of a deterministic procedure to solve a computational problem. You have met algorithms in your daily life, *e.g.*, with recipes that you find in your favourite cookbook, assembly instructions for your new piece of Swedish furniture, or connection queries for public transport. But you have also met many algorithms already in previous semesters, maybe without explicitly referring to them with this name. Here are two examples.

▷ From your Calculus lecture, you know a formal procedure to produce the derivative of a polynomial.

▷ In Linear Algebra you have learned how to compute the inverse of a non-singular matrix M using elementary row operations simultaneously on M and the unit matrix.

In each case the given procedure applies to any element (an *instance*) from a feasible set of inputs, which is the set of univariate polynomials $\mathbb{R}[x]$ in the first, and the set of all non-singular square matrices in $\mathbb{R}^{n\times n}$ in the second case, in the same way and produces the correct result in a finite number of steps. Moreover, whoever uses this procedure need not have any additional knowledge on the input or the mathematical background.

Algorithms don't just appear in mathematics or computer science. We use them in many places throughout our daily life. Some tasks where we refer to algorithms for the answer are, *e.g.*, the following:

▷ Finding the shortest route in a network. This may be for car travel, or for a short connection with public transport.

- ▷ Scheduling lectures at the university with minimal overlap.
- ▷ Searching large data sets, for instance using one of the internet search engines.
- ▷ Recognizing faces in photos of your favorite social network.
- ▷ Routing trucks for delivery of parcels.
- ▷ Loading and unloading cargo ships, so that the load is balanced, and containers needed first are at the top.

There are many more examples of algorithms which we use each day, and many of the problems solved by them also have an inherent discrete structure. You will probably discover some if you just consider what you use and rely on during a normal day.

3.1 The Largest Subarray Sum Problem

Let us do a simple example in more detail that will illustrate the topics we discuss in this chapter. We consider the following question.

> Given a finite sequence $A = (a_1, a_2, \dots, a_n)$ of rational numbers, compute the largest possible sum of a consecutive subset of the sequence.

This is the *Largest Subarray Sum Problem*. For example, the largest sum in the sequence

$$A = (3,\ 5,\ -9,\ 6,\ 7,\ -8,\ 4,\ 5\ -6,\ 5)$$

is 9, given as the sum $14 = 6 + 7 - 8 + 4 + 5$. How can we solve this for an arbitrary finite sequence A of numbers, where we don't anymore see the right choice immediately by looking at it?

The first option one might think of is the *brute force* solution: Compute the sum of all consecutive sub-sequences of A and pick the maximum of those. Algorithm 3.1 shows this approach in *pseudo-code*. It only returns the sum, not the indices of the first and last element in the subarray. As a simple exercise you can try to rewrite the algorithm so that it includes the two indices in the return.

Here, and in all following examples of algorithms, we use this notation in *pseudo-code*. This is a, not completely standardized, convention to write down the steps of an algorithm in a formal language abstracting from the structure of a particular programming language. With pseudo-code we try to pick up and formalize the important steps disregarding all technicalities that might be necessary for particular languages (like, *e.g.*, declaring variables, obtaining memory from the system) and the exact form of the keywords (*e.g.*, the loop keyword `for` in some languages is `foreach`). However, it should capture enough theoretical information that someone with sufficient knowledge of a programming language, but little knowledge of the mathematical background, should be able to implement the method.

Back to our problem. Is the proposed method a *good* procedure to solve this problem? More precisely, from a mathematical point of view we should at least care about the first two of the following questions, and if we want to apply our methods in our research or other applications, we should also consider the third.

(i) Is the output correct for all arrays that we allow as input?

(ii) Does it always terminate in finite time?

Algorithm 3.1: Brute force solution for the Largest Subarray Sum Problem

```
Input  : A finite array $A = (a_1, a_2, \dots, a_n)$ of real numbers.
Output : The maximal sum of a subarray in $A$.
max_sum ← 0                // the empty subarray has sum 0
for $i \leftarrow 1, \dots, n$ do
    cur_sum ← 0
    for $j \leftarrow i, \dots, n$ do
        cur_sum ← cur_sum + $a_j$
        if cur_sum > max_sum then
            max_sum ← cur_sum
return max_sum
```

(iii) Is our procedure efficient?

The positive answer to the first two is pretty obvious for our given problem and the proposed algorithm. We should look at the last question. To answer this we first need a way to measure *efficiency*. Only then can we compare our approach to other solutions.

As a simple measure we could use the number of additions and comparisons we have to make when following our procedure. All those happen inside the inner loop over the variable j, and within each round of this loop we do one addition and one comparison. The number of rounds is determined by the variable i of the outer loop. So in total we do

$$f_1(n) := 2\,(n + (n-1) + (n-2) + \cdots + 1) = n(n+1) = n^2 + n$$

additions and comparisons. For large arrays, when you need such a procedure and cannot decide by just looking at the array, you need essentially n^2 steps for an array of length n. To decide whether this is efficient we must find out whether we can solve the same problem with less computations.

Algorithm 3.2 shows another approach to this problem, known as *Kadane's algorithm*. Though shorter, you find it most likely less obvious to realize that the algorithm returns the correct result. This is a common observation. Being more efficient in a computation usually requires us to exploit more of the structure hidden in the problem, and we first have to analyze, extract, and prove the properties we want to use.

Algorithm 3.2: Kadane's Algorithm

```
Input  : A finite array $A = (a_1, a_2, \dots, a_n)$ of real numbers.
Output : The maximal sum of a subarray in $A$.
max_sum ← 0                // the empty subarray has sum 0
cur_sum ← 0
for $i \leftarrow 1, \dots, n$ do
    cur_sum ← max(0,cur_sum+$a_i$)
    max_sum ← max(cur_sum, max_sum)
return max_sum
```

To check correctness in our case observe that, whenever you have a subarray S of your array A in which an *initial* subarray T (*i.e.,* T starts at the same index as S) whose sum is zero or less, then the total sum of S can only increase if you remove T from it.

Let us again measure the efficiency by counting the number of comparisons and additions. Note that line 4 contains one each, and line 5 contains a comparison. But this time, we only have one loop with n rounds, so

$$f_2(n) = 3n\,.$$

For all $n \geq 3$ these are less steps than in the first algorithm, and for large n this difference is quite significant! So we should prefer this procedure over the first. Can we be even more efficient? Clearly, at some point we have to look at each element in the array, which involves at least n comparisons. So we can never be faster than kn for some $k \geq 1$. So we may be able to reduce the constant 3 in the product $3n$ to something smaller, but we cannot get rid of the factor of n. Hence, the number of computations will always be *linear in* n.

3.2 Algorithms

Now let us start to formalize our observations so far. The first thing we need to make precise is the notion of a *problem*, that we then try to solve with our algorithm.

Definition 3.1. A (algorithmic) *problem [Problem]* is a pair $\Lambda = (\mathcal{I}, (S_I)_{I\in\mathcal{I}})$ of a set $\mathcal{I}$ of *instances [Instanzen]* and a family $(S_I)_{I\in\mathcal{I}}$ of sets of *solutions [Lösung]*.

We also have a measure that associates an *input size [Eingabegröße]* $|I|$ to each instance I.

Example 3.2. In the *Largest Subarray Sum Problem* the instances are all arrays of finite length with elements in the reals, and the solutions are the reals. The input size of the instance, in the measure of efficiency we used, is the length of the array.

Note that the set S_I for some instance I may be larger than the true solutions, we only require that it contains all of them (otherwise we may not be able to write the set down). Most often we also just mention the instances when we talk about a problem, and the sets of solutions are implied.

Here are some more examples of *problems* that are discussed in graph theory.

(i) The problem of finding a spanning tree in a graph is given by the set of instances $\mathcal{I} = \{\text{graphs on } n \text{ vertices}\}$ and solutions $S_I = \{\text{spanning trees in graph } I\}$. The input size of I is the number $n + n$ of n nodes and m edges in the graph.

(ii) A *travelling salesperson tour* (a *TSP tour*) on the complete graph K_n with a weight function $w : \binom{[n]}{2} \to \mathbb{R}_{\geq 0}$ (think of distances between the nodes, which may represent cities) is a cycle in K_n that contains all vertices and minimizes the sum of the weights of the edges in the path.

The *Travelling Salesperson Problem (TSP)* is given by the set of instances $\mathcal{I}$ =

$\{$weight functions w on $K_n,\ n \geq 1\}$ and solutions $S_I = \{$TSP tours in graph $I\}$. The input size of I is the number n of nodes.

(iii) The problem of finding an *Eulerian path* (*i.e.,* a path in the graph using all edges) in a graph is given by $\mathcal{I} = \{$graphs$\}$ and $S_I = \{$Eulerian path in graph $I\}$.

(iv) The problem of deciding whether a given graph is connected is given by $\mathcal{I} = \{$graphs$\}$ and $S_I \subseteq \{$'yes', 'no'$\}$.

The last problem in this list is somewhat different from the others. Here, the set of solutions just states whether some condition holds or not. Such problems are called *decision problems [Entscheidungsprobleme]*.

Definition 3.3. An *algorithm [Algorithmus]* $\mathcal{A}$ is a finite and deterministic list of instructions that perform operations on some data. We say that $\mathcal{A}$ *solves* Λ *[$\mathcal{A}$ löst Λ]* if, for any given instance I of Λ, it terminates after a finite number of steps and returns a solution $\mathcal{A}(I) \in S_I$.

So an algorithm $\mathcal{A}$ must terminate after *finitely many steps* with *correct output* on *all* allowed input.

3.3 Running Time of Algorithms

In our first example, computing the largest subarray sum, we estimated the complexity of our algorithm by counting the number of comparisons and additions. Clearly, this is not a suitable method in general, as not all computations will involve additions or comparisons of real numbers. We need to come up with a more comprehensive method to measure the number of steps (the *running time*) our algorithm takes on a given input.

Thus, we need a formalization of the *computational model [Berechnungsmodell]*. There are various *computational models* around serving different purposes (depending on the type and size of the input relevant for the running time, the intended application etc). We choose here a rather simple such model. It will, nevertheless, be sufficient for our purposes and capture the essential properties of all our algorithms.

Our measure for the steps in a algorithm will be the *RAM model [RAM-Modell]* ("Random Access Machine"), often also called *unit cost model* or *standard model*. In this model, we assume that all *elementary operations* require constant time ("*unit cost*"), regardless of the involved operators, and that we have an unbounded linear storage (memory) segmented into *registers*, which can hold one basic item of data. We can read and write to a register in constant time ("*random access*").

Here, elementary operations are the usual unary and binary operations like additions, subtractions, multiplications, divisions, comparisons, negations, assignments, etc. Basic data items can *e.g.* be integers, rationals, pointers to other registers etc.

This is clearly not a completely accurate picture for computations or modern computers (you might find multiplying larger numbers more time consuming). Yet, it is sufficient as long as

- ▷ the size of the problem is small enough to fit in memory, and
- ▷ integers and other numbers are all about the same size and small enough to fit into standard data types allowed by computers, also in intermediate steps.

Such an abstraction of a computational model allows us to free the analysis of an algorithm from a particular computing device and to extract the relevant parts of the problem.

The running time of an algorithm quantifies the amount of time taken by the algorithm as a function of the input size.

Definition 3.4. The (worst case) *running time [Laufzeit]* or *time complexity [Zeitkomplexität]* of an algorithm $\mathcal{A}$ is the function

$$f_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$$
$$f_{\mathcal{A}}(k) := \max \left\{ \begin{array}{c} \text{number of elementary operations} \\ \text{performed by } \mathcal{A} \text{ on input } I \end{array} \;\middle|\; I \in \mathcal{I},\ \text{size}(I) \leq k \right\}$$

In practice, the running time is commonly expressed using the $\mathcal{O}$-notation that we discuss in the next section. Note that we could also analyze the *space complexity [Speicherkomplexität]* of an algorithm, *i.e.,* an upper bound on the amount of memory required, (so, in our model the largest index of a register read or written to). In this course we will only consider *time complexity*.

3.4 Asymptotic Growth

One of the questions we raised above for algorithms is about their *efficiency*. For this, we need a measure that captures the essential characteristics of an algorithm and levels out effects that only occur for some specific (*e.g.* small) input. One way to achieve this is to consider the behavior for *large* input sizes and compare running times of various algorithms on such input.

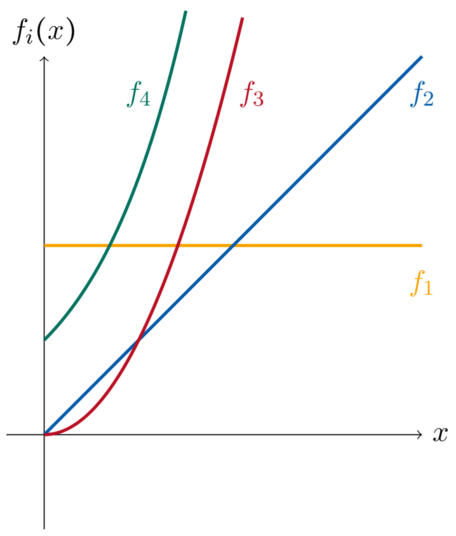
For a comparison of running time functions as defined in Definition 3.4 we want to get rid of parts specific for our formulation, or parts that contribute little compared to the rest. The usual approach to this are simple functions that give lower and upper estimates. This is maybe best understood with some examples.

Example. Assume that a, b are constants, and consider the functions

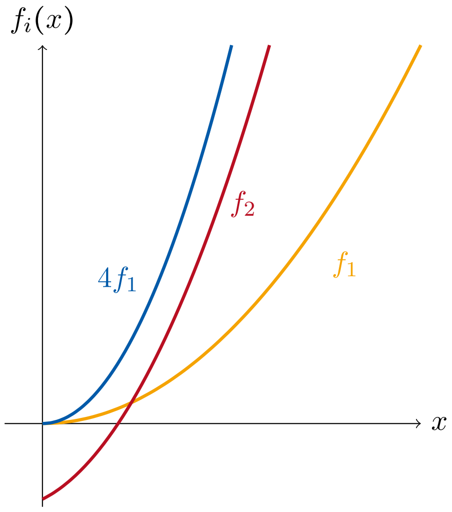
$$f_1(n) = a\,, \qquad f_2(n) = n\,, \qquad f_3(n) = n^2\,, \qquad f_4(n) = b^n\,.$$

We have plotted these functions in Figure 3.1(a). From the graph we can make the following observations:

- ▷ f_4 “grows faster than” f_3
- ▷ f_3 “grows faster than” f_2
- ▷ f_2 “grows faster than” f_1



(a) Different growth behaviour.



(b) Similar growth behaviour.

Figure 3.1: Growth of functions.

Example. Consider the functions

$$f_1(n) := n^2, \qquad f_2(n) := \frac{5}{2}n^2 + 5n - 20\,.$$

Both functions tend to ∞ for $n \to \infty$, see Figure 3.1(b). We can bound one function using the other from above and below by

$$n^2 \le \frac{5}{2}n^2 + 5n - 20 \le 4n^2$$

for $n \ge 3$. In this sense these functions exhibit a “similar growth behavior”.

Guided by these examples we will now try to distinguish classes of functions by their asymptotic growth behavior. This is formalized in the following definition.

Definition 3.5. Let $g : \mathbb{N} \to \mathbb{R}_{\ge 0}$. The set of functions *asymptotically bounded above* by g is

$$\mathcal{O}(g) := \{f : \mathbb{N} \to \mathbb{R}_{\ge 0} \mid \exists c > 0, n_0 \in \mathbb{N} : f(n) \le cg(n)\ \forall n \ge n_0\}.$$

We sometimes write $f = \mathcal{O}(g)$ instead of $f \in \mathcal{O}(g)$.

Example 3.6. Let p be a polynomial of degree d. Then $p \in \mathcal{O}(n^a)$ for any $a \ge d$. In fact, if $p = \sum_{i=0}^{d} \alpha_i x^i$ for $\alpha_i \in \mathbb{R}$, $\alpha_d \ne 0$, we can use $c = \sum_{i=0}^{d} |\alpha_i|$.

Similarly, we can bound functions from below.

Definition 3.7. Let $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Then we define

$$\Omega(g) := \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0, n_0 \in \mathbb{N} : f(n) \geq cg(n)\ \forall n \geq n_0\},$$

the set of all functions *asymptotically bounded below* by g.

With these definitions we have

$$f \in \mathcal{O}(g) \iff g \in \Omega(f).$$

Furthermore, both $\mathcal{O}$ and Ω are transitive:

$$\begin{aligned} f \in \mathcal{O}(g), g \in \mathcal{O}(h) &\implies f \in \mathcal{O}(h) \\ f \in \Omega(g), g \in \Omega(h) &\implies f \in \Omega(h) \end{aligned}$$

The next definition combines $\mathcal{O}$ and Ω.

Definition 3.8. Let $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Then we define

$$\Theta(g) := \{f : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c_1, c_2 > 0, n_0 \in \mathbb{N} : c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0\},$$

i.e.,

$$f \in \Theta(g) \iff f \in \mathcal{O}(g) \text{ and } f \in \Omega(g).$$

Example. Let p be a polynomial of degree d. Then
- ▷ $p \in \Omega(n^b)$ for any $b \leq d$, and
- ▷ $p \in \Theta(n^d)$ but not in $\Theta(n^b)$ for any $b < d$.

Definition 3.9. Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. We say that g *grows faster [wächst schneller]* than f, in symbols $g \succ f$ if for all $\varepsilon > 0$ there exits $n_\varepsilon \in \mathbb{N}$ such that

$$f(n) \leq \varepsilon g(n)$$

for all $n \geq n_\varepsilon$. If g has only finitely many zeroes then we can rewrite this to

$$g \succ f \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

We define

$$o(g) := \{f \mid f \prec g\} \qquad \text{and} \qquad \omega(g) := \{f \mid g \prec f\}.$$

Example. Here are some more examples for the previous definitions.

(i) $n^a \prec n^b$ for $a < b$

(ii) $\log n \prec n^\delta$ for all $\delta > 0$. So for a constant $c > 0$ we have

$$c \prec \log n \prec n^\varepsilon \prec c^n \prec n^n.$$

(iii) $log_b n = \frac{\log_a n}{\log_a b}$ for $a, b > 1$, so

$$\mathcal{O}(\log n) = \mathcal{O}(\log_2 n) = \mathcal{O}(\log(3n)).$$

(iv) $f \in o(1)$ means that $f(n) \to 0$ for $n \to \infty$, while $f \in \mathcal{O}(1)$ implies that f is bounded.

The various symbols $\mathcal{O}, \Omega, \Theta, o, \omega$ that we have introduced in this section are called *Landau symbols [Landau-Symbole]*.

The following proposition collects some more properties. The proof is left as an exercise.

Proposition 3.10. *Let $f_i \in \mathcal{O}(g_i)$ for $i = 1, 2$. Then*

(i) $f_1 f_2 \in \mathcal{O}(g_1 g_2)$.

(ii) $f_1 + f_2 \in \mathcal{O}(\max(g_1, g_2))$

Similar implications are true for the other Landau symbols. The second property implies that the set $\mathcal{O}(g)$ is a convex cone *in the space of functions.*

3.5 The Longest Ascending Subsequence Problem

Let us look at one more algorithmic task where we can apply our new tools. We consider the *Longest Ascending Subsequence Problem*, where we are given a finite sequence of (real or integer) numbers, and we want to find the length of the longest subsequence of (not necessarily consecutive) elements such that each following element in the subsequence is strictly increasing.

Example. Consider the sequence

$$A = (6,\ 2,\ 7,\ 3,\ 5,\ 1,\ 4,\ 5,\ 10,\ 6,\ 4,\ 7).$$

The longest ascending subsequence is $S = (2, 3, 4, 5, 6, 7)$, and its length is 6.

The instances in this problem are the finite sequences, the sets of solutions are the finite *ascending* sequences, and the input size of an instance is the length of the sequence.

The first idea to compute the longest ascending subsequence in a sequence A might be to test all subsequences of A, and, if they are ascending, compare its length with the currently best candidate. If it is longer, then replace that candidate and continue. The pseudo-code is in Algorithm 3.3.

Algorithm 3.3: Brute force solution for the Longest Ascending Subsequence Problem

```
Input  : A finite sequence $A = (a_1, a_2, \dots, a_n)$ of real numbers.
Output : The length of the longest ascending subsequence in $A$.
1 n ← 0  // the empty subsequence is ascending and its length
    is 0
2 foreach subset $T$ of $A$ do
3     if $T$ is ascending then
4         m ← length(T)
5         if $m > n$ then
6             n ← m
7 return $n$
```

To compute its true running time we would need to be more precise in the description of the algorithm. In particular, we would need to work out how to generate all subsets of a set. However, we can already give a *lower* bound for its running time. In the algorithm we have a loop over all subsets of a set with n elements. The number of subsets with k elements in a set with n elements is the binomial coefficient $\binom{n}{k}$, so in a set of n elements we can find

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n$$

different subsets. So even if we can generate subsets in constant time (which we cannot), we need at least 2^n steps. This fast increase of possible substructures (here: the subsequences) with linearly growing size of the structure itself (here: the sequence A) is sometimes referred to as *combinatorial explosion*, a problem one quite commonly faces for combinatorial structures or in combinatorial algorithms. A linear (or polynomial) increase in the input size may lead to an exponential growth of the possible structures. This makes it computationally impossible to enumerate all structures, or loop over all of them.

In our case, this implies that the running time is in $\Omega(2^n)$. As a small exercise you can work out that it is also in $\mathcal{O}(2^n)$. So the running time of this algorithm is *exponential*. Even for the quite moderate example above it computes $2^{12} = 4096$ subsets. Can we do better?

If you reconsider how you found the longest ascending subsequence in the example at the beginning (you tried, right?), then you probably did not consider all subsets but tried to add elements to an increasing chain you already found. This can be formalized and leads to a much faster algorithm. In particular, if $S = (s_1, s_2, \dots, s_k)$ is an ascending subsequence, then also $S' = (s_1, s_2, \dots, s_{k-1})$ is such a sequence whose last element has a smaller index than the last element of S.

We can turn this observation into an algorithm. For this, let $l(j)$ be the length of a longest ascending subsequence in A whose last element is a_j. We can add the element a_{k+1} to a sequence that ends with some a_j, $j \leq k$ if and only if $a_j < a_{k+1}$. So if

Algorithm 3.4: Longest Ascending Subsequence Problem

```
Input  : A finite sequence $A = (a_1, a_2, \ldots, a_n)$ of real numbers.
Output : The length of the longest ascending subsequence in $A$.
1 for $i \leftarrow 1, \ldots, n$ do
2     l(i) ← 0
3 for $i \leftarrow 1, \ldots, n$ do
4     for $j \leftarrow 1, \ldots, i-1$ do
5         if $a_j < a_i$ and l(j)+1 > l(i) then
6             l(i) ← l(j)+1
7 return max(l(j) | $1 \leq j \leq n$)
```

$J = \{j \leq k | a_j < a_{k+1}\}$, then

$$l(k+1) = \max_{j \in J}(l(j)+1) .$$

The length of the longest ascending subsequence is finally the maximum over all $l(j)$. See Algorithm 3.4 for the pseudo-code. We can work out the running time.

▷ The initialization in lines 1–2 has $2n$ assignments.

▷ The inner body of the loop in lines 4 and 5 has at most 2 additions, 3 comparisons and one Boolean evaluation.

▷ We run this inner part $\binom{n}{2}$ times, do 2 assignments to fill the variables i and j in each round, and one subtraction and assignment to determine the upper bound of the inner loop.

Summing up, the running time function is

$$f(n) := 2n + \binom{n}{2}((2+1+1)+(2+3+1)) = 5n^2 - 3n .$$

Using Example 3.6 we deduce that $f \in \mathcal{O}(n^2)$, so our new algorithm is quadratic in the input size. This is a considerable improvement to the previous version.

With this approach we have discovered the quite important algorithmic principle of *dynamic programming*. In our example, we have realized that given an ascending subsequence S in A, *any* subsequence of S is again an ascending subsequence of A. More formally, *any substructure of a valid structure is again valid*. Whenever we encounter a problem whose solutions satisfy this condition we can use the same bottom up strategy to obtain an algorithm for the problem. We start generating small solutions (ascending subsequences) and successively extend them until we reach the required optimal solution (the longest such). This simple idea is indeed a quite powerful method to produce efficient algorithms. It as many applications. We will encounter another example in Chapter 7, where we compute shortest paths in graphs with the *Bellman-Ford-Algorithm*.

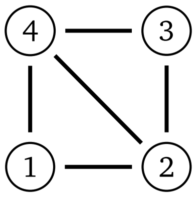


Figure 3.2: An example of a graph.

3.6 Representations of Graphs

The representation of graphs by a drawing, which we have mostly used so far, is quite intuitive for humans, but is clearly not suitable for a computer. Here we need more abstract ways to encode the information contained in a graph. It is not difficult to come up with ideas for this. Yet, it is also quick to realize that our choice may impact the running time. For instance, we can completely encode a graph by writing down its list of edges, *e.g.*

$$(1,2),(1,3),(1,4),(2,4),(3,4)\,.$$

This is the graph shown in Figure 3.2. This list contains all necessary information in a format we can store in a computer. But if we need the number of vertices in the graph we have to scan through all edges and count the number of different vertex labels. This is an operation linear in the number of edges. We could easily improve the data format by adding one extra bit of information, namely the number of vertices, to avoid this and get the number in constant time.

In the following sections we will, in the analysis of our algorithms, usually not explicitly specify how the graph is given, but assume that it is stored in a way so that we can extract all relevant information quickly. Nevertheless, let us briefly discuss common choices. We make the following assumptions on our input graph G:

- ▷ The number of nodes of G is n, and the number of edges is m.
- ▷ The nodes of G are numbered $1,\ldots,n$, and the edges are given as pairs of nodes.

As our example for the different data structures we use the graph given in Figure 3.2

(i) An *edge list [Kantenliste]* is a single list or array that contains edges as pairs of nodes. In our example we get:

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

An edge list uses $\mathcal{O}(m)$ memory (number of registers, not bits!). We need time $\mathcal{O}(m)$ to check if $\{i,j\}\in E$.

(ii) The *adjacency matrix [Adjazenzmatrix]* of G is the matrix $A = A(G) \in \{0,1\}^{n\times n}$

with

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

In our example we get:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

A is a $0/1$-Matrix with 0 on the diagonal. Storing an adjacency matrix needs $\mathcal{O}(n^2)$ registers, but we can check whether $\{i,j\} \in E$ in time $\mathcal{O}(1)$ (constant time).

We can easily adapt this structure to store further information. Examples for this can include

a) the number of edges for multigraphs, or

b) weights (lengths) of edges, or

c) loops in the graph. In this case we would put nonzero entries on the diagonal to count the number of loops.

(iii) An *adjacency list [Adjazenzliste]* is a list of the nodes of G, where each entry points to a list that contains all adjacent nodes (with pointers to them). In our example we get:

$$\begin{array}{ccc} \boxed{1} & \longrightarrow & \boxed{2}\boxed{4} \\ \boxed{2} & \longrightarrow & \boxed{1}\boxed{3}\boxed{4} \\ \boxed{3} & \longrightarrow & \boxed{2}\boxed{4} \\ \boxed{4} & \longrightarrow & \boxed{1}\boxed{2}\boxed{3} \end{array}$$

Formally an adjacency list is a function

$$\begin{aligned} \mathrm{Adj} : [n] &\longrightarrow 2^V \\ i &\longmapsto \mathrm{Adj}(i) \end{aligned}$$

with $j \in \mathrm{Adj}(i) \iff \{i,j\} \in E$. We need $\mathcal{O}(n+m)$ registers in memory to store an adjacency list, and we need time $\mathcal{O}(n)$ to check whether $\{i,j\} \in E$.

(iv) The node-edge *incidence matrix [Inzidenzmatrix]* of G is the matrix $D = D(G) \in \{0,1\}^{n \times m}$ with

$$d_{ij} = \begin{cases} 1 & \text{if } i \in j, \\ 0 & \text{otherwise,} \end{cases}$$

i.e., the rows of D correspond to the vertices of G, the columns to the edges. In our example we get:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} \{1,2\} & \{1,4\} & \{2,3\} & \{2,4\} & \{3,4\} \end{array} \\ \left(\begin{array}{ccccc} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{array}\right) \end{array}$$

Its memory usage is $\mathcal{O}(nm)$.

We can use $D(G)$ for a simple proof of Theorem 2.8. Namely, summing the matrix $D(G)$ row-wise gives $\sum \deg(i)$, while summing column-wise gives $2|E|$. The matrix $M := DD^T \in \mathbb{R}^{n\times n}$ is symmetric and

$$m_{ij} = \begin{cases} \deg(v) & \text{if } i = j, \\ a_{ij} & \text{otherwise,} \end{cases}$$

i.e.,
$$M = \begin{pmatrix} \deg(v_1) & & \\ & \ddots & \\ & & \deg(v_n) \end{pmatrix} + A\,.$$

(v) An *incidence list [Inzidenzliste]* for G consists of two lists: a list of nodes and a list of edges, where

▷ for every node we have a list of incident edges, and

▷ for every edge we have pointers to its two nodes.

We need $\mathcal{O}(n + m)$ registers to store it and time $\mathcal{O}(m)$ to check if $\{i, j\} \in E$.

We should again observe that storing a graph with these data structures depends on the labeling of the nodes. This labeling may or may not be relevant for our given problem, and we may need tools to deal with this, *e.g.* if we want to check whether two graph structures are the same.

The input size of a graph can be defined as the input size for one of the representations introduced above. For an encoding via an incidence matrix or an adjacency list a graph $G = (V, E)$ has input size $\langle G\rangle = \langle \text{IncidenceMatrix}(G)\rangle \in \mathcal{O}(|V| + |E|)$. Sometimes we implicitly make a non-degeneracy assumption on the graphs we consider. Namely, we assume that $n \in \mathcal{O}(m)$, in other words, we assume that there are not too many isolated nodes.

For directed graphs you have to make a couple of adjustments to these structures. For each node we have two different types of edges, the incoming and outgoing edges. We split adjacency lists into two parts accordingly. For adjacency matrices we use $a_{ij} = |\{\text{arcs from } i \text{ to } j\}|$ for the entries. You should work this out in more detail for yourself.

4 Sorting and Searching

Contents

We consider two fundamental algorithmic problems, sorting a list of numbers and searching for a particular element in an array. Both tasks are common routines in many other algorithms, not only in graph theory.

Sorting is the task to sort all elements of a finite list with respect to a given (total) order. So the input is a finite list together with a function for the pairwise comparison of two elements. The function should return whether the two elements are *equal*, or whether the first is *smaller* or *larger* than the second in the given order. In this chapter we always use integers as elements and sort with respect to the usual comparison via $<$ (*less than*), so we obtain a list in *ascending* order. That is, given a list $A = (a_0, a_1, \dots, a_n)$ of integers, we want to return a list $B = (b_0, b_1, \dots, b_n)$ with the same elements, but such that $b_i \leq b_j$ for all $i < j$.

However, the generalization to other comparison functions for more complex data elements should be straightforward. A common example for a different comparison function is the lexicographic order used to order strings, *e.g.* to produce a list sorted by last names of the data set of all registered participants of this course. Another common example of a more complex order function is the order on dates or times, where *e.g.*, *May 1, 2020* is considered smaller (*earlier*) than *May 2, 2020*.

The related problem of *Searching* is the task to check whether a given element is in a list. Again, we discuss this for lists of integers, so given a list $A = (a_0, a_1, \dots, a_n)$ and some integer a we want to decide whether there is some index j with $a = a_j$. We will see that this is simple for sorted lists, in which case we can even obtain the position we should add the element, if it is not in the list.

4.1 Simple Sorting Algorithms

How can we efficiently modify a list of integers to bring it into ascending order? A first attempt for this task might be to improve locally via pairwise exchange of adjacent elements that are in the wrong order. So formally, if we are given a list

$$A = (a_1, \ldots, a_j, a_{j+1}, \ldots, a_n)$$

where we find some $1 \le j \le n-1$ such that $a_j > a_{j+1}$, then we replace A with the list

$$A = (a_1, \ldots, a_{j+1}, a_j, \ldots, a_n).$$

We can repeat this as long as we find such pairs.

Are we done if we cannot find further adjacent pairs in the wrong order? Assume that all adjacent pairs in A are in the correct order, but there are $1 \le k < l \le n$ with $l - k \ge 2$ such that $a_k > a_l$. Then we can consider the chain

$$a_k \le a_{k+1} \le a_{k+2} \le \ldots \le a_l < a_k ,$$

which, if we look at the first and last element, is clearly a contradiction. So our list is not only *locally*, but also *globally* sorted. Thus, we have a sorted list if there are no further *adjacent* elements in the wrong order.

Do we always eventually obtain a sorted list when we randomly swap adjacent elements that are in the wrong order? Let us look at the following set.

$$S := \{ (i, j) \mid i < j \text{ and } a_i > a_j \} .$$

Note that this contains *all* pairs of indices of elements that appear in the wrong order. This is clearly a finite set, and the list is sorted if and only if $S = \varnothing$. Now observe that if $(j, j+1) \in S$, then swapping a_j and a_{j+1} removes this pair from S. Any $(k, j) \in S$ with $k < j$ is replaced by $(k, j+1)$, any $(k, j+1) \in S$ for $k < j$ with (k, j), and similarly for $k > j+1$. Hence, after we swap a_j and a_{j+1} the set S has exactly one element less. As S is finite, our method must finish after finitely many steps.

You can of course also swap any two, not necessarily adjacent, elements in the wrong order. This will also produce a sorted list in a finite number of steps, but controlling the size of S is more difficult, as this may drop by more then one in each step. You are invited to work out the details.

For a proper algorithm we, however, need a deterministic procedure, so we need to fix an order in which we check for pairs that we have to swap. The most common way here is the following. We start at the first index $i = 1$ and successively check the pairs $(i, i+1)$ for increasing i. We repeat this as long as we find a pair that we need to swap in our traversal.

We can determine how often we need to repeat, and this will also show that with increasing repetitions we can each time stop earlier. Namely, if we run through the array once, always exchanging an element with its successor if necessary, then the element in the last position in the array will certainly always also be the largest element in the whole array. So this element will never be swapped again.

If we repeat the traversal, also the second last element will be in the correct position, and so on. Hence, after we have traversed the array $n-1$ times (if all but one element are in the correct position, then also the last one is!), our array is sorted, and in the j^{th} iteration, $1 \le j \le n-1$, we can stop already at position $n-j$, as we know that the

Algorithm 4.1: BubbleSort

```
Input  : list $A = (a_1, \dots, a_n)$
Output : $A$ sorted in ascending order
for $i \leftarrow 1, \dots, n-1$ do
    for $j \leftarrow 1, \dots, n-i$ do
        if $a_j > a_{j+1}$ then
            $b \leftarrow a_j$
            $a_j \leftarrow a_{j+1}$
            $a_{j+1} \leftarrow b$
```

5	4	7	1	3	6	2
4	5	1	3	6	2	7
4	1	3	5	2	6	7
4	1	3	2	5	6	7
1	3	2	4	5	6	7
1	2	3	4	5	6	7
1	2	3	4	5	6	7

Table 4.1: BubbleSort

remaining elements are already sorted.

This method is known as *BubbleSort*, which is shown in Algorithm 4.1. Table 4.1 shows an example for the list $A = (5, 4, 7, 1, 3, 6, 2)$. The next theorem summarizes this and provides the running time.

Theorem 4.1. BubbleSort(A) *sorts the list in time* $\Theta(n^2)$.

Proof. Correctness already follows from our considerations above. So we only have to discuss the running time. For this observe that we run the body of the inner loop $b(n) := \frac{1}{2}(n-1)(n-2) = \frac{1}{2}(n^2-3n+2)$ times, and each time we do at least one comparison and at most additionally three assignments.

Assigning and incrementing the loop variables i and j needs $n-1$ assignments, $n-1$ comparisons, and $n-1$ additions for i, and $n-i$ assignments, $n-i$ comparisons and $n-i$ additions for j (where we need the comparisons to check whether the loop variable has hit the boundary, and if not, we add one and assign to the same variable). So executing the loops needs

$$l(n) \;:=\; 3(n-1) \;+\; 3\frac{1}{2}(n-1)(n-2) \;=\; \frac{3}{2}(n^2-n)$$

elementary operations.

Hence, the running time $f(n)$ is between $l(n) + b(n)$ and $l(n) + 4b(n)$. Using our definitions from the previous chapter we see that $f \in \Theta(n^2)$. □

We observe that the essential property of this algorithm is the fact that in each iteration one more element is in the correct position. We can improve slightly if we just do the swap that puts the next element in correct position, dropping all other swaps. More precisely, in iteration i we assume that all elements in positions $j > i$ are in the correct place, pick the largest element among those with index at most i and swap this into position i. This approach is called *SelectionSort*. Its pseudo-code is in Algorithm 4.2. Though it does less swaps, it still traverses the list n times, so we don't improve on the asymptotic running time. See Table 4.2 for an example.

Algorithm 4.2: SelectionSort

```
Input  : array $A = (a_1, \dots, a_n)$
Output : $A$ sorted in ascending order
1 for $i \leftarrow n, \dots, 2$ do
2     $k \leftarrow i$
3     for $j \leftarrow 1, \dots, i-1$ do
4         if $a_j > a_k$ then
5             $k \leftarrow j$
6     $b \leftarrow a_k$
7     $a_k \leftarrow a_i$
8     $a_i \leftarrow b$
```

5	4	7	1	3	6	2
5	4	2	1	3	6	7
5	4	2	1	3	6	7
3	4	2	1	5	6	7
3	1	2	4	5	6	7
2	1	3	4	5	6	7
1	2	3	4	5	6	7

Table 4.2: SelectionSort

Theorem 4.2. SelectionSort(A) *sorts the list in time* $\Theta(n^2)$.

Proof. The algorithm sorts a list in finitely many steps. We only need to check the running time.

If the outer loop is in iteration i, then the inner loop does $i-1$ assignments $i-1$ comparisons and $i-1$ additions for the loop variable j, and inside the loop one comparison and at most one further assignment. So the running time $l(i)$ of the inner loop is is bounded from below and above by $4(i-1) \le l(i) \le 5(i-1)$.

In the outer loop we have again $n-1$ assignments, $n-1$ comparisons and $n-1$ additions for the loop variable i, four assignments and $l(i)$ steps for the inner loop. So in total we need

$$f(n) := 3(n-1) + 4(n-1) + l(2) + \cdots + l(n)$$

steps, which is in $\Theta(n^2)$. □

By now you have probably realized that it is a futile effort to accurately count the number of elementary operations in the body of a loop or on the loop variable, as long as this number is bounded by some constant per iteration. The final running time is given by the number of times we have to run the body of the loop. In SelectionSort, we have two nested loops that both run essentially n times both at least and in the worst case, so the total running time is in $\Theta(n^2)$.

Another variation of this idea leads to *InsertionSort*. Here we maintain a sorted sub-list S with only some of the elements of A, and insert further elements at the correct *relative* position in this sub-list, by comparing them one by one with the elements in the sorted sub-list S, starting from the last. Here, the elements of the sub-list S are always in the correct order, but not necessarily already at the correct position in A.

This is the algorithm many people intuitively choose when picking up a deck of cards to their hand and sort them on the way. You pick one card after the other and always insert it at the correct position among the cards already in your hand.

This approach is made explicit by the pseudo-code in **??**. A simple example is shown

```
Algorithm: InsertionSort
  Input  : array $A = (a_1, \dots, a_n)$
  Output : $A$ sorted in ascending order
for $i \leftarrow 2, \dots, n$ do
    $b \leftarrow a_i$
    $k \leftarrow i$
    while $k \geq 2$ and $a_{k-1} > b$ do
        $a_k \leftarrow a_{k-1}$
        $k \leftarrow k - 1$
    $a_k \leftarrow b$
```

5	4	7	1	3	6	2
4	5	7	1	3	6	2
4	5	7	1	3	6	2
1	4	5	7	3	6	2
1	3	4	5	7	6	2
1	3	4	5	6	7	2
1	2	3	4	5	6	7

Table 4.3: InsertionSort

in Table 4.3. We do not improve on the running time, but for some applications, where the elements we have to sort are only revealed to us one at a time, this is a useful algorithm to choose.

Theorem 4.3. InsertionSort(A) *sorts the list in time* $\mathcal{O}(n^2)$.

Proof. Correctness of the algorithm is immediate from the definition.

As in the previous two examples we have two nested loops that do a constant number of elementary operations per iteration.

The out loop runs $n - 1$ times, while the inner loop runs at most i times in the i^{th} iteration of the outer. On an already sorted list the inner loop does not execute, so we don't get the same lower bound as before, and the total running time is $\mathcal{O}(n^2)$. □

4.2 MergeSort

By now we have seen three different algorithms that sort a list in a number of steps proportional to n^2. Can we do this with less steps? We look at one further example, the *MergeSort Algorithm*. This algorithm uses a *Divide-and-Conquer* approach to sort a list that is based on the following two observations.

(i) If we have two *sorted* lists A_1 and A_2 of lengths n_1 and n_2, then we can produce a *sorted* list A of length $n := n_1 + n_2$ in time linear in n. For this, we simply check the first elements of A_1 and A_2, pick the smaller, remove it from either A_1 or A_2, place it into A, and repeat.

(ii) If the list consists only of a single element, then sorting the list is trivial.

We can easily turn this into a *recursive* algorithm to sort a list A of n integers.

(i) split A into two parts of (approximately) the same size,

(ii) sort each part individually, and

(iii) merge the sub-lists.

For (ii) we recursively use the same approach until the list only contains one element. The algorithm in pseudo-code is shown in Algorithm 4.3.

Algorithm 4.3: MergeSort

```
Input  : array $A = (a_1, \dots, a_n)$
Output : $A$ sorted in ascending order
1  if $n > 1$ then
2      $m \leftarrow \lceil \frac{n}{2} \rceil$                         // split the list
3      $L \leftarrow A_{1,\dots,m}$                                     // lower part
4      $U \leftarrow A_{m+1,\dots,n}$                                   // upper part
5      $L \leftarrow \mathrm{MergeSort}(L)$                             // recursive calls
6      $U \leftarrow \mathrm{MergeSort}(U)$ $i \leftarrow 1,\ j \leftarrow 1$
7      for $k \leftarrow 1, \dots, n$ do                                // merge the two lists
8          if $j > n - m$ or $(i \le m$ and $L_i \le U_j$ then          // take from $L$ if first
9              $A_k \leftarrow L_i$                                     // element is smaller
10             $i \leftarrow i + 1$                                     // or $U$ is empty
11         else
12             $A_k \leftarrow U_j$
13             $j \leftarrow j + 1$
```

In the algorithm we twice call the same algorithm again on a shorter list. Note that n is different for each call. The notation $L \leftarrow A_{1,\dots,m}$ is a short form for copying the elements with indices $1 \le i \le m$ into a new list preserving the order. In a proper implementation on a computer we should avoid copying the list, and instead pass two additional variables that indicate the lower and upper index within A which we want so sort in a recursive step. An example for the algorithm is in Table 4.4.

A simple consideration shows that MergeSort indeed returns a sorted list. (But convince yourself!) We consider its running time. Merging two lists can clearly be done in linear time (in the length of the combined list), but for an estimate of the running time we have to check how often we need to merge lists. Let us work this out in more detail.

In our analysis for the running time on a list of length n we split the running time into two parts. We consider the contributions of the two recursive calls on lists of size approximately $\frac{n}{2}$ and everything else separately.

To simplify our analysis we assume that the length n of our list we want to sort is a power of two, then we recursively call MergeSort twice on lists of the same length. This suffices for this purpose as we are only interested in the asymptotic behaviour (and the running time of our algorithm is monotone in the size of the input).

Let us first consider the split and merge part, disregarding the recursive calls in lines 5 and 6. And for this, we consider split and merge separately. The number $s(k)$ of elementary operations needed to split the list in lines 2 to 4 satisfies

$$a_l k \ \le \ s(k) \ \le \ a_u k$$

for some $a_l, a_u > 0$ and sufficiently large k. To merge the sub-lists returned from the recursions in lines 7 to 13 we do a loop of length k, that in each round does a constant number of comparisons and assignments. Hence, there are constants b_l, b_u such that

20	9	18	2	14	5	12	4	7	initial list
20	9	18	2	14	5	12	4	7	first split, first half
20	9	18	2	14	5	12	4	7	second split, first half
20	9	18	2	14	5	12	4	7	third split, first half
20	9	18	2	14	5	12	4	7	fourth split, first half
20	9	18	2	14	5	12	4	7	fourth split, second half
9	20	18	2	14	5	12	4	7	fourth split, merge
9	20	18	2	14	5	12	4	7	third split, second half
9	18	20	2	14	5	12	4	7	third split, merge
9	18	20	2	14	5	12	4	7	second split, second half
9	18	20	2	14	5	12	4	7	third split, first half
9	18	20	2	14	5	12	4	7	third split, second half
9	18	20	2	14	5	12	4	7	third split, merge
2	9	14	18	20	5	12	4	7	second split, merge
2	9	14	18	20	5	12	4	7	first split, second half
2	9	14	18	20	5	12	4	7	second split, first half
2	9	14	18	20	5	12	4	7	third split, first half
2	9	14	18	20	5	12	4	7	third split, second half
2	9	14	18	20	5	12	4	7	third split, merge
2	9	14	18	20	5	12	4	7	second split, second half
2	9	14	18	20	5	12	4	7	third split, first half
2	9	14	18	20	5	12	4	7	third split, second half
2	9	14	18	20	5	12	4	7	third split, merge
2	9	14	18	20	4	5	7	12	second split, merge
2	4	5	7	9	12	14	18	20	first split, merge

Table 4.4: MergeSort

the number $m(k)$ of operations in the merging step satisfies

$$b_l k \leq m(k) \leq b_u k$$

for sufficiently large k. Setting $c_l := \min(a_l, b_l)$ and $c_u := \max(a_u, b_u)$ we obtain that the number $f_s(k)$ of elementary operations in a run of $\mathrm{MergeSort}$ of length k without the recursion satisfies

$$c_l k \leq m(k) \leq c_u k$$

for sufficiently large k.

Now let us consider the two recursive calls. We can view the sequence of recursive steps as a binary tree. If we start with a list of length n then on each level j we work on 2^j lists of length $n/2^j$. Hence, on level j we spend time at most $c_u \cdot n/2^j \cdot 2^j = c_u n$ for all operations except the descents.

By assumption, $n = 2^m$ is a power of 2, so we do $m = \log_2(n)$ splits to reach lists of length 1. Hence, we have $\log_2(n)$ levels to work on, so in total the number $f(n)$ of elementary operations in $\mathrm{MergeSort}$ including the recursion satisfies

$$f(n) \leq c_u \cdot n \log_2 n\,.$$

We can repeat the same argument for the lower bound to obtain

$$f(n) \geq c_l \cdot n \log_2 n\,.$$

So in total we have proved the following theorem.

Theorem 4.4. $\mathrm{MergeSort}$ *sorts a list A in time $\Theta(n \log n)$.*

Remark. Our analysis is a special case of the so called *Master Theorem* that gives running times for various types of recursive algorithms.

Can we do even better, *i.e.,* can we sort in time less than $\mathcal{O}(n \log n)$? The answer depends on the methods we can use. The MergeSort algorithm relies on binary decisions, the comparison of two numbers. In this case the arrangement in a binary tree as done during the Merge Sort algorithm is best possible, we cannot get away with less comparisons, and hence also not with shorter running time.

Theorem 4.5. *Every sorting algorithm that relies on pairwise comparison has running time in $\Omega(n \log n)$.*

Proof. Consider the tree of decisions we could make during the course of the algorithm for any list A of n mutually distinct numbers. Each node in the tree corresponds to one possible comparison, and a path from the root to a leaf is the chain of decisions we do for one particular input. Depending on this chain the outcome of the algorithm, which is a permutation of L, is different.

Algorithm 4.4: LinearSearch

```
Input  : array $A = (a_1, \ldots, a_n)$ and number $b$
Output : the index of $b$ in $A$, or $n+1$ is $b$ is not in $A$.
1 for $i \leftarrow 1, \ldots, n$ do
2     if $a_i = b$ then
3         return $i$
4 return $n+1$
```

Algorithm 4.5: BinarySearch

```
Input  : sorted array $A = (a_1, \ldots, a_n)$ and number $b$
Output : the smallest index of $b$ in $A$, or $n+1$ is $b$ is not in $A$.
1  $l \leftarrow 1$
2  $u \leftarrow n$
3  while $l < u$ do
4      $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
5      if $a_m < b$ then
6          $l \leftarrow m+1$
7      else
8          $u \leftarrow m$
9  if $a_l = b$ then
10     return $l$
11 return $n+1$
```

There are $n!$ different permutations for a set of n elements, *i.e.*, $n!$ different possible orders of L, so the tree as $n!$ leaves. The number of comparisons we have to make in the worst case is the length h of a longest path from a leaf to the root. As we rely on pairwise comparisons, each node has at most two children (adjacent nodes further away from the root), so we have at most 2^h leaves in the tree. Hence

$$h \geq \log_2(n!) = \log_2\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log_2(i) \geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) = \Omega(n \log n)\,,$$

where for the last inequality we use the monotonicity of log and discard the lower half of the summands. □

If, however, we have more information about our problem we may be able to improve (and, in fact, in many cases we can). For example, if we know in advance that all numbers lie in a certain range of length m not significantly larger than the size n of the list we want to sort, then *Bucket Sort* runs in linear time: Create an list $C = (c_0, \ldots, c_m)$ of *counters* of length m initialized with 0. Now run through your input list $A = (a_1, \ldots,_n)$ once and for each a_j increment the counter at position a_j by one. In a second loop go over the list of counters C and place c_j times the number j into the output list.

4.3 Binary Search

Let us briefly also discuss the search problem, *i.e.*, the task to determine whether an element b appears in a given list $A = (a_1, \ldots, a_m)$, and if it does, return its index in A. We can solve this in time $\Theta(n)$ by comparing b to all elements in A, and in the worst case we really have to compare to all elements. This is the linear search algorithm shown in Algorithm 4.4.

For sorted lists we can do much better. Here, we can use an approach similar to the one of merge sort. For the list A of length n we pick the index $m := \frac{n}{2}$ and compare b to a_m. If they are equal we have found the index. If not, then either b is smaller than a_m, and thus necessarily it occurs at most in the lower half of A. Or it is bigger and thus occurs in the upper half of A. In any case we can discard half of the list and repeat the procedure with the remaining half. This is the binary search algorithm shown in Algorithm 4.5.

Theorem 4.6. $\mathrm{BinarySearch}$ *for a number b and a sorted list A of length n runs in time* $\Theta(\log n)$.

Proof. The algorithm terminates as in each iteration either l increases by at least one or u decreased by at least one.

If b is in the list, then the index we are looking for is always between l and u. Hence, we get the correct index. If b is not in the list, then at some point $l > u$ and we return $n + 1$.

In any case we do between $\lfloor \log_2 n \rfloor$ and $\lceil \log_2 n \rceil + 1$ iterations. The number of elementary operations inside the loop is constant. This implies the running time. □

5 Graph Traversal

Contents

In Section 3.6 we have discussed options to encode a graph for algorithmic and computational purposes. In particular, we learned about incidence matrices and adjacency lists. All these structures encode *local* data for the graph. An adjacency list, *e.g.*, tells us the neighbors of each node. However, given such an adjacency list we cannot directly decide whether the graph is, *e.g.*, bipartite, or connected. Those are *global* properties, and our local data structures do not allow us to deduce such immediately. Thus, we need a method to explore the whole graph, given our local information.

This chapter introduces two essentially different algorithmic techniques to traverse and *visit* all nodes of a graph given by local data in a controlled and efficient way. Both methods explore the graph along a spanning tree. We have identified those in Section 2.5.2 as efficient substructures of a graph already capturing some interesting properties.

These methods are rarely important in itself, but are a key method for many other efficient algorithms, where at each *visited node* this algorithm operates on data associated with the node (*e.g.*, its degree). In our exposition of the two algorithms we just maintain a value visit in each node that is set to TRUE once we have seen the node. When using the graph traversal as a tool this is the point were we would access the data at a node.

The two methods differ in the order in which they traverse the nodes of a graph. Both have their advantages, and we work out the defining property of each order in the next sections.

The first method is *Breadth First Search (BFS)*. It explores a graph in layers around a root node r. It starts to visit the node r, and then successively visits all nodes in the open neighborhood of the already visited nodes. Thus, it visits nodes in the order of their *distance* from r, measured in the number of edges of a shortest path from a node to r (see Theorem 5.10).

version of January 26, 2021

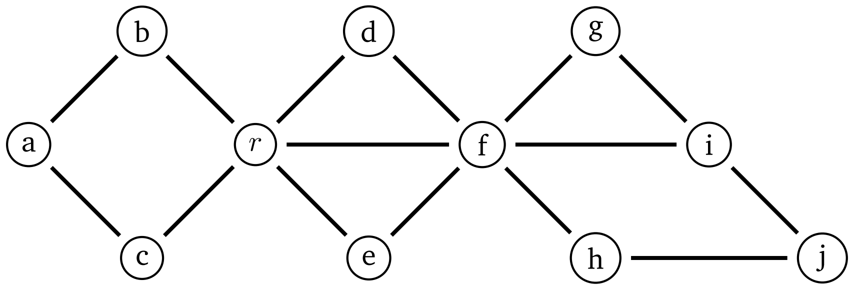


Figure 5.1: The graph for our examples in the next sections. We assume that all neighbourhoods are listed lexicographically in loops.

The second method is *Depth First Search (DFS)*. As Breadth First Search it starts with some root node r, but tries do explore the full extension of the graph first. It does this by following a path through the graph for as long as possible without revisiting a node. Once stuck, it backtracks until it can take a turn to some yet unvisited node.

You may have met this algorithmic idea already as the *left hand rule* to find the path out of a maze. In this approach you chose at each crossing the leftmost turn that leads to a crossing you have not yet been, and if no further turn is possible at a crossing, and you have not yet found the exit, you backtrack.

Both algorithms have the same running time (Theorems 5.2 and 5.4). So which of the two one should choose mostly depends on the problem that requires a graph traversal. We will see in later sections applications where we can use one but not the other.

We introduce the precise definition and pseudo-code for the two algorithms and determine the running times in the following sections. We discuss properties, extensions and some applications in subsequent sections.

5.1 Breadth First Search

In *Breadth-First-Search (BFS) [Breitensuche]* we traverse and *visit* the nodes of a graph $G = (V, E)$ in levels around a given *root node* r.

The algorithm starts at r, visits this node (*i.e.,* may do something with the associated data) and adds it to the set S of *visited nodes*. It continues to visit nodes one by one in the open neighbourhood of the already visited ones. This step is repeated until the neighborhood $\mathrm{N}(S)$ of the visited nodes is empty. At this point we have visited all nodes in the connected component of r.

Let $N_0 := \{r\}$ and $N_j := \mathrm{N}(N_{j-1})$ for $j \geq 1$. Then each node $v \in V$ is in at most one set N_j for some $j \geq 0$ (it is not in any if v is not in the connected component of r). We call j the *level* of v. We obtain a *level function* $\mathsf{level} : V \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$, where we assign level ∞ to nodes not in any N_j.

We do not determine the neighborhoods $\mathrm{N}(N_j)$ of N_j explicitly. Instead, we consider the neighborhoods $\mathrm{N}(u)$ of each $u \in N_j$ successively. Each unvisited node in $\mathrm{N}(u)$ is in $\mathrm{N}(N_j)$, so the algorithm visits it and places it in N_{j+1}. Thus, for each $v \in N_{j+1}$ there is a unique node u in whose neighborhood we first found v. We call u the *parent* or *predecessor* of v. This defines a function $\mathsf{parent} : V \to V \cup \{\mathsf{nil}\}$ (where we need the one

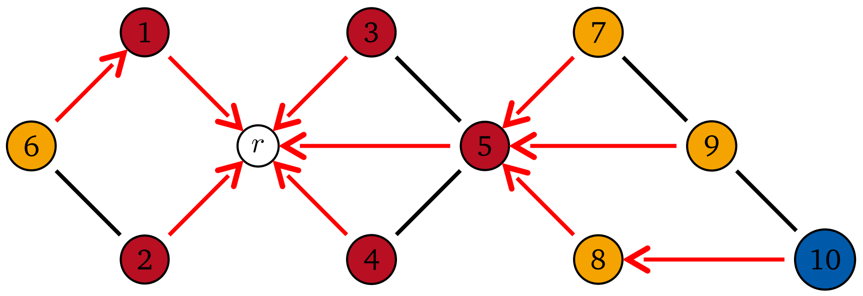


Figure 5.2: The levels and the parent relation for BFS. The red nodes are on level 1, the yellow nodes on 2, and the blue node on level 3. The arrows point from a node to its parent node.
The nodes are numbered in the order they are visited by BFS (assuming the neighborhoods are ordered lexicographically with labels as in Figure 5.1).

extra element in the image for the root node r, which has no parent node).

Note that this function may depend on the order in which we traverse the neighborhood of a node. We do our examples with the graph of Figure 5.1 and use the lexicographic order for the neighborhood of each node. We will see that parent defines a spanning tree in the graph. Figure 5.2 shows an example of the levels and the parent relation assigned by BFS for our example.

The algorithm does not maintain the sets N_j explicitly. Instead, it places an unvisited node found in the neighborhood of some $u \in N_j$ at the end of a *waiting line* Q, starting with the node r. It retrieves the next node u for which it checks the neighborhood from the front. As we check the complete open neighborhood N_{j+1} of N_j, for $j \geq 0$, before switching to the next neighborhood, N_{j+1} is in Q before any node from N_{j+2} is added at the end. Hence, it retrieves all nodes of N_{j+1} before it sees a node of N_{j+2} and implicitly obeys the order we want.

Such a data structure Q that realizes a waiting line is called a *first-in-first-out (FIFO)* data structure, or *queue*. Its implementation in a programming language is usually based on a list, for which we have the two operations $\mathsf{push\_back}(Q, v)$ that adds v to the end of the list and $\mathsf{pop\_front}(Q)$ that removes the first element and returns it. In efficient implementations both operations are assumed to take time $\mathcal{O}(1)$. In our analysis of the algorithm we can therefore treat them as elementary operations.

The pseudo-code for the Breadth First Search Algorithm formalizing all we have now discussed is in Algorithm 5.1.

Proposition 5.1. *Let $G = (V, E)$ be a graph with n nodes and $r \in V$. BFS terminates after finitely many steps and visits each node of G reachable from r exactly once.*

A node v is *reachable* from r if and only if there is a path $r = v_0 v_1 \dots v_k = v$ from r to v. By Definition 2.13 (and Proposition 2.14) those are the nodes in the connected component of r.

Proof. A node v is added to Q in line 15 if and only if $\mathsf{visit}(v) = 0$, and $\mathsf{visit}(v)$ is set to 1 in the following line. Hence, v is visited and enters Q at most once. In each iteration

Algorithm 5.1: BFS(r)

```
Input  : graph $G = (V, E)$, $r \in V$
Output : parent parent : $V \to V \cup \{\text{nil}\}$
         level level : $V \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$

1  foreach $v \in V \setminus \{r\}$ do                  // initialize
2      level($v$) ← $\infty$
3      parent($v$) ← nil
4      visit($v$) ← FALSE
5
6  $Q$ ← ($r$)                                 // initialize queue with root
7  level($r$) ← 0
8  visit($r$) ← TRUE                                   // visit the root
9
10 while $Q \neq \emptyset$ do                       // work on the queue until empty
11     $u$ ← pop_front($Q$)                        // get first element of queue
12     foreach $v \in \mathrm{N}(u)$ do                // queue all unseen neighbors
13         if visit($v$) = FALSE then
14             visit($v$) ← true                          // visit the node
15             push_back($Q, v$)                      // add $v$ at the end of $Q$
16             parent($v$) ← $u$                     // we discovered $v$ from $u$
17             level($v$) ← level($u$) + 1                   // now $v$ is seen
```

of the loop in line 10 one node is removed from Q, so the algorithm stops after at most n iterations.

It remains that check that each node reachable from r is visited. We prove this by induction over the length k of such a path. The only node connected to r with a path of length 0 is r itself, and r is visited at the very beginning.

Now, by induction, we assume that all nodes connected to r by a path of length at most $k-1$ are visited. Let v be the a node connected to r by a path of length k and let w be the node prior to v on that path (there may, of course, be several paths with the same length, and we choose one at will). Then w is visited at some point, and thus also added to Q. The algorithm only stops if Q is empty, so at some point w is returned from Q in line 11. At this point either v is already visited or it will be visited. □

This allows us to determine the running time of BFS. In the following theorems we assume that our graph $G = (V, E)$ has $n := |V|$ nodes and $m := |E|$ edges given in a form that the number of elementary operations needed to access all $\deg(v)$ neighbors of a node is proportional to $\deg(v)$. This is, *e.g.*, true if the graph is given by an adjacency list.

Theorem 5.2. *Let G be a connected graph. Then BFS runs in time $\Theta(n+m)$.*

Proof. In a connected graph each node is reachable from r, so we visit each node in the graph exactly once.

Algorithm 5.2: DFS(r)

```
Input  : graph $G = (V, E)$, $r \in V$
Output : parent parent : $V \to V \cup \{\text{nil}\}$
         level level : $V \to \mathbb{Z}_{\geq 0}$

foreach $v \in V$ do
    level($v$) ← $\infty$
    parent($v$) ← nil
    visit($v$) ← FALSE
$S \leftarrow (r)$                          // initialize stack with the root
level($r$) ← $0$                                 // set level of root

while $S \neq \emptyset$ do                   // work on the stack until empty
    $u$ ← pop_front($S$)                      // get first element of stack
    if visit($u$) = FALSE then
        visit($u$) ← TRUE                                 // and visit it
        foreach $v \in N(u)$ do
            if visit($v$) = FALSE then
                push_front($S, v$)            // add $u$ at the front of $S$
                level($v$) ← level($u$) + 1   // we set a tentative level
                parent($v$) ← $u$              // and a tentative parent
```

Initialization in lines 1 to 4 takes time $\Theta(n)$. Each node is placed into and removed from Q exactly once. This requires $\Theta(n)$ elementary operations. For each node u taken from Q we look at all adjacent nodes. Hence in total we consider each edge twice. This requires $\Theta(m)$ elementary operations. So in total we get a running time in $\Theta(m + n)$. □

The indicator function visit we have introduced is convenient for the proofs, but not really necessary, unless we use it in an application. The level function level(v) is finite if and only if a node has been visited, so we could use this instead of visit.

5.2 Depth First Search

Depth-First-Search (DFS) [Tiefensuche] again starts with a root node r, but this time, instead of visiting all its neighbors, we try to follow a path from r in the graph for as long as possible without revisiting a node. If we get stuck in this we return on this path (we backtrack) until we find a node from which we can start a new path.

We can define the same two functions level(v) and parent(v) as for BFS. This time, we set the parent parent(v) of a node v to the node u we came from when we first visited v, and define level(v) := level(u) + 1. The algorithm is shown in Algorithm 5.2.

In this algorithm we place all neighbors of a node at the *front* of our list S where we remember nodes we still have to look at. We also retrieve them from the front. This is similarly to a pile of books, you should take off the last one you placed on it first. Such

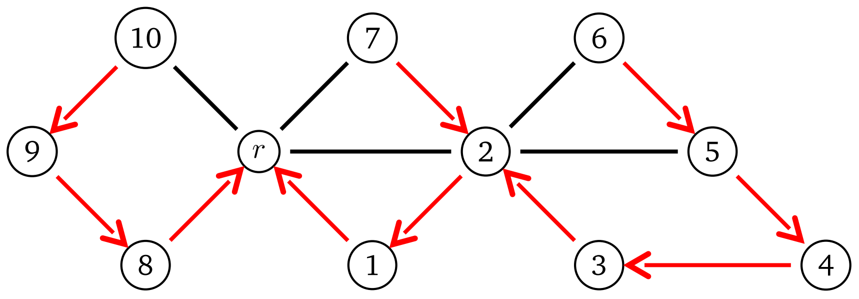


Figure 5.3: The levels and the parent relation for DFS. The red nodes are on level 1, the yellow nodes on 2, and the blue node on level 3. The arrows point from a node to its parent node.
The nodes are numbered in the order they are visited by DFS (assuming the neighborhoods are ordered lexicographically with labels as in Figure 5.1).

a data structure is called a *last-in-first-out (LIFO)* data structure usually implemented as a *stack*. For a stack we have the two operations push_front(S, v) that adds v to the front of the list and pop_front(S) that removes the first element and returns it. Again, efficient implementations can do this in time $\mathcal{O}(1)$, so we treat them as elementary operations.

Also here, we will see that parent defines a spanning tree in the graph. Figure 5.3 shows an example of the levels and the parent relation assigned by DFS for the graph shown in Figure 5.1.

The following two statements are similar to Proposition 5.1 and Theorem 5.2, with subtle differences in the proof.

Proposition 5.3. *Let $G = (V, E)$ be a graph with n nodes and $r \in V$. DFS terminates after finitely many steps and visits each node of G reachable from r exactly once.*

Proof. A node v found in the neighborhood of a node u is pushed onto S in line 14 if and only if both visit(u) = FALSE and visit(v) = FALSE, and visit(u) is set to TRUE in this process. Hence, we get at most $\deg(v)$ copies of v in S. In each iteration of the loop in line 8 one node is removed from S, so the algorithm stops after at most n iterations.

The same induction over the length k of a path from r to v as for BFS now shows that each node v reachable from r is visited. The only node connected to r with a path of length 0 is r itself, and r is visited as the first node in line 11.

Now, we assume that all nodes connected to r by a path of length at most $k - 1$ are visited. Let v be the a node connected to r by a path of length k and let w be the node prior to v on that path (there may, of course, be several paths with the same length, and we choose one at will). Then w is visited at some point, and all its unvisited neighbors are pushed onto S. The algorithm only stops if S is empty, so at some point v is returned from S in line 9. At this point either v is already visited or it will be visited. □

Theorem 5.4. *Let G be a connected graph. Then DFS runs in time $\Theta(n + m)$.*

Proof. In a connected graph each node is reachable from r, so we visit each node in the graph exactly once.

Initialization in lines 1 to 4 takes time $\Theta(n)$. Each node u is placed into and removed from S at most $\deg(u)$ times. By Theorem 2.8 this requires $\Theta(n)$ elementary operations. For each node u taken from Q we look at all adjacent nodes. Hence in total we consider each edge twice. This requires again $\Theta(m)$ elementary operations. So in total we get a running time in $\Theta(m+n)$. □

Observe that this Theorem and Theorem 5.2 immediately extend to graphs with several components. In this case the running time is linear in the number of nodes of the whole graph (because we need to initialize!) and the number of edges of the connected component containing r. As we do not know this before computing a spanning tree, we can only bound from above and get a running time in $\mathcal{O}(n+m)$.

5.3 Connectivity

We provide the interpretation of the function parent. For this, let us define a subgraph $T = (W, F)$ with node set $W := \{v \mid \mathsf{level}(v) < \infty\}$ and edge set $F := \{\{u, \mathsf{parent}(u)\} \mid \mathsf{parent}(u) \neq \mathsf{nil}\}$.

Proposition 5.5. *Let $G = (V, E)$ be a graph and $G_r = (V_r, E_r)$ the connected component containing r. Then the subgraph $T = (W, F)$ produced by either BFS or DFS is a spanning tree in G_r, and $\mathsf{level}(v)$ is the length of the unique path between v and r in T.*

Proof. Assume that T contains a cycle C, let v be a node with maximal $\mathsf{level}(v)$ on C and u, w its two adjacent nodes. Any edge is between a node and its parent, and any node has at most one parent. So v is a parent of either u or w. Assume that $v = \mathsf{parent}(u)$. But then $\mathsf{level}(u) = \mathsf{level}(v) + 1$, contradicting the choice of v. So T is a tree, and by Proposition 5.1, resp. Proposition 5.3, it is spanning in G_r.

In a tree there is a unique path between any two nodes, so there is a unique path $r = v_0 v_1 \ldots v_k = v$ in T. By construction, $v_{j-1} = \mathsf{parent}(v_j)$ and $\mathsf{level}(r) = 0$, so $\ell(v_j) = j$ for $1 \leq j \leq k$. □

Theorem 5.2 or Theorem 5.4 immediately provides a method to check whether a graph is connected.

Corollary 5.6. *BFS and DFS determine whether a graph is connected in time $\mathcal{O}(n+m)$.*

Proof. We pick any node r in $G = (V, E)$. If the connected component of r has $m_r < m$ nodes, then both algorithms run in time $\mathcal{O}(n + m_r) \subseteq \mathcal{O}(n+m)$. Now we can check in time $\mathcal{O}(n)$ whether there are any unvisited nodes left. □

Algorithm 5.3: BFS-Traversal(G)

Input : graph $G = (V, E)$

```
Function BFS-Visit($v$):
    $Q \leftarrow (v)$
    while $Q \neq \emptyset$ do
        $u \leftarrow$ push_front($Q$)
        foreach $v \in \mathrm{N}(u)$ do
            if level($v$) = $\infty$ then
                $Q \leftarrow Q \oplus (v)$
                $visit(v) \leftarrow$ TRUE
                parent($v$) $\leftarrow u$
                level($v$) $\leftarrow$ level($u$) + 1

foreach $v \in V$ do
    level($v$) $\leftarrow \infty$
    parent($v$) $\leftarrow$ nil
    visit($v$) $\leftarrow$ FALSE
foreach $v \in V$ do
    if visit($v$) = FALSE then
        BFS-visit(v);
```

Algorithm 5.4: DFS-Traversal(G)

Input : graph $G = (V, E)$

```
Function DFS-Visit($v$):
    $S \leftarrow (v)$
    while $S \neq \emptyset$ do
        $u \leftarrow$ push_front($S$)
        visit($u$) $\leftarrow$ TRUE
        foreach $v \in \mathrm{N}(u)$ do
            if visit($v$) = FALSE then
                parent($v$) $\leftarrow u$
                level($v$) $\leftarrow$ level($u$) + 1
                $S \leftarrow (v) \oplus S$

foreach $v \in V$ do
    level($v$) $\leftarrow \infty$
    parent($v$) $\leftarrow$ nil
    visit($v$) $\leftarrow$ FALSE
foreach $v \in V$ do
    if visit($v$) = FALSE then
        DFS-Visit(v);
```

So far, both algorithms only traverse the connected component that contains the root node r. This may be sufficient for some applications, *e.g.*, for the test of connectivity in the previous corollary, but usually one wants to visit every node, irrespective of the connected component it is contained in. Also, we may want to determine the exact number of connected components. This can easily be achieved by wrapping our algorithm with a function that restarts BFS or DFS with a new root node, as long as there are still unvisited nodes left. This is shown in Algorithm 5.3 and Algorithm 5.4.

These algorithms no longer return a spanning tree in the graph, but a *spanning forest*, with one tree per connected component. The level function now tells us the length of the path from a node to the chosen root in its component. Hence, it follows from our considerations so far, that BFS and DFS allow us to compute all connected components in time $\Theta(n + m)$. We fix this with the following theorem.

Theorem 5.7. *BFS-Traversal and DFS-Traversal visit every node and determine all connected components in a graph in time $\Theta(n + m)$.* □

5.4 Distance

The level function returned by Breadth-First-Search contains more information, which we want to explain now. For this, let us define the *path distance* between nodes in a graph.

Definition 5.8. Let $G = (V, E)$ be a graph. The *edge distance [kürzester Abstand]* between nodes $v, w \in V$ is

$$d(v,w) := \begin{cases} \min\{k \mid v = v_0 v_1 \dots v_k = w \text{ path}\} & \text{if } v,\ w \text{ are connected,} \\ \infty & \text{otherwise.} \end{cases}$$

Any path of length $d(v, w)$ between the nodes $v, w \in V$ is a *shortest path [kürzester Weg]* between v and w.

With the following lemma we justify the name *distance function* for $d(v, w)$.

Lemma 5.9. *Let $G = (V, E)$ be a graph and let $u, v, w \in V, \{v, w\} \in E$. Then $d(u, v) \leq d(u, w) + 1$.*

Proof. If $d(u, w) < \infty$ then $d(u, v) < \infty$ and $\{v, w\}$ extends any path from u to w. Otherwise the inequality is trivial. □

This shows that $d(v, w)$ indeed has the properties we would intuitively expect from a distance measure. Namely, it

- ▷ is *symmetric, i.e.,* $d(v, w) = d(w, v)$, it
- ▷ is *reflexive, i.e.,* $d(v, v) = 0$, and it
- ▷ satisfies the *triangle inequality*.

The last property is the only nontrivial property we should expect from a distance measure. Namely, we require that any detours can only increase the length, or formally

$$d(u,w) \leq d(u,v) + d(v,w) \qquad \text{for any three } u, v, w\,.$$

Theorem 5.10. *Let G be a connected graph, $r \in V(G)$, and* $\mathsf{level} : V \longrightarrow \mathbb{Z}_{\geq 0} \cup \{\infty\}$ *obtained from BFS. Then* $\mathsf{level}(v) = d(r, v)$ *for all nodes $v \in V(G)$.*

Proof. The spanning tree T contains a (unique) path from r to any node v. By construction, $\mathsf{level}(v)$ is the number of edges on this path to v. Thus, $d(r, v) \leq \mathsf{level}(v)$.

Moreover, when considering node u in line 11 of Algorithm 5.1, there exists no node v with $\mathsf{level}(v) < \infty$ and $\mathsf{level}(v) > \mathsf{level}(u) + 1$. This is true, since the nodes in Q are sorted in increasing order of level and only nodes with level one larger can be added to Q.

Assume that there exists a node v such that $d(r, v) < \mathsf{level}(v)$ and assume that v is the node with smallest distance to r with this property. Let P be a shortest path from r to v in G. Let $\{u, v\}$ be the last edge on P (which does not belong to T). By assumption, $d(r, u) = \mathsf{level}(u) < \infty$. Thus,

$$\mathsf{level}(v) > d(r,v) = d(r,u) + 1 = \mathsf{level}(u) + 1\,,$$

since $\{u, v\}$ lies on a shortest path to v and by assumption $d(r, u) = \mathsf{level}(u)$. Consequently, the above observation shows that at the time node u is considered in line 11, node v cannot be in Q and thus $\mathsf{level}(v) = \infty$. But then v would have been considered in line 17, because of edge $\{u, v\}$ and we would set $\mathsf{level}(v) = \mathsf{level}(u) + 1$, a contradiction. □

* 5.5 Node Orders

We can characterize both algorithms also by the order in which they visit the nodes of our graph. For a graph $G = (V, E)$ with node set $V = \{v_1, \ldots, v_n\}$ and a given fixed order $\sigma := (v_1, v_2, \ldots, v_n)$ we define the functions

$$\begin{aligned} I^- : V &\longrightarrow [n] \cup \{\infty\} \\ v_i &\longmapsto \begin{cases} \min\left(j < i \mid v_i \in \mathrm{N}(v_j)\right) & v_i \in \mathrm{N}(v_j) \text{ for some } j < i \\ \infty & \text{otherwise} \end{cases} \end{aligned}$$

and

$$\begin{aligned} I^+ : V &\longrightarrow [n] \cup \{0\} \\ v_i &\longmapsto \begin{cases} \max\left(j < i \mid v_i \in \mathrm{N}(v_j)\right) & v_i \in \mathrm{N}(v_j) \text{ for some } j < i \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

We say that σ is a *breadth-first* order of the nodes in V if $I^-(v_i) = \min(I^-(v_k) \mid k \geq i)$ for all $1 \leq i \leq n$, and σ is a *depth-first* order of the nodes in V if $I^+(v_i) = \max(I^+(v_k) \mid k \geq i)$ for all $1 \leq i \leq n$. The proof of the following theorem is left as an exercise.

Theorem 5.11. *Let $G = (V, E)$ be a connected graph.*

(i) BFS visits the node in breadth-first order.

(ii) DFS visits the node in depth-first order. □

5.6 Variations and Applications

Algorithms that rely on a stack as their key data structure can often be formulated quite elegantly with recursion. This is also true for *Depth First Search*, and the pseudo-code for a recursive version of DFS-Traversal is shown in Algorithm 5.5. Instead of explicitly pushing the nodes of the neighborhood onto the stack, where they are returned from one by one in the next iterations, we recursively start a new loop over the next neighborhood.

With our convention to list nodes in the neighborhood lexicographically, this will visit the nodes in a different order than the iterative version. See Figure 5.4. This is because in the iterative version we push nodes onto the stack in lexicographic order, so the lexicographically last node is returned first. In the recursive version we consider first the lexicographically first node. Of course, we can change them to produce the same

Algorithm 5.5: DFS-Traversal-Recursive

```
Input  : graph $G = (V, E)$
Output : parent function parent : $V \to V \cup \{nil\}$
1  Function DFS-Visit($r$):
2      visit($r$) ← 1
3      foreach $v \in N(r)$ do                 // run over unseen neighbors
4          if visit($v$) = 0 then
5              parent($v$) = $r$
6              DFS-Visit($v$)                   // recursion
7
8  foreach $v \in V$ do                          // initialize
9      visit($v$) ← 0
10     parent($v$) ← nil
11 foreach $v \in V$ do                          // run over unseen nodes
12     if visit($v$) = 0 then
13         DFS-Visit($v$)
```

order if in one we run through the neighborhood of a node in reverse lexicographic order.

A recursive formulation may follow your intuition of the algorithm more closely and can be implemented faster and with less lines of code. However, depending on your programming language and compiler, it may come with a small runtime penalty. With each recursive call, the computer has to put a copy of the current variable assignments aside to be able to restore them once the recursive call returns (*i.e.,* it internally maintains a stack for you, on which it pushes the current state of the variables before the next recursive call).

Here are some more applications of BFS and DFS. Again let G be a graph with n nodes and m edges.

Proposition 5.12. *We can check in time $\mathcal{O}(n)$ whether G is acyclic or a tree.*

Proof. To check whether G is acyclic, we can stop BFS or DFS if in the loop over $N(u)$

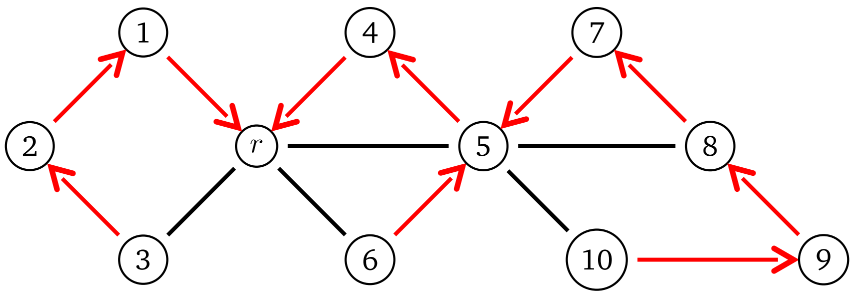


Figure 5.4: The spanning tree generated by recursive DFS

if we find a node $v \neq \mathsf{parent}(u)$ with $\mathsf{level}(v) < \infty$. Up to this point we have seen at most $\mathcal{O}(n)$ edges.

Using this we can decide whether G is a tree by checking whether G is acyclic and whether $m = n - 1$. This is done in time $\mathcal{O}(n)$. □

Proposition 5.13. *Using BFS, we can compute $d(u, v)$ for all nodes $u, v \in V$ in time $\mathcal{O}(n(n + m))$.*

Proof. Call BFS for each node $v \in V$ and set $d(v, w) = \mathsf{level}(w)$ for $w \in V$. □

Proposition 5.14. *Let G be a connected graph and T the spanning tree obtained by BFS. If $e = uv \in G$ is an edge not in T, then $|\mathsf{level}(v) - \mathsf{level}(u)| \leq 1$.*

Proof. Assume $\mathsf{level}(v) \geq \mathsf{level}(u) + 2$ and let P be the path in T from r to u. Then $P + e$ is a path of length $\mathsf{level}(u) + 1 \leq levelf(v)$ from r to v. This contradicts Theorem 5.10. A similar argument works for the case $\mathsf{level}(u) \geq \mathsf{level}(v) + 2$. □

Proposition 5.15. *Let G be a connected graph. We can check with BFS in time $\mathcal{O}(n + m)$ whether G is bipartite.*

Proof. Call BFS starting from an arbitrary root $r \in V$, let T be the produced spanning tree and check whether there is some edge $e = uv$ with $\mathsf{level}(u) = \mathsf{level}(v)$ (this edge cannot be in T). This can be done in $\mathcal{O}(n + m)$, and we claim that G is bipartite if and only if no such e exists.

If $\mathsf{level}(u) = \mathsf{level}(v)$ for some edge $e = uv$ not in T, then the cycle closed by e has length $\mathsf{level}(u) + \mathsf{level}(v) + 1 = 2\ell(u) + 1$, which is odd.

Conversely, assume $|\mathsf{level}(v) - \mathsf{level}(u)| = 1$ for all edges in G. If $v_0 v_1 \dots v_k = v_0$ is a cycle in G, then

$$\sum_{i=0}^{k-1} \mathsf{level}(v_i) - \mathsf{level}(v_{i+1}) = 0$$

and each summand is ± 1, so we have an even number of summands. Hecne, C is even. □

6 Minimum Spanning Trees

Contents

In this chapter we enhance the graph structure by *weights* or *costs* on the edges (think of lengths, or transportation costs). Given these weights on an edge we consider the problem of connecting all vertices of a graph G using only a subset F of the edges whose total sum of the weights is as small (or, if we consider *costs*, as *cheap*) as possible.

Clearly, such a set F must contain a spanning tree of the graph, and a simple consideration shows, that the edges in F are actually a spanning tree. Hence, we have solved this problem in the previous chapter, if all costs on the edges are identical. We have seen, that any spanning tree of a connected graph G with n nodes has $n-1$ edges, so any spanning tree has total weight $c(n-1)$.

We, however, need a new idea if those weights are not all equal. The solution we consider first, *Kruskal's Algorithm* will introduce us to the large class of "greedy" algorithms. We consider a more efficient algorithm later. Let us first give a proper definition of a *weight* or *cost function*.

Definition 6.1. Let $G = (V, E)$ be a graph. A *weight (or cost) function [Gewichts- oder Kostenfunktion]* on G is a function

$$w : E \to \mathbb{R}.$$

A *weighted graph [gewichteter Graph]* is a graph G together with a weight function w on its edges.

The *weight [Gewicht]* of a subgraph $H = (W, F)$ of G is

$$w(H) := \sum_{f \in F} w(f).$$

A *minimum spanning tree [minimal aufspannender Baum]* in G is is a spanning tree T of G of minimum weight.

Minimum Spanning Tree (MST) Problem.

input:	a connected undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$.
problem:	find a spanning tree T of G of minimum weight.

Definition 6.2. Let $G = (V, E)$ be a graph. A *cut [Schnitt]* $\mathrm{C}_G(S) \subseteq E$ induced by some $S \subseteq V$ in G is the set of edges that connect S with its complement $V \setminus S$,

$$\mathrm{C}_G(S) \;:=\; \{ e \in E \mid |e \cap S| \;=\; |e \cap V \setminus S| \;=\; 1 \} \,.$$

Proposition 6.3. *Let $G = (V, E)$ be a weighted connected graph with a spanning tree $T = (V, F)$. T is a minimum spanning tree if and only if any $e \in F$ has minimum weight among all edges in the cut C induced by the two components of $T - e$.*

Proof. Observe that by Theorem 2.22(iv) we indeed have a unique cut defined by e.

Assume first that T is a minimum spanning tree. For any $f \in C$ the graph $T - e + f$ is a spanning tree of weight $w(T) - w(e) + w(f) \geq w(T)$, so $w(f) \geq w(e)$.

Conversely, assume that any edge in T has minimum weight in the cut it induces. If T is not a minimum spanning tree, then any minimum spanning tree S differs from T in at least one edge. Pick one minimum spanning tree $S = (V, F')$ such that $F \cap F'$ is maximal. Choose $e \in F \setminus F'$ and let C be the cut induced by e. $S + e$ has a cycle, so there is $f \in F' \cap C$.

By assumption $w(f) \geq w(e)$. If $w(e) = w(f)$, then $w(S - f + e) = w(S) - w(f) + w(e) = w(S)$ and $S - f + e$ is a minimum spanning tree that has one more edge in common with T, contradicting our choice. So $w(f) > w(e)$, but then $w(S - f + e) < w(S)$, and S was not minimum. So T is a minimum spanning tree. $\square$

In this proposition we have used the property that trees are *minimally connected*. We can in the same way characterize minimum spanning trees using the the dual property that trees are *maximally acyclic*. This is the next proposition, which as a pretty similar proof, if we exchange *cut* and *minimal* with *cycle* and *maximal*.

Proposition 6.4. *Let $G = (V, E)$ be a weighted connected graph with a spanning tree $T = (V, F)$. T is a minimum spanning tree if and only of any $f \in E \setminus F$ has maximum weight among all edges in the unique cycle C in $T + f$.*

Proof. By Theorem 2.22(v) there is indeed a unique cycle C.

Assume first that T is a minimum spanning tree. For any $e \in C$ the graph $T + f - e$ is again a spanning tree. It has weight $w(T) + w(f) - w(e)$, so $w(f) \geq w(e)$.

Conversely, assume that any edge in $E \setminus F$ has minimum weight in the cycle it induces in T. If T is not a minimum spanning tree, then any minimum spanning tree S differs

Algorithm 6.1: Kruskal Algorithm

```
Input  : connected graph $G = (V, E)$
         weight function $w : E \to \mathbb{R}$
Output : minimal spanning tree $T = (V, F)$
         of $G$
1 $T \leftarrow (V, \varnothing)$
2 sort $E = \{e_1, \ldots, e_m\}$ such that
    $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
3 for $j = 1, \ldots, m$ do
4     if $T + e_j$ is acyclic then
5         $T \leftarrow T + e_j$
```

Algorithm 6.2: Dual Kruskal Algorithm

```
Input  : connected graph $G = (V, E)$
         weight function $w : E \to \mathbb{R}$
Output : minimal spanning tree $T = (V, F)$
         of $G$
1 $T \leftarrow (V, E)$
2 sort $E = \{e_1, \ldots, e_m\}$ such that
    $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_m)$
3 for $j = 1, \ldots, m$ do
4     if $T - e_j$ is connected then
5         $T \leftarrow T - e_j$
```

from T in at least one edge. Pick one minimum spanning tree $S = (V, F')$ such that $F \cap F'$ is maximal. Choose $f \in F' \setminus F$ and let C be the cycle induced by f. $S - f$ is disconnected, so there is $e \in F \cap C$ such that $S - f + e$ is a tree.

By assumption $w(f) \geq w(e)$. If $w(e) = w(f)$, then $w(S-f+e) = w(S) - w(f) + w(e) = w(S)$ and $S - f + e$ is a minimum spanning tree that has one more edge in common with T, contradicting our choice. So $w(f) > w(e)$, but then $w(S - f + e) < w(S)$, and S was not minimum. So T is a minimum spanning tree. □

6.1 Kruskal's Algorithm

Intuitively it is advantageous to include the cheapest edges of G in the minimum spanning tree. This is certainly not always possible, as they may lie on a cycle, and we need to make a choice in such a case. Yet, using this idea, one possible approach is the following.

> Start with the graph $T(V, \varnothing)$ and successively pick the edge in G with lowest weight available and not yet in T that does not create a cycle in T.

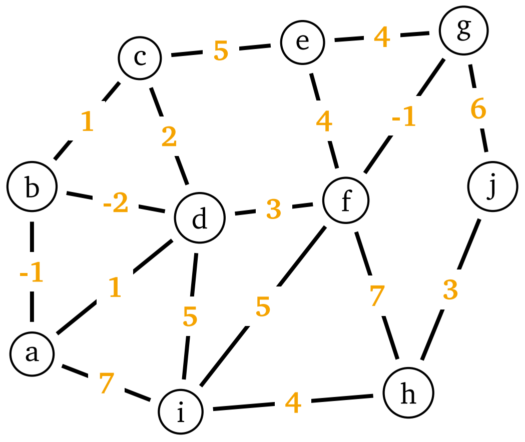
Clearly, this terminates, as G has only finitely many edges, and it produces a spanning tree by construction. The surprising property of this simple approach is the fact that it actually also produces a *minimum* spanning tree. This is *Kruskal's algorithm*. Its pseudo-code is in Algorithm 6.1. Alternatively, we can start with the full graph and remove edges until we have a tree. In this case we should look at edges in the order of *decreasing* weight.

> Start with the graph $T(V, E)$ and successively pick the edge in G with highest weight available still contained in T whose removal does not disconnect T.

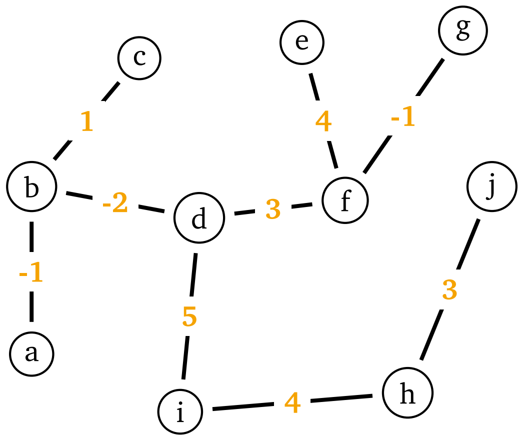
We will see that we again obtain a *minimum* spanning tree. This the *Dual Kruskal-Algorithm* shown in Algorithm 6.2.

Example. Consider the graph in Figure 6.1(a). The sorted list of edges is given in the following table.

sorted edges	bd	ab	fg	ad	bc	cd	df	hj	ef	eg	hi	ce	di	fi	gj	ai	fh
edge picked	✓	✓	✓		✓		✓	✓	✓		✓		✓				



(a) A graph.



(b) A minimal spanning tree of the graph obtained with Kruskal's Algorithm.

Figure 6.1: A graph with a minimum spanning tree obtained by Kruskal's algorithm.

The second row in the table marks the edges that are chosen by the algorithm that looks at the edges from left to right. We obtain the spanning tree in Figure 6.1(b).

Theorem 6.5. *Let G be a connected graph. Both Kruskal's Algorithm and the Dual of Kruskal's Algorithm construct a minimum spanning tree.*

Proof. Both algorithms construct a spanning tree. We need to check that it is minimal.

We consider the Kruskal's Algorithm first. Let $f_j \in F$ be the edge chosen as j^{th} edge in the algorithm, for $1 \le j \le n-1$. For some $1 \le k \le n-1$ let C the cut in G induced by f_k. Then no edge $f \in C$ creates a cycle in $T - f_k$, so also not in the subgraph with edges $f_1, \ldots, f_{k-1}$. Hence, $w(f_k) \le w(f)$ for any $f \in C$, as f_k is chosen. So T is minimal by Proposition 6.3.

Now for the dual. Label the edges removed from T by $f_1, f_2, \ldots, f_{m-n+1}$ in the order we delete them to T during the algorithm. Let C be the cycle induced by f_k in $T + f_k$. Then $w(f_k) \ge w(f)$ for any $f \in C$ by the algorithm. But then T is minimal by Proposition 6.4. □

Now let us consider the running time of the algorithms. Both algorithms have two distinctive parts that contribute to the running time. We

(i) have to sort the list of edges according to their weight, and

(ii) we must decide whether adding or removing an edge creates a cycle in T or makes T disconnected, respectively.

We will see that an efficient implementation of the second part requires more work than the first.

Proposition 6.6. *Let $G = (V, E)$ be a graph with n nodes and m edges. A naive implementation of the (Dual) Kruskal Algorithm runs in time $\mathcal{O}(m^2)$.*

Proof. Using MergeSort we can sort, by Theorem 4.4, the edges by weight in time $\Theta(m \log m)$. We can check whether $T + e$ is acyclic or still connected using either BFS or DFS. Both algorithms require time $\mathcal{O}(n + m)$, and we have to run them at most m times.

So the total running time is in $\mathcal{O}(m \log m + m(m + n)) = \mathcal{O}(m^2)$, as $n \leq m + 1$ for a connected graph. □

Let us now concentrate on Kruskal's algorithm and discuss options to improve the running time. Instead of running *BFS* or *DFS* to check whether $T + e$ is still acyclic, we can maintain a list $(c_v)_{v \in V}$ with elements $c_v \in V$ that contain a representative for the connected component that contains v. Initially, each node is its own component, so $c_v = v$. $T + e$ for $e = uv$ is still acyclic if and only if $c_v \neq c_u$. Whenever we add an edge to T we have to update the representative of one component, which requires us to check every c_v for $v \in V$. This can be done in time $\mathcal{O}(n)$, and we need at most $n - 1$ updates until all nodes are in the same component. So the overall running time is $\mathcal{O}(m \log m + n^2)$. For graphs with many edges this is dominated by the first summand, for graphs with few edges by the second.

In the latter case we can improve if we organize the check whether $T + e$ is acyclic in a better way. Instead of updating the complete list $(c_v)_{v \in V}$ when we connect two components, we organize this list in a tree-like fashion. In each component we maintain a spanning tree with a distinguished root node, which is the representative of this component, and let c_v point to its parent in this tree, and the root to itself. For a given node u we find the representative r by following the parent relation until we are at the root. So we start with $r = u$ and replace r with c_r until $r = c_r$.

If we now want to check whether two nodes u, v are in the same component, we determine and compare their roots r_u and r_v. If we add the edge $e = uv$, then we set $c_{r_u} = r_v$ or vice versa. This can be done in constant time.

Finding the root is linear in the height of the tree, *i.e.,* the longest path from any node to its root. Hence, we need to control this height. If we always attach a tree with smaller height to the one with larger height, the height of a tree can only increase if we merge trees of the same height. With this condition a tree of height h contains at least 2^h nodes, so $h \leq \log_2(n)$. Hence, we can check in time $\mathcal{O}(\log n)$ whether u and v are in the same component.

The overall running time becomes $\mathcal{O}(m \log m) + \mathcal{O}(m \log n) = \mathcal{O}(m \log n)$, as $m \leq n^2$. As sorting has also a lower bound of $\Omega(m \log m)$, we cannot improve further, This finally proves the following theorem.

Theorem 6.7. *Kruskal's algorithm can be implemented to run in time $\mathcal{O}(m \log m)$.* □

Data structures that implement the approach we have developed above to check for containment in the same partition and merge two sets of this partition are called

Algorithm 6.3: Kruskal Algorithm

```
Input  : connected graph $G = (V, E)$
         weight function $w : E \to \mathbb{R}$
Output : minimum spanning tree $T = (V, F)$ of $G$
1  $T \leftarrow (V, \emptyset)$
2  for $v \in V$ do
3      $c_v \leftarrow v$                  // parent in tree
4      $h_v \leftarrow 0$                  // height of tree
                                              // rooted in $v$
5  sort $E = \{e_1, \ldots, e_m\}$ such that
     $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
6  for $j = 1, \ldots, m$ do
7      $u, v \leftarrow$ endpoints of $e_j$
8      $r_u \leftarrow \text{Find}(u)$
9      $r_v \leftarrow \text{Find}(v)$
10     if $r_u \neq r_v$ then
11         $\text{Union}(u, v)$
12         $T \leftarrow T + e_j$
```

Algorithm 6.4: Find(v)

```
1  while $c_v \neq v$ do
2      $v \leftarrow c_v$;
3  return $v$;
```

Algorithm 6.5: Union(u,v)

```
1  $r \leftarrow \text{Find}(u)$;
2  $s \leftarrow \text{Find}(v)$;
3  if $h_r \leq h_s$ then
4      $c_r \leftarrow s$;
5      $h_s \leftarrow \max(h_s, h_r + 1)$;
6  else
7      $c_s \leftarrow r$;
8      $h_r \leftarrow \max(h_r, h_s + 1)$;
```

union-find structures. The *Find* method returns the representative of the set, and the *Union* method merges two sets. A full description of Kruskal's algorithm with these improvement is given in Algorithm 6.3.

Remark 6.8. Kruskal's algorithm is an example of a "*greedy algorithm*". The basic characteristic of greedy type algorithms is that such an algorithms builds up a solution iteratively and in each round of the iteration it does the locally best choice without consideration of future decisions.

In our case of Kruskal's algorithm we chose, in each iteration, the edge with smallest possible weight that we can add. The above argument shows that this locally optimal choice indeed does not block choices in later steps that might lead to a better solution.

6.2 Prim's Algorithm

Kruskal's Algorithm always chooses the edge with the globally smallest weight. We can instead also try to take a *local approach* by growing a connected component starting from some node u. We maintain a partial tree on s subset W of the node set V of G. In each iteration we choose the edge of minimal weight in the cut induced by W. By Proposition 6.3 this leads to a minimum spanning tree. The algorithm is shown in Algorithm 6.6.

Example 6.9. Consider again the graph shown in Figure 6.1(a). On this graph Prim's Algorithm performs the steps given in Table 6.1, if we start with the node a. Each row gives the current node set W, the edges in the cut, and the edge e chosen from the cut. We again obtain the minimal spanning tree shown in Figure 6.1(b).

Algorithm 6.6: Prim's Algorithm (simple version)

```
Input  : connected graph $G = (V, E)$
         weight function $w : E \to \mathbb{R}$
         root node $r \in V$
Output : minimal spanning tree $T = (V, F)$
1 $W \leftarrow \{r\}$                         // current node set of $T$
2 $F \leftarrow \emptyset$                         // current edge set of $T$
3 while $W \neq V$ do
4     $e \leftarrow$ edge of minimal weight in $C_G(W)$
5     $v \leftarrow e \cap V \setminus W$
6     $W \leftarrow W \cup \{v\}$
7     $F \leftarrow F \cup \{e\}$
```

Proposition 6.10. *Prim's algorithm computes a minimum spanning tree in a connected graph.*

Proof. This is a direct consequence of Proposition 6.3. Indeed, by construction, each edge e has minimum weight in the cut it induces. □

As for Kruskal's algorithm the running time depends on the implementation. In a naive implementation we find the edge of minimal weight in the cut by checking each edge. If we maintain an array that records whether a ndoe is already added to the partial tree, then we can check in constant time whether an edge is in the cut, so we can find the edge with minimal weight in the cut in time $\mathcal{O}(m)$. We add $n-1$ edges in total, we obtain a running time of $\mathcal{O}(nm)$.

Proposition 6.11. *Prim's Algorithm can be implemented to run in time $\mathcal{O}(mn)$.* □

Using an appropriate data structure we can considerably improve on this. In the

W	cut	e
{a}	{ab,ad,ai}	ab
{a,b}	{ad,ai,bc,bd}	bd
{a,b,d}	$\{ai, bc, cd, df, di\}$	bc
{a,b,c,d}	{ai,ce,di,df,di}	df
{a,b,c,d,f}	{ai,ce,di,fe,fg,fh,fi}	fg
{a,b,c,d,f,g}	{ai,ce,di,fe,fh,fi,ge,gj}	ef
{a,b,c,d,e,f,g}	{ai,di,fh,fi,gj}	di
{a,b,c,d,e,f,g,i}	{fh,gj,hi}	hi
{a,b,c,d,e,f,g,h,i}	{hj,gj}	hj
{a,b,c,d,e,f,g,h,i,j}	∅	∅

Table 6.1: Prim's Algorithm on the graph of Figure 6.1(a).

Algorithm 6.7: Prim's Algorithm

```
Input  : connected graph $G = (V, E)$,
         weight function $w : E \to \mathbb{R}$,
         root node $r \in V$
Output : minimal spanning tree $T = \big(V, \{\{v, \text{parent}(v)\} \mid v \in V \setminus \{r\}\}\big)$ of $G$
init $Q$                                          /* priority queue */
foreach $v \in V$ do
    parent($v$) $\leftarrow$ nil
    $\text{dist}(v) \leftarrow \infty$
    Insert($Q, v, \text{dist}(v)$)                /* distance from tree */
$\text{dist}(r) \leftarrow 0$
DecreaseKey($Q, r, \text{dist}(r)$)
while $Q \neq \varnothing$ do
    $v \leftarrow$ExtractMin($Q$)                 /* vertex with minimal distance */
    foreach $u \in \mathbb{N}(v)$ do
        if $u \in Q$ and $w(u, v) < \text{dist}(u)$ then
            parent($u$) $\leftarrow v$
            $\text{dist}(u) \leftarrow w(u, v)$
            DecreaseKey($Q, u, \text{dist}(u)$)
```

algorithm we need to find the edge of minimum weight connecting the component already inserted in the partial tree T with its complement.

We can view this as an information on the set of vertices C in the complement, where we assign each $v \in C$ the length of the shortest edge connecting it to T, and ∞, if there is no such edge. This is a map $d : C \to \mathbb{R} \cup \{\infty\}$.

In each iteration we want the node $v \in C$ with minimal $d(v)$. Once we add v to T in the algorithm we need to update the map d.

These operations can be implemented efficiently with a *priority queue*, which implements a queue for a set of elements with an associated *priority*, which, in our case, is the weight given by d. Unlike a normal *queue*, which we have met in the Breadth-First Search Algorithms, a *priority queue* does not return the first element in the list, but the one with *highest priority* (in our case, the one with lowest weight). Such a structure usually supports (at least) the following four operations (named appropriately for our purpose, where we want the element with minimal weight)

(i) ExtractMin: find and delete the element with lowest weight

(ii) Insert: insert an element (needed during initialization)

(iii) DecreaseKey: decrease a weight (*i.e.*, modify the map d, observe that the values of d can only *decrease)*!)

An efficient implementation of such a structure (*e.g.* with a *Binary Heap)* can do all operations in time $\Theta(\log k)$, where k is the length of the queue. This algorithm is shown in Algorithm 6.7.

Theorem 6.12. *Prim's algorithm computes a minimal spanning tree and can be implemented to run in time $\mathcal{O}(m \log n)$.*

Proof. We call Insert and ExtractMin n times each, and in total we need to decrease a value of d at most m times (every time we add a node to T we may have to update its neighbors). So the total running time in an implementation with a binary heap is $\mathcal{O}(2n\log(n) + m\log(n)) = \mathcal{O}(m\log(n))$ as the graph is connected. □

Using an even more efficient implementation of a priority queue, *e.g.* with a *Fibonacci Heap*, we can obtain a running time of $\mathcal{O}(m + n\log(n))$. Observe that this running time is better than the best running time achievable with Kruskal's algorithm.

The best known running time of for the minimum spanning tree problem was achieved by Chazelle[1]. He proved that one can do this in $\mathcal{O}(m\alpha(m))$. Here the *inverse Ackermann function* $\alpha(m)$ is an extremely slow growing function (e.g., $\alpha(9876!) = 5$).

* 6.3 Matroids

The observation that *Kruskal's Algorithm* (or its Dual) successfully work in this locally greedy fashion has its origin in a quite general structure, for which spanning trees are just one example. We want to explain this in this section. We will introduce *matroids* and show that spanning trees in graphs are a special instance of this.

Definition 6.13. A *matroid [Matroid]* is a pair $\mathcal{M} := (S, \mathcal{I})$ of a finite set S and a subset $\varnothing \subsetneq \mathcal{I} \subseteq 2^S$ with the following properties:

(i) If $I \in \mathcal{I}$ and $J \subseteq I$, then $J \in \mathcal{I}$.

(ii) If $I, J \in \mathcal{I}$ and $|I| < |J|$, then there is $j \in J \setminus I$ such that $I \cup \{j\} \in \mathcal{I}$.

The sets in I are called *independent [unabhängig]*. All other sets in 2^S are called *dependent [abhängig]*.

Here are some examples.

(i) Let $S := \{1, \ldots, n\}$ for some $n \in \mathbb{Z}_{\geq 0}$ and $1 \leq k \leq n$. Let $\mathcal{I}$ be the set of all subsets of S with at most k elements. Then $U_k^n := (S, \mathcal{I})$ is a matroid, the k-uniform matroid.

(ii) Let V be a vector space over a finite field, and $\mathcal{I}$ the collection of sets of linearly independent vectors in V. Then $\mathcal{M}_V(V, \mathcal{I})$ is a matroid. The second property of a matroid follows from *Steinitz' Exchange Theorem*.

(iii) Let $G := (V, E)$ be a connected graph, and $\mathcal{I}$ the sets of edges of *forests* in G. Then $\mathcal{M}_{\text{forest}}(G) := (E, \mathcal{I})$ is matroid.

[1]Bernard Chazelle. "A minimum spanning tree algorithm with inverse-Ackermann type complexity". In: *J. ACM* 47.6 (2000), pp. 1028–1047. DOI: 10.1145/355541.355562.

Algorithm 6.8: Greedy Algorithm for Matroids

```
Input  : matroid $\mathcal{M} = (S, \mathcal{I})$
         weight function $w : S \to \mathbb{R}$
Output : minimum basis $B$ of $\mathcal{M}$
1 $B \leftarrow \emptyset$
2 sort $S = \{s_1, \ldots, s_m\}$ such that $w(s_1) \leq w(s_2) \leq \cdots \leq w(s_m)$
3 for $j = 1, \ldots, m$ do
4     if $B + s_j$ is independent then
5         $B \leftarrow B + s_j$
```

Definition 6.14. Let $\mathcal{M} := (S, \mathcal{I})$ be a matroid. A (inclusion-)maximal independent set B is a *basis [Basis]* of $\mathcal{M}$ (*i.e.,* $B \in \mathcal{I}$ and there is no $I \in \mathcal{I}$ with $B \subsetneq I$).

The bases of $\mathcal{M}_V$ are the bases of V in the sense of linear algebra. The bases of $\mathcal{M}_{\text{forest}}(G)$ are the spanning trees in G. It follows immediately from the second property of a matroid that all bases have the same size r. This is called the *rank [Rang]* of the matroid.

Here is another characterizations of a matroid.

Proposition 6.15. *Let S be a finite set and $\emptyset \subsetneq \mathcal{B} \subseteq 2^S$. Then $\mathcal{B}$ is the set of bases of a matroid if and only of if for each $B, B' \in \mathcal{B}$ and some $x \in B \setminus B'$ there is $y \in B' \setminus B$ such that $B - x + y \in \mathcal{B}$.*

For our example of linearly independent sets in a vector space over a finite field, this is precisely the *Steinitz Exchange Theorem*. There are further equivalent definitions, *e.g.* in terms of all *minimally non-independent* sets (the *circuits [Kreise]*, *i.e.,* sets that are not independent, but all proper subsets are).

A *weight function* on a matroid $\mathcal{M} = (S, \mathcal{I})$ is a function $w : S \longrightarrow \mathbb{R}$ that assigns a *weight* to each element in S.

A *minimum basis* of a matroid $\mathcal{M}$ is a basis b such that $w(B) := \sum_{b \in B} w(b)$ is minimal among all bases of $\mathcal{M}$. Algorithm 6.8 shows an algorithm that claims to produce a minimal basis.

Algorithms of this type are called *greedy algorithms [Greedy-Algorithmen]*, as they always do the locally best choice disregarding any possible future choices.

Theorem 6.16. *Let $\mathcal{M} + (S, \mathcal{I})$ be a matroid and $w : S \longrightarrow \mathbb{R}$ a weight function. Then Algorithm 6.8 produces a minimum basis of $\mathcal{M}$.*

Proof. Let r be the rank of $\mathcal{M}$. Clearly, the algorithm returns a basis, so it has size r. Let $s_1, s_2, \ldots, s_r$ be the order in which the elements are added to B. Assume that B is not minimum, and choose a minimum basis C that coincides with B on the longest initial subsequence $s_1, \ldots, s_k$ of elements (*i.e.,* no minimum basis contains $s_1, \ldots, s_{k+1}$).

By the second property of a matroid, there is $s \in C$ such that $C - s + s_{k+1}$ is independent (and a basis, as it has r elements). But then $w(s) \geq w(s_{k+1}$ as the algorithm has chosen s_{k+1} and not s.

If $w(s) = w(s_{k+1})$, then $C - s + s_{k+1}$ is minimum and coincides with B on $s_1, \ldots, s_{k+1}$, contradicting the choice of C. But if $w(s) > w(s_{k+1})$, then $w(C - s + s_{k+1}) = w(C) - w(s) + w(s_{k+1} < w(C)$, so C was not minimum, again a contradiction. So B is a minimum basis. $\square$

If we apply this to $\mathcal{M}_{\text{forest}}(G)$ we obtain Kruskal's algorithm for minimum spanning trees. Yet, there are many more examples of matroids, where this simple greedy algorithm produces a minimum set in the appropriate interpretation.

One can even prove that a greedy algorithm in the given form produces the correct result *if and only if* the structure it is applied to is a matroid.

7 Shortest Paths

Contents

The Breadth-First Search Algorithm discussed in Chapter 5 computes distances from a given node r to all other nodes if all edges have the same weight (*i.e.,* if the length of a path is proportional to the number of edges). This was a consequence of the observation that the function level, which gives the number of edges in the computed spanning tree to the root, coincides with the length of a shortest path, We proved this in Theorem 5.10. This approach fails if we assign different lengths or weights to the edges and ask for the distance defined by the sum of the weights along a path. You can already observe this in a triangle.

In this chapter we discuss approaches to solve this more general shortest-path problem. In many applications where one is interested in shortest paths the graphs considered are directed. Think, *e.g.*, of routes in street networks, where you may have one-way streets or crossings at which only some turns are allowed. We will thus switch to *directed* graphs in this chapter. We introduced basic notions for these graphs already in Definition 2.4 and add some more in the next section.

We discuss two methods to solve the shortest path problem, *Dijkstra's Algorithm* in Section 7.3 and the *Bellman-Ford Algorithm* in Section 7.4. The first is a *greedy-like* algorithm that builds up a shortest path tree from a root to all other nodes. It extends the *Breadth-First Search Algorithm* of Chapter 5, which computes such a tree if all edges have the same weight. Its approach to deal with varying weights is similar to *Prim's Algorithm* discussed the previous Chapter 6. The second algorithm picks up the idea of the *Longest Ascending Subsequence Problem* from Chapter 3 to construct larger optimal solutions from shorter ones.

Before we can start with the actual algorithms we discuss directed graphs in more detail in Section 7.1 and introduce the shortest path problem in Section 7.2.

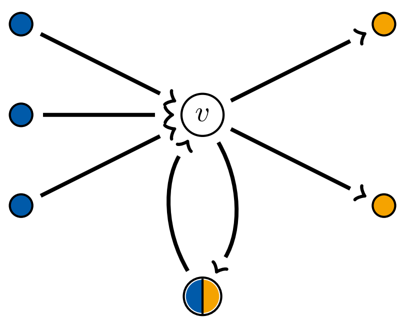

v
in-neighbourhood $\mathrm{N}^-(v)$
in-degree $\deg^-(v) = 4$
out-neighbourhood $\mathrm{N}^+(v)$
out-degree $\deg^+(v) = 3$

Figure 7.1: In- and out-neighborhood of a node.

7.1 Directed Graphs

Recall that a directed graph is a pair $G = (V, E)$ of a set of nodes V and a subset E of all *ordered* pairs $e = (u, v)$ of nodes $u, v \in V$ with $u \neq v$. The node u is the *tail* and v is the head of e. These directed edges are sometimes called *arcs*. For directed graphs we have to refine some notions introduced in Chapter 2.

Definition 7.1. Let $G = (V, E)$ be a *directed* graph. The *in-neighbourhood and out-neighbourhood* of a node $v \in V$ are

$$\mathrm{N}^-(v) := \{u \in V \mid (u, v) \in E\} \qquad \text{and} \qquad \mathrm{N}^+(v) := \{u \in V \mid (v, u) \in E\},$$

respectively. Correspondingly, the *in-degree* and *out-degree* are the sizes of these sets,

$$\deg^-(v) := |\mathrm{N}^-(v)| \qquad \text{and} \qquad \deg^+(v) := |\mathrm{N}^+(v)| .$$

In the same way as for undirected graphs we may extend this to sets and define

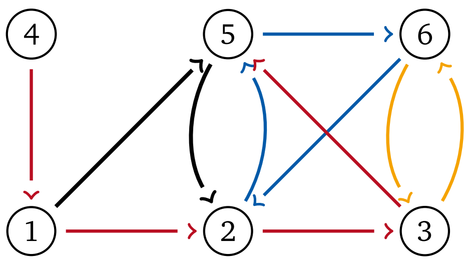
$$\begin{aligned} \mathrm{N}^-(S) &:= \{v \in V \setminus S \mid \exists u \in S : (v, u) \in E\} \quad \text{and} \\ \mathrm{N}^+(S) &:= \{v \in V \setminus S \mid \exists u \in S : (u, v) \in E\} \end{aligned}$$

for a subset $S \subseteq V$. An example of the in- and out-neighborhood of a node is shown in Figure 7.1.

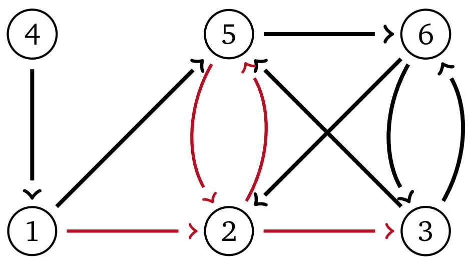
Definition 7.2. A *path* in a directed graph $G = (V, E)$ is a sequence $v_0 v_1 \ldots v_k$ of pairwise distinct nodes such that $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$, *i.e.*, edges have to point in the *right* direction. Similarly we define directed walks and trails.

A *directed cycle* is a sequence $v_0 \ldots v_k$ of nodes such that $v_0, \ldots, v_{k-1}$ are pairwise distinct, $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$, $v_0 = v_k$ and $k \geq 1$ (*i.e.*, unlike in the undirected case, we can have cycles with only two nodes and edges).

A node $v \in V$ is *reachable [erreichbar]* from a node s if there is a (directed) path from s to v. Note that reachability in directed graphs is *not symmetric*.



(a) A *directed path* $(4, 1, 2, 3, 5)$ and two *directed cycles* $(2, 5, 6, 2)$ and $(3, 6, 3)$ in the same directed graph.



(b) A *directed trail* $(1, 2, 5, 2, 3)$. Note that while the edges are pairwise distinct, this is *not* a directed path since the node 2 occurs twice.

Figure 7.2: Directed path, cycle and trail in a directed graph. Note that *e.g.* $1, 2, 3, 5$ and 6 are all reachable from 4, while 4 is not reachable from any node.

See Figure 7.2 for examples. Also connectivity becomes trickier for directed graphs. For a directed graph G we let $\widehat{G}$ be the undirected graph obtained from G by replacing each arc (u, v) with the edge $\{u, v\}$ (and deleting one edge if this procedure creates double edges).

Definition 7.3. We say that G is

- ▷ *weakly connected [schwach zusammenhängend]* if $\widehat{G}$ is connected,
- ▷ *connected [zusammenhängend]* if for any two nodes u, v at least one is reachable from the other,
- ▷ *strongly connected [stark zusammenhängend]* if there is (directed) path between any two nodes.

The *strong components [starke Zusammenhangskomponenten]* of G are the maximal strongly connected subgraphs of G.

The various notions of connectedness in directed graphs are connected by the following chain of implications:

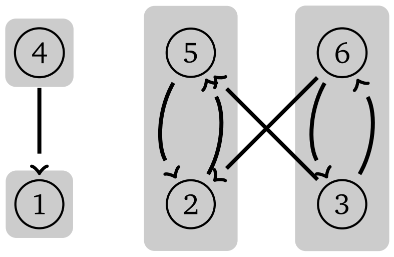
$$\text{strongly connected} \quad \Longrightarrow \quad \text{connected} \quad \Longrightarrow \quad \text{weakly connected.}$$

See Figure 7.3 for examples. The converse directions are not true in general.

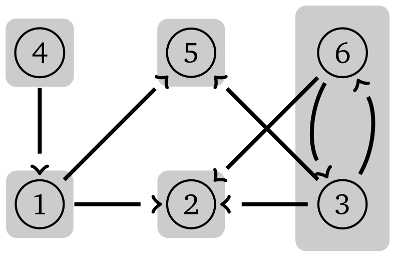
Both weakly and strongly connected induce an equivalence relation on the nodes. The first is the one we discussed in Theorem 2.12. In the same way we obtain this for strong connection, which gives us the strong components. The second notion defined is not symmetric, and thus cannot be used to define an equivalence relation.

7.2 Shortest Paths

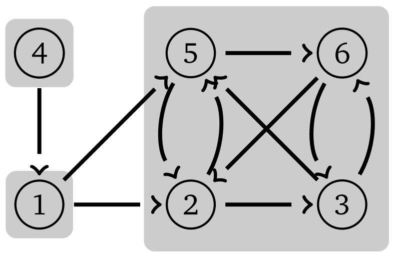
Let $G = (V, E)$ be a directed graph. As in the previous Chapter 6 we consider a *weight function [Gewichtsfunktion]* $w : E \to \mathbb{R}$. We briefly write $w(u, v)$ for $w((u, v))$ for an



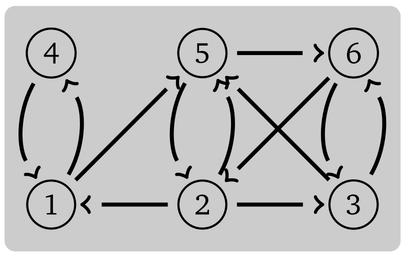
(a) A directed graph that is not connected (in any sense).



(b) A directed graph that is weakly connected but not connected. (The underlying undirected graph is connected, but there is *e.g.* no directed path between 1 and 3.)



(c) A directed graph that is connected but not strongly connected.



(d) A strongly connected directed graph.

Figure 7.3: The different notions of connectedness in directed graphs. The strong components are sketched in gray.

arc $(u, v) \in E$. The *length [Länge]* or *weight [Gewicht]* of a walk $p = v_0 v_1 \ldots v_k$ is

$$w(p) := \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Definition 7.4. Let $G = (V, E)$ be a directed graph and $w : E \to \mathbb{R}$. We define

$$d_k(u, v) := \min\{w(p) \mid p \text{ is a walk from } u \text{ to } v \text{ of length at most } k\},$$

where $\min(\emptyset) = \infty$, i.e., $d_k(u, v) = \infty$ if no walk from u to v exists.

The *(weighted) distance function [(gewichtete) Abstandsfunktion]* on G is

$$d(u, v) := lim_{k\to\infty} d_k(u, v).$$

Note that in general the functions $d_k(u, v)$ and $d(u, v)$ are *not* symmetric in u and v. The weight function w may take negative values. This is the reason for the, at least at first glance, complicated definition of the distance, which we discuss now.

If w has negative values, then it may create negative cycles in the graph as shown in Figure 7.4. Hence, the limit we take for $d(u, v)$ may be $-\infty$, and a path realizing the minimum distance does not exist. On the other hand, if the distance is finite, it is obviously realized by a path, as, whenever we would revisit a node the walk we have

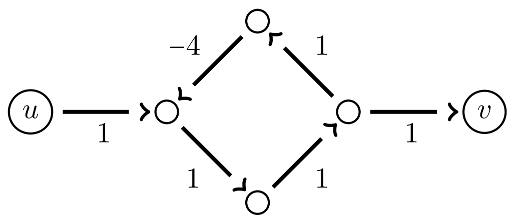


Figure 7.4: A negative cycle in a directed graph. There is no shortest walk of *finite* length between u and v.

travelled between the visits must have weight at least 0. We obtain the same result as for undirected walks and paths.

Proposition 7.5. *Let $G = (V, E)$ be a directed graph and $w : E \to \mathbb{R}$. If for some $u, v \in V$ the distance $d(u, v)$ is finite, then there is a path P from u to v such that $w(P) = d(u, v)$.* □

With this observation we can define the key notion of the whole chapter.

Definition 7.6. A *shortest path [kürzester Weg]* between u and v is a path p from u to v of length $d(u, v)$, if this is finite.

See Figure 7.5 for an example. A path in a graph with n nodes has at most $n - 1$ edges. Thus, if the distance between two nodes is finite, then the limit in the definition stabilizes after at most $n - 1$ steps.

Corollary 7.7. *If $d(u, v)$ is finite, then $d(u, v) = d_{n-1}(u, v)$.* □

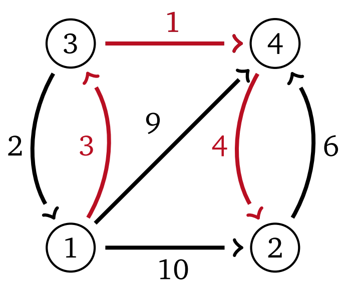
Shortest paths have a similar structure as the subsequences that we considered in Section 3.5. As for subsequences, we will show with the next proposition that any sub-path of a shortest path $u = v_0v_1 \ldots v_k = v$ between nodes $u, v \in V$, *i.e.,* the portion of the path between two nodes v_i, v_j for $i < j$ it contains, is a shortest path between v_i and v_j. So again, *sub-structures* of optimal structures are optimal.

Proposition 7.8. *Let $G = (V, E)$ be a directed graph and $w : E \to \mathbb{R}$.*

If $u, v \in V$ and $P : u = v_0v_1 \ldots v_k$ is a shortest path between u and v with k edges, then $v_iv_{i+1} \ldots v_j$ is a shortest path between v_i and v_j, for any $0 \le i \le j \le k$.

Moreover, $d(v_i, v_j) = d_{j-i}(v_i, v_j)$.

Proof. Assume that there are i, j such that there is a shorter path P' between v_i and v_j. We can replace the portion $v_iv_{i+1} \ldots v_j$ of P with that path. This gives a walk of shorter



(a) A directed graph with a weight function.

$u \setminus v$	1	2	3	4
1		8	3	4
2	∞		∞	6
3	1	5		1
4	∞	4	∞	

(b) The distance function d.

Figure 7.5: A directed graph and its distance function. The table always gives the length $d(u, v)$ of a shortest path from u to v if such a path exists. Note in particular the shortest path need not be given by the path with the fewest number of edges; *e.g.* the shortest path between the nodes 1 and 2 is $1, 3, 4, 2$ with weight 8 while the direct edge $(1, 2)$ has weight 10. Also note that the distance function need not be symmetric, *e.g.* $d(1, 3) = 3 \neq d(3, 1) = 2$.

length between u and v, which we can reduce to a path of at most the same length. Hence, P was not a shortest path.

For the last statement observe that the path $v_i v_{i+1} \dots v_j$ has $j - i$ edges. □

This observation is at the heart of the *Bellman-Ford Algorithm* that we discuss in Section 7.4, but it is also needed in the proof of correctness for *Dijkstra's Algorithm* in the next section.

A shortest path between two nodes $u, v \in V$ may visit all nodes in the graph. We have seen an example in Figure 7.5. If we know a shortest path between u and v, then, by the previous proposition, we also know the shortest path from u to any other node in the graph. Hence, the problem of computing a shortest path between a node and all other nodes in the graph has the same worst case running time as the seemingly simpler problem of computing the shortest path just between a given pair of nodes. We can thus directly consider the problem to compute all distances from a given root node.

(Single Source) Shortest Path Problem.

input: a directed graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$, and a node $r \in V$.

problem: compute all distances $d(r, v)$ for $v \in V$.

We may instead or additionally ask for an explicit shortest path from r to any other node. The problem to compute just the distance between a pair of nodes is the *single pair shortest path problem*. Computing the full matrix $d(u, v)$ for all $u, v \in V$ is the *all pairs shortest path problem*.

We discuss two algorithms for the single source shortest path problem in the next two Sections 7.3 and 7.4. The first algorithm, *Dijstra's Algorithm*, is pretty similar to *Prim's algorithm* that we discussed in Section 6.2. It grows a tree of shortest paths around the root r. It is faster than the second algorithm we discuss, but it fails if weights can be

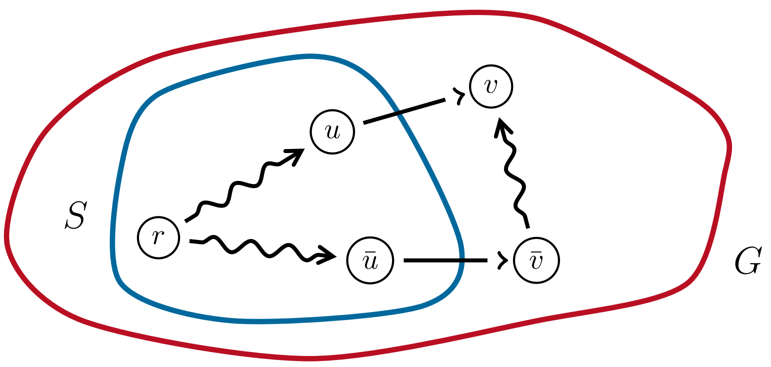


Figure 7.6: Extending the set of known shortest paths

negative (it needs the property that the weight of an initial part of a walk is monotone). Also, efficiently implementing the algorithm is an involved task.

The second algorithm, the *Bellman-Ford Algorithm*, is conceptionally much simpler and can detect whether there is a negative cycle in the graph, but it is, at least theoretically, less efficient. In Section * 7.5 we also briefly discuss a solution to the all pairs shortest path problem.

7.3 Dijkstra's Algorithm

Let us assume for a moment that all weights on arcs are strictly positive in a directed graph $G = (V, E)$ with a given root node r. Let $S \subseteq V$ be a set of nodes containing r. Let $v \in V \setminus S$ be a node with minimal distance D to r among all nodes in $V \setminus S$. Then any path from r to v that uses another node $\bar{v} \in V \setminus S$ has length $\ell > d(r, \bar{v}) \geq D$, as weights are positive and $\bar{v}$ has at least distance D from r, see Figure 7.6. Hence, the first node u on a shortest path from r to v is in the set S.

This suggests the following approach to determine distances. We start with r and continue growing a shortest path tree S around r by always adding the node with minimal distance that is not already in S. Yet, to choose a node with minimal distance from r outside S, it seems that we need to know already the distances we want to compute. We can resolve this with the above observation, where we have seen that for the node with minimal distance from r the next node on the path is in S. Hence, we can find the node with minimal distance by just checking all nodes reachable via an arc in the cut given by (the nodes of) S in G. Let us call this set of nodes the *boundary* B_S of S. For each node $v \in B$ we assign a *tentative distance* $D(u)$ as the minimum over $d(r, u) + w(u, v)$ for all u in the *in-neighborhood* $\mathrm{N}^-(v) \cap S$. This is an upper bound for the true distance (as it is the length of *some* path from r to v). Our considerations now imply that for the node $v \in B_S$ with minimal $D(v)$ (among all $v \in B_S$) the value $D(v)$ is actually the true distance, *i.e.*, $D(v) = d(r, v)$.

This is essentially *Dijkstra's Algorithm*. Observe the similarity to *Prim's Algorithm* in finding a minimal element in the cut induced by an already processed set S. We will see that we can relax the assumption of positive weights and allow arcs of weight 0.

For a given root $r \in V$ we will compute with this method the distances $D(v) := d(r, v)$

Algorithm 7.1: Dijkstra's Algorithm

```
Input  : (directed) graph  $G = (V, E)$,
         weight function   $w : E \longrightarrow \mathbb{R}_{\geq 0}$,
         root node         $r \in V$
Output : distance  $D : V \longrightarrow \mathbb{R}_{\geq 0}$,
         parent    $\mathsf{parent} : V \longrightarrow V \cup \{\mathsf{nil}\}$
foreach $v \in V$ do
    $D(v) \leftarrow \infty$
    $\mathsf{parent}(v) \leftarrow \mathsf{nil}$
    $\mathsf{visit}(v) \leftarrow$ FALSE
$D(r) \leftarrow 0$
$Q \leftarrow V$                          // priority queue, weights from $D$
while $Q \neq \varnothing$ do
    $u \leftarrow \mathsf{ExtractMin}(Q)$               // get the node $v$ with
                                  // current minimal $D(v)$ left in $Q$
    $\mathsf{visit}(u) \leftarrow$ TRUE
    foreach $v \in N^+(u)$ do
        if $\mathsf{visit}(v)$ = FALSE then
            if $D(u) + w(u, v) < D(v)$ then
                $\mathsf{parent}(v) \leftarrow u$
                $D(v) \leftarrow D(u) + w(u, v)$
                $\mathsf{DecreaseKey}(Q, v, D(v))$   // adjust the priority of $v$
```

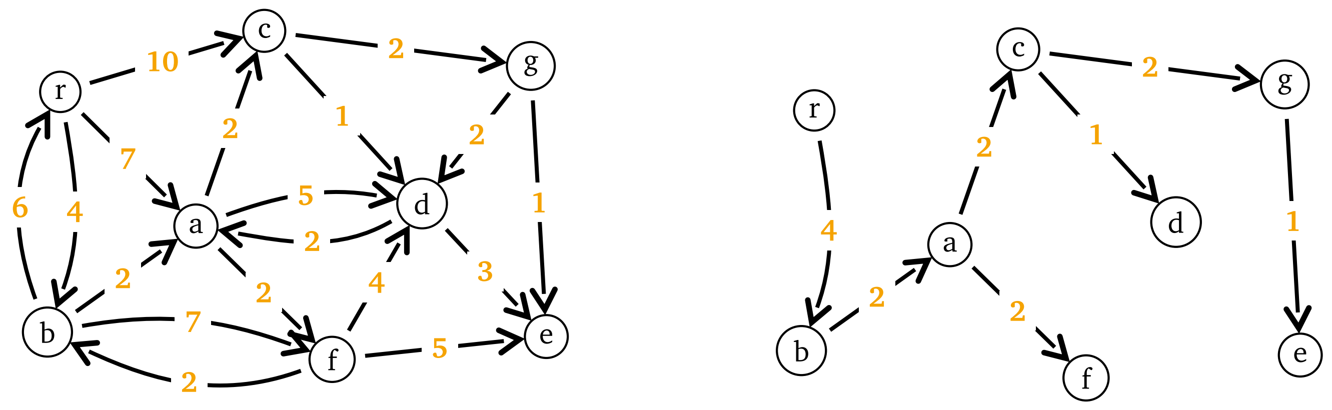
and a *shortest-path-tree [Kürzeste-Wege-Baum]* S rooted in r, *i.e.,* a tree such that the unique path from r to a node v in the tree is a shortest path from r to v in the graph G. The tree is again, in the same way as for Breadth-First Search and Depth-First Search, given by a function $\mathsf{parent} : V \to V \cup \{\mathsf{nil}\}$ that assigns the parent node in the tree to any node $v \neq r$ that is reachable from r in G.

We use the *tentative distance* introduced above for all nodes in G, not just for the boundary B_S. The value $D(v)$ will always be an upper bound to $d(r, v)$, and the true distance for $v \in S$, *i.e.,* for nodes in the growing tree S (of which we keep track by setting its nodes *visited*). Instead of computing B_S explicitly and then obtaining $D(v)$ for $v \in B_S$ by iterating over $N^-(v)$ we can instead iterate over $N^+(u)$ for all nodes in $u \in S$ and update the value of $D(v)$ for the nodes found. With this approach we do not have to store the boundary explicitly, and if we store the current value of $D(v)$ for each node, we even have to run over each neighborhood only once (we initialize $D(v)$ to ∞ or some large value, so that nodes neither in S nor in B_S do not interfere). The algorithm in pseudo-code is given in Algorithm 7.1.

Let us do an example before we prove its correctness and determine the running time.

Example 7.9. Consider the directed graph G shown in Figure 7.7(a) with the weight function given by the arc labels. We apply Dijkstra's Algorithm for the root node $r = 1$ to obtain distances from r to all other nodes.

Table 7.1 shows the updates on the distance function and the priority queue after each traversal through a complete neighborhood of a node in Dijkstra's algorithm. In



(a) A directed graph with edge weights. (b) A shortest path tree for node r.

Figure 7.7: An example for Dijkstra's algorithm.

the column of each node we show the pair $(d(v), \mathsf{parent}(v))$ if the values change.

We obtain the shortest-path-tree (given by parent) shown in Figure 7.7(b).

Theorem 7.10. *Let $G = (V, E)$ be a directed graph, $w : E \to \mathbb{R}_{\geq 0}$, and $r \in V$. Dijkstra's algorithm with root r determines $D(v) := d(r, v)$ for all v and returns a shortest-path-tree via* $\mathsf{parent} : V \to V \cup \{nil\}$.

We introduce some more notation. For a subgraph $H = (W, F)$ of G let $w_F : F \to \mathbb{R}_{\geq 0}$ be the weight function of G restricted to F and d_H the distance function of H with these weights. With this choice, distances between nodes in G and H coincide if there is a shortest path in G between the nodes whose edges are entirely in H.

Proof. The algorithm removes one node from Q in each iteration, so it terminates in finite time.

We consider three additional variables S, G_v, and $G_{v,w}$ throughout the algorithm,

It.	u	r	a	b	c	d	e	f	g	Q
Init		(0,0)	(∞,0)	(∞,0)	(∞,0)	(∞,0)	(∞,0)	(∞,0)	(∞,0)	{r,a,b,c,d,e,f,g}
1	r		(7,r)	(4,r)	(10,r)					{a,b,c,d,e,f,g}
2	b	-	(6,b)	-				(11,b)		{a,c,d,e,f,g}
3	a	-	-	-	(8,a)	(11,a)		(8,a)		{c,d,e,f,g}
4	f	-	-	-			(13,f)	-		{c,d,e,g}
5	c	-	-	-	-	(9,c)		-	(10,c)	{d,e,g}
6	d	-	-	-	-	-	(12,d)	-		{e,g}
7	g	-	-	-	-	-	(11,g)	-	-	{e}
8	e	-	-	-	-	-	-	-	-	∅

Table 7.1: Updates of priority queue and distance function in the course of the algorithm

which we define as

$$S := V \setminus Q \qquad G_v := G \cap (S \cup \{v\}) \qquad G_{v,w} := G \cap (S \cup \{v,w\}) \,,$$

where $G \cap W$ for some $W \subseteq V$ is the subgraph of G induced by W. So G_v is the graph induced by the node set $S \cup \{v\}$ and $G_{v,w}$ the graph induced by $S \cup \{v,w\}$.

We claim that throughout the algorithm the following three statements are true:

(i) For all $v \in V$: $D(v) = d_{G_v}(r,v)$

(ii) For all $v \in S$: $D(v) = d(r,v)$

(iii) For all $v \in S \setminus \{r\}$: $(\mathsf{parent}(v), v)$ is the last edge on a shortest path from r to v

All three claims are certainly true before we start the loop in line 7. We need to track changes to D whenever we extract a node u from Q (and thus add a node to S).

Let us first look at u. The value of $D(u)$ is not changed, but u is moved from Q to S. Let $r = v_0 v_1 \ldots v_j = u$ be a shortest r-u-path p and v_i the first node on this path that is not in S. By assumption we have

$$D(v_i) \ \geq \ \min_{v \in Q} D(v) \ = \ D(u)\,.$$

All weights are non-negative, so by Proposition 7.8 and the fact that the path $v_0 v_1 \ldots v_i$ lies entirely in G_{v_i} we get

$$w(p) \ \geq \ d(r, v_i) \ = \ d_{G_{v_i}}(r, v_i) \ = \ D(v_i)\,,$$

where the latter equality follows from (i). Together with the previous inequality we get that

$$d(r,u) \ = \ w(p) \ \geq \ D(v_i) \ \geq \ D(u) \ = \ d_{G_u}(r,u) \ \geq \ d(r,u)\,.$$

Hence, we have equality throughout and (ii) is maintained when we remove u from Q and add it to S. By construction, the value of $D(u)$ was set using $(\mathsf{parent}(u), u)$ as the last arc, so (iii) is maintained.

We still need to look at the neighborhood of u and check that (i) is maintained. Combining (i) and (ii) we obtain for all $v \in S$

$$d_{G_v}(r,v) \ = \ d(r,v)\,.$$

Hence, there is a shortest r-v-path that is entirely in G_v.

Let now $v \in \mathrm{N}^+(u) \cap Q$ and p a shortest r-w-path in $G_{u,v}$. If that path does not use the node u, then clearly (i) is maintained. If it does, then, using the previous consideration, we can assume that the edge (u,v) is the last edge on the path. So

$$\begin{aligned} d_{G_{u,v}}(r,v) \ &= \ \min\left(d_{G_u}(r,u) + w(u,v), d_{G_v}(r,v)\right) \\ &= \min\left(D(u) + w(u,v), D(v)\right) \end{aligned}$$

Hence, we update the values of D precisely in such a way that (i) is maintained when adding u to S. Further, if the path to v via u is indeed shorter we update parent to the correct new parent.

In the end Q is empty, so $S = V$, and our invariants (ii) and (iii) give the desired result. □

We consider now the running time of the algorithm. In a simple implementation we just iterate over Q to determine the node with minimal $D(u)$. This can be done in $\mathcal{O}(n)$. Each node is extracted exactly once from R, so we iterate in line 10 once over each neighborhood. This can be done in total time $\mathcal{O}(m)$. So *Dijkstra's Algorithm* can be implemented to run in time $\mathcal{O}(mn)$.

We can do considerably better if we use a proper implementation of a priority queue as we already discussed for *Prim's algorithm*. Recall that a *priority queue* is a list that supports (at least) the three operations

▷ Insert, that inserts a new element in the queue,

▷ ExtractMin, that returns the element of minimal weight (in our case $D(v)$), and

▷ DecreaseKey, that updates the priority of an entry (in our case sets the new value of $D(v)$).

Using a *binary heap* as the underlying data structure, each of the three operations can be implemented to run in time $\Theta(\log k)$, where k is the length of the queue.

In our algorithm, we insert and retrieve all n nodes in the queue, and do at most m updates on the values, so we do $2n + m$ operations on the queue.

Theorem 7.11. *The algorithm can be implemented to run in time $\mathcal{O}((m + n)\log n)$.* □

Using a *Fibonacci Heap* instead of a binary heap one can even push this down to $\mathcal{O}(m + n\log n)$.

7.4 The Bellman-Ford-Algorithm

The algorithm of Dijkstra is again an example of a more greedy-like algorithm, as in each iteration of the `while`-loop we choose the locally best improvement. This greedy choice is, however, based on the substructure property of Proposition 7.8, about which we maintain information in the priority queue Q.

We can use the property of Proposition 7.8 more directly in a *dynamic programming [dynamische Programmierung]* approach, in the same way as we have done for the subsequence problem of Section 3.5. This will lead us to an algorithm to compute shortest paths from a given root node that can handle negative weights and also detect, and stop the algorithm with a corresponding notice, if a graph has a negative cycle.

To exploit Proposition 7.8 we use a *bottom up* strategy to build the solution. We successively construct a tentative distance function in the graph, that is correct or paths that use up to k edges, for increasing k. That is, we construct the functions $d_k(r, v)$ from Definition 7.4 for a given root node r and all $v \in V$. Once we have arrived at $k = n - 1$ we know that we have the full distance function $d(r, v)$, unless we have found a negative cycle. We will see that this comes at the cost of a slower running time of $\mathcal{O}(mn)$.

Algorithm 7.2: Bellman-Ford

```
Input  : (directed) graph  $G = (V, E)$,
         weight function   $w : E \longrightarrow \mathbb{R}$,
         root node         $r \in V$
Output : FALSE if $G$ contains a negative cycle reachable from $r$, otherwise
         distance  $D : V \longrightarrow \mathbb{R}$,
         parent    $\mathsf{parent} : V \longrightarrow V \cup \{\mathsf{nil}\}$
foreach $v \in V$ do
    $D(v) \leftarrow \infty$
    $\mathsf{parent}(v) \leftarrow \mathsf{nil}$
$D(r) \leftarrow 0$

for $i \leftarrow 1, \ldots, |V| - 1$ do                         // get $d_i$
    foreach $(u, v) \in E$ do
        if $D(u) + w(u, v) < D(v)$ then
            $D(v) \leftarrow D(u) + w(u, v)$
            $\mathsf{parent}(v) \leftarrow u$

foreach $(u, v) \in E$ do                           // check for neg. cycle
    if $D(u) + w(u, v) < D(v)$ then
        return FALSE
return TRUE
```

In the algorithm we do not construct the functions d_k explicitly. As in Dijkstra's Algorithm, we use a single function $D(v)$ that is always an upper bound on the distance and refine this in each step. The proof will show that after k iterations it coincides with d_k. This algorithmic idea was found by Richard E. Bellman and Lester Ford, and independently also by Edward F. Moore. The pseudo-code of the *Bellman-Ford algorithm* is in Algorithm 7.2.

We do an example before we prove correctness and determine the running time.

Example 7.12. In Table 7.2 we show the Bellman-Ford Algorithm performed on Figure 7.7(a). The first column shows the iteration over the number of nodes. This is just a counter from 1 to 7. In each iteration we run over the edges in lexicographically increasing order. As distances of a node may change more than once per iteration we show edges with the same tail node in separate rows, if they have an incident edge that actually leads to an update. In this case we list the edges that lead to an update in the next column.

The remaining columns show the current distance and the current root node in the same way as in our example for Dijkstra's Algorithm. After the third iteration we have already found all distances, and the remaining iterations do not change anything anymore. We obtain the shortest path tree shown in Figure 7.7(b).

Let us now analyze the algorithm. We split this in several steps.

It.	tail	edges	r	a	b	c	d	e	f	g
Init			(0,0)	(∞,0)	(∞,0)	(∞,0)	(∞,0)	(∞,0)	(∞,0)	(∞,0)
1	r	ra, rb, rc		(7,r)	(4,r)	(10,r)				
2	a	ac, ad, af				(9,a)	(12,a)		(9,a)	
	b	ba		(6,b)						
	c	cd					(10,c)			(11,c)
	d	de						(13,d)		
	g	ge						(12,g)		
3	a	ac, af				(8,a)			(8,a)	
	c	cd					(9,c)			(10,c)
	g	ge						(11,g)		
4-7	no further changes									

Table 7.2: Updates of priority queue and distance function in the course of the algorithm

Lemma 7.13. *If G does not contain a cycle of negative weight then $D = d(r, v)$ for all $v \in V$ after the loop in line 6 to line 10 has finished.*

Proof. We prove by induction that after the i-th iteration $D(v) = d_i(r, v)$, *i.e.*, the distance is correct if a shortest path uses at most i edges, if such a path exists. This is clearly true for $i = 0$.

In the i-th loop the algorithm checks, for each edge e, whether a path p with at most $i - 1$ edges extended by e (if possible) has smaller weight than the currently known weight between the two endpoints of $p \cup e$. So $D(v)$ contains the correct value also after the i-th iteration.

Unless we have a negative cycle, no shortest path can use more than $n - 1$ edges, as otherwise we would visit at least one node twice. □

Theorem 7.14. *Let G be a directed graph with weight function $w : E(G) \to \mathbb{R}$. If G has no negative weight cycle then Bellman-Ford returns* TRUE *and the distance function $d(v) = d(r, v)$, otherwise it returns* FALSE.

Proof. By the previous lemma, if G has no negative weight cycle, then $D(v)$ is the distance function from r (and parent constructs the corresponding tree). At termination we have for all $(u, v) \in E$

$$D(v) = d(r, v) \leq d(r, u) + w(u, v) = D(u) + w(u, v),$$

so Bellman-Ford returns TRUE.

Conversely, assume that G contains a cycle $v_0v_1 \ldots v_k$ with $v_0 = v_k$ of negative weight that can be reached from r. Then $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$. If the algorithm returns TRUE then

$$D(v_i) \;\leq\; D(v_{i-1}) + w(v_{i-1}, v_i)$$

and summing over all i gives

$$\sum_{i=1}^{k} D(v_i) \;\leq\; \sum_{i=1}^{k} D(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

and hence

$$0 \;\leq\; \sum_{i=1}^{k} w(v_{i-1}, v_i) < 0\,.$$

This is a contradiction. □

Theorem 7.15. *The Bellman-Ford Algorithm runs in time $\mathcal{O}(mn)$.*

Proof. In line 6 and line 7 we initiate a nested loop: For each node we run over all edges. The cost of the operations in the inner loop are constant for each edge, so we obtain a running time of $\mathcal{O}(mn)$ for this part. The check in lines 10–12 can be done in $\mathcal{O}(m)$ and initialization in lines 1–3 in $\mathcal{O}(n)$, so the overall running time is $\mathcal{O}(nm)$ as claimed. □

* 7.5 All Pairs Shortest Path: Floyd-Warshall

So far we have considered *single* source shortest path problems. In this section we want to compute *all* distances in the graph, *i.e.,* we want to obtain the matrix

$$D = (d_{ij}) \qquad \text{with} \qquad d_{ij} = d(i,j).$$

One way to obtain this would be to run Dijkstra or Bellman-Ford for every node. This gives an $\mathcal{O}((n+m)n\log n)$ or $\mathcal{O}(n^2m)$ algorithm. The simple algorithm we will discuss in this section needs $\mathcal{O}(n^3)$ and works also for negative weights.

We assume that we are given a weight function $w : E \to \mathbb{R}$, possibly with negative weights but no cycles of negative weight. We extend w to $V \times V$ by $w(i,j) = \infty$ for $(i,j) \notin E$ and $w(i,i) = 0$.

Now D is built iteratively as a sequence $D^{(0)}, \ldots, D^{(n)} = D$, where $D^{(k)} = (d_{ij}^{(k)})$ gives distances from i to j such that all intermediate nodes are in $\{1, \ldots, k\}$.

Initially, $D^{(0)} = (w(i,j))$ as a path that does not use any intermediate node must be a direct arc from i to j. Further, given $D^{(k-1)}$

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}).$$

This follows as any shortest path between i and j using only nodes $1, \ldots, k$ as interme-

Algorithm 7.3: Floyd-Warshall

```
Input  : directed graph $G = (V, E)$, $V = \{1, \dots, n\}$, weight function
         $w : V \times V \to \mathbb{R}$ without negative cycles
Output : distance matrix $D = (d_{ij}^{(n)}) \in \mathbb{R}^{n \times n}$
1 for $i \leftarrow 1, \dots, n$ do
2     for $j \leftarrow 1, \dots, n$ do
3         $d_{ij}^{(0)} \leftarrow w(i, j)$
4 for $k \leftarrow 1, \dots, n$ do
5     for $i \leftarrow 1, \dots, n$ do
6         for $j \leftarrow 1, \dots, n$ do
7             $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
```

diate nodes either does not use k, so it has already been found for $D^{(k-1)}$ or we can split it into a path from i to k and from k to j. The pseudo-code is in Algorithm 7.3.

Theorem 7.16. *The algorithm is correct and runs in time $\mathcal{O}(n^3)$ (even $\Theta(n^3)$).*

Proof. Correctness of the algorithm follows from the above. The running time is dominated by the nested loop over k, i, and j in lines 4 to 7. This can be done in $\mathcal{O}(n^3)$. □

8 Flows

Contents

In this chapter we want to discuss *flows* in a *network*. This is the mathematical abstraction of various standard problems in applications of optimization:

(i) In a network of streets we want to route a given amount of traffic between a start point and a destination respecting the capacity constraints given by the amount of traffic that can run over a street.

(ii) In a network of gas pipelines we want to route gas between a gas field and a distribution terminal respecting the limits of each pipeline.

(iii) For an evacuation plan of a building we need information on how many people can leave a given floor of the building through the stair cases to an emergency exit respecting the widths of corridors and stairs.

In all following considerations we assume that we do not have isolated nodes in the graph. This implies that $n < 2m$, so $n = \mathcal{O}(m)$.

8.1 Networks and Flows

In all applications above we have specified nodes in the network, the source and the target for our routing problem, and constraints on the individual arcs in the network. The following definition collects these pieces of information in a single notion.

Definition 8.1. A (flow) *network [(Fluß-)Netzwerk]* is a tuple of a directed graph $G = (V, E)$, *source [Quelle]* node $s \in V$, a *sink [Senke]* node $t \in V$ with $s \neq t$, and a *capacity [Kapazität]* function $c : E \to \mathbb{R}_{\geq 0}$.

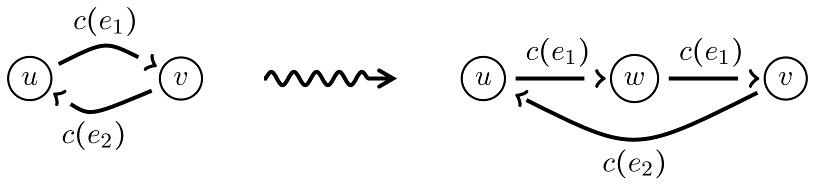


Figure 8.1: Removal of anti-parallel arcs.

If s, t, and c are known from the context we often only use G to denote the whole network. Note that more generally one uses the term *network* for directed graphs, usually with a given weight function on the edges, but not necessarily with a source and target node. In this chapter we include source and target in our definition to avoid repeating this every time.

Remark 8.2. Directed graphs may have two arcs between nodes u and v, one pointing from u to v, one pointing from v to u. Though this is no theoretical problem for the construction of flows in a network using the tools we discuss in this chapter, it makes notation and discussion more complicated.

Hence, for simplicity we assume in the following that there is at most one arc between any pair of nodes, *i.e.*, for any two nodes $u, v \in E$ at most one of $(u, v) \in E$ or $(v, u) \in E$. This is also not a real restriction on the problems we can discuss.

▷ Parallel arcs can be merged into one, while adding their capacity values.

▷ If anti-parallel arcs (u, v), (v, u) appear, one can add a new node w and replace (u, v) by (u, w) and (w, v), with $c(u, w) = c(w, v) := c(u, v)$. See Figure 8.1 for an illustration.

Arcs incident to a node that point to a node in the in-neighborhood or out-neighborhood are the *in-arcs [Eingehende Bögen]* and *out-arcs [Ausgehende Bögen]* of a node $v \in V$. We denote them by

$$\begin{aligned} \delta^-(v) &:= \{e = (u, v) \in E\} \\ &= \{(u, v) \mid u \in \mathrm{N}^-(v)\} \end{aligned} \qquad \text{and} \qquad \begin{aligned} \delta^+(v) &:= \{e = (v, w) \in E\} \\ &= \{(v, w) \mid w \in \mathrm{N}^+(v)\}\ . \end{aligned}$$

With this definition we can define a notion of a *flow* in a network.

Definition 8.3. An *s-t-flow [s-t-Fluss]* in a network G is a map $f : E \to \mathbb{R}$ such that

▷ $0 \le f(e) \le c(e)$ for all $e \in E$ *capacity constraints [Kapazitätsschranken]*

▷ $\sum_{e \in \delta^-(u)} f(e) = \sum_{e \in \delta^+(u)} f(e)$ for all $u \neq s, t$. *flow conservation [Flusserhaltung]*

The *value [Wert]* of a flow is

$$|f| := \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e),$$

which is the net outflow of the source.

In the next section we will discuss a method to maximize the flow value in a network. Motivated by applications there are many variations and generalizations of this problem which we will not discuss in this course. For example, we could introduce the requirement that the flow on arcs may not fall below a certain level (to keep pipes open or clean), or we could think of flow problems with more than one source or target. Questions of this type are still a topic of active current research.

8.2 Maximum Flows

Definition 8.4. An s-t-flow f is *maximum* if $|f| \geq |f'|$ for all other s-t-flows f' in G.

In this section we want to discuss the question how we can determine the value of a maximal flow in a graph, and how we can construct such a flow. Formally, we aim to solve the following problem.

Flow Problem.

input: a network $G = (V, E)$ with a source node $s \in E$, a sink $t \in E$, and capacities $c : E \to \mathbb{R}_{\geq 0}$.

problem: find a flow $f : E \to \mathbb{R}_{\geq 0}$ of maximum value in G.

The following notion of a *cut* will turn out to be a crucial ingredient for the construction of a maximum flow. It extends the definition of a cut for undirected graphs in Definition 6.2 to directed graphs.

Definition 8.5. Let $S \subset V$ with $s \in S$, $t \notin S$. The arc set

$$\delta^+(S) \;:=\; \{(u,v) \in E \mid u \in S,\ v \notin S\}$$

is an *s-t-cut [s-t-Schnitt]*. The *capacity [Kapazität]* of the cut is

$$c(\delta^+(S)) \;:=\; \sum_{e \in \delta^+(S)} c(e)\,.$$

We will see that there is a close connection between cuts and flows in a network. An example can be seen in Figure 8.2. Observe that these cuts are much less obvious to spot in directed graphs, as there may also be arcs pointing in the opposite direction from $V \setminus S$ into S. Also observe, that, unlike cuts in undirected graphs, this is not symmetric in S and $V \setminus S$.

Let us first show that we can obtain the flow value using any s-t-cut in the graph.

Proposition 8.6. *For any $S \subseteq V$ such that $s \in S$, $t \notin S$ and any s-t-flow f, we have*

$$|f| \;=\; \sum_{e \in \delta^+(S)} f(e) \;-\; \sum_{e \in \delta^-(S)} f(e)\,.$$

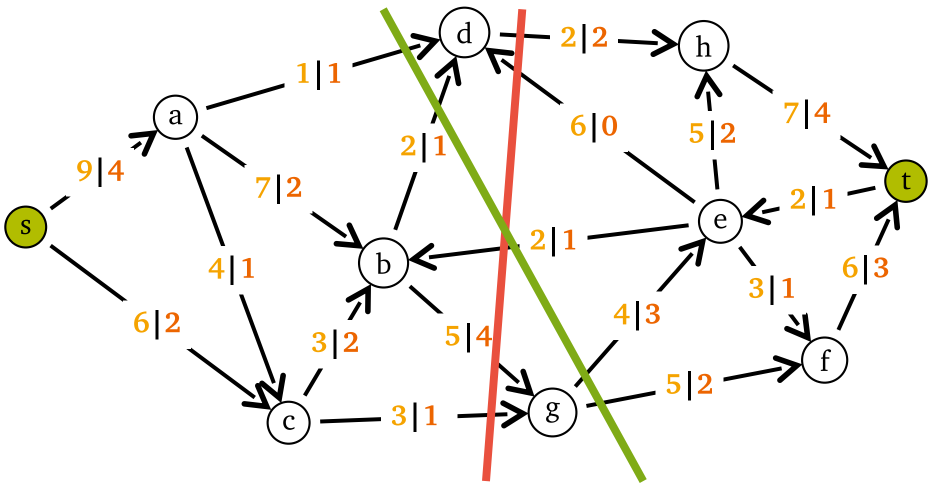


Figure 8.2: A network with a capacity in yellow and a flow in red given as a pair of numbers for each arc. The green line gives a cut of value 12, the red line gives a cut of value 10 in the network.

Moreover,

$$|f| \leq c(\delta^+(S)) .$$

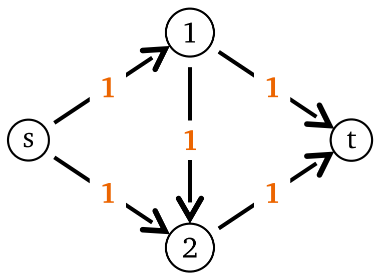
Proof. We have, by definition of $|f|$ and the flow conservation rule:

$$\begin{aligned} |f| &= \sum_{e\in\delta^+(s)} f(e) - \sum_{e\in\delta^-(s)} f(e) \\ &= \sum_{v\in S}\Big(\sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e)\Big) \\ &= \sum_{e\in\delta^+(S)} f(e) - \sum_{e\in\delta^-(S)} f(e). \end{aligned}$$

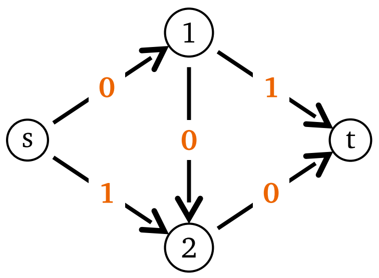
Moreover, since $0 \leq f(e) \leq c(e)$ for all $e \in E$, it follows that $|f| \leq c(\delta^+(S))$. □

Example 8.7. Consider the network in Figure 8.2. The given flow has a value of 6. A net value of 13 can flow out of the source. This is certainly an upper bound for the maximal value of a flow in this network. However, the red cut shows that at most 10 is (theoretically) possible.

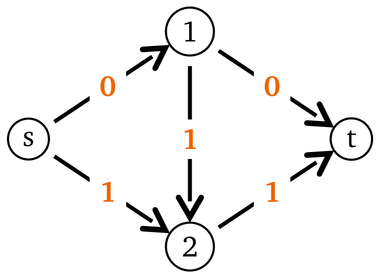
The main insight (shown in the following section) is that equality $|f| = c(\delta^+(S))$ holds for some $S \subseteq V \setminus \{t\}$ with $s \in S$.



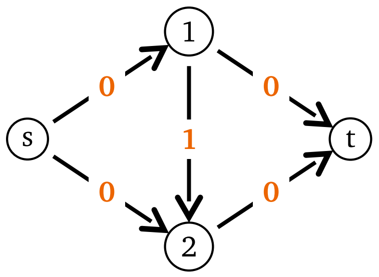
(a) We can route a flow of value 1 on the path s–1–2–t.



(b) The network with reduced capacities. No further s-t-path with a positive reduced capacity on each arc exists.



(c) The reduced capacities of the restarted algorithm with first routing a flow of value 1 on the path s–1–t.



(d) Further routing a value 1 on the path s–2–t yields the shown reduced capacities and a flow with value 2. This flow is optimal, since $c(\delta^+(\{s\})) = 2$.

Figure 8.3: Our first attempt to find a maximal flow may fail if we choose the wrong path. In Figure 8.3(a) one we choose a path that allows an increment by 1, and Figure 8.3(b) shows that no further increment is possible with our method. However, if we start with another path, we may obtain a total flow of 2, as shown in Figure 8.3(c) and Figure 8.3(d).

8.3 The Ford-Fulkerson-Method

Our main goal in this section will be to determine an s-t-flow with maximal value in G. The general (rather simple) idea will be the following:

▷ Start with the 0-flow, *i.e.*, $f(e) = 0$ for all $e \in E$.

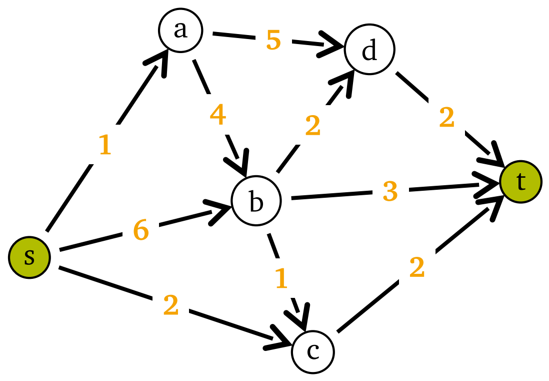
▷ Find an s-t-path and push as much flow onto the path as possible.

▷ Determine all left-over capacities.

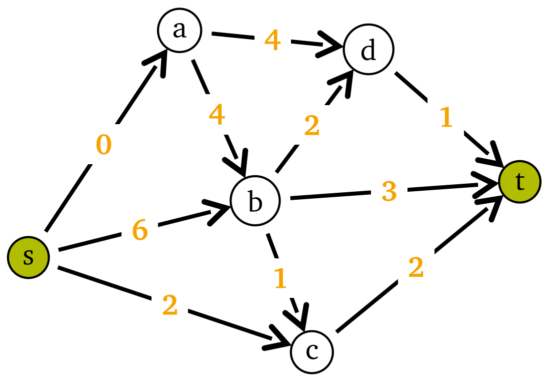
▷ Repeat this procedure with the new capacities.

The outcome is clearly an s-t-flow. See Figure 8.4 on the next page for an example how this method can work and produces a maximal flow.

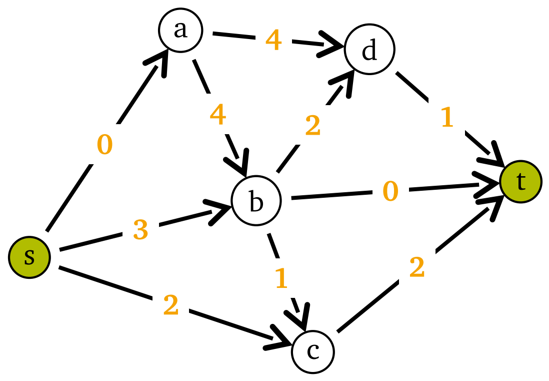
However, it is also pretty simple to show an example where this fails. In the example of Figure 8.3 you see in Figure 8.3(a) and Figure 8.3(b) that the first attempt to choose a path and reduce the capacities does end with a flow of value 1, while another choice of paths as in Figure 8.3(c) and Figure 8.3(d) shows that a total flow of 2 is possible. In order to escape from such situations, we need the following concept.



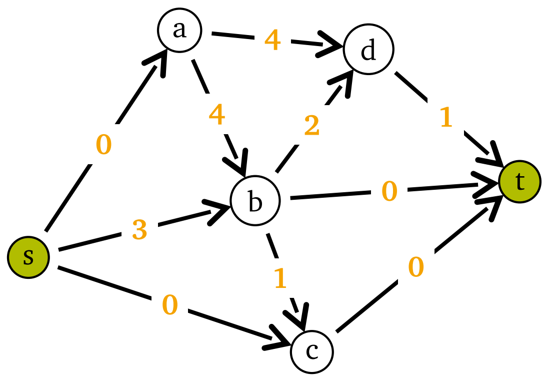
(a) We can route a flow with value 1 along the path $s-a-d-t$.



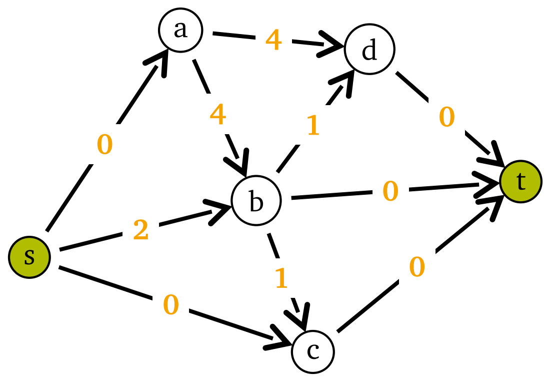
(b) The graph with reduced capacities. We can still route a flow with value 3 along the path $s-b-t$.



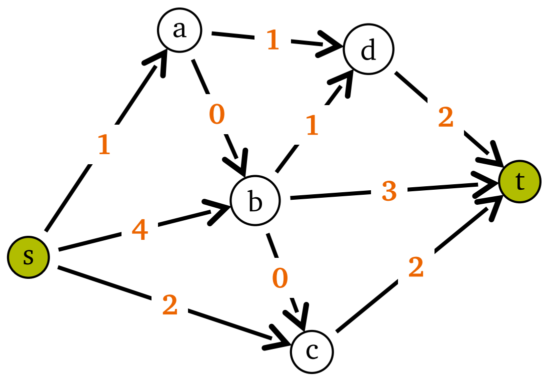
(c) Reduced capacities along $s-b-t$. We can still route a flow with value 2 along the path $s-c-t$.



(d) Reduced capacities along $s-c-t$. We can still route a flow with value 1 along the path $s-b-d-t$.

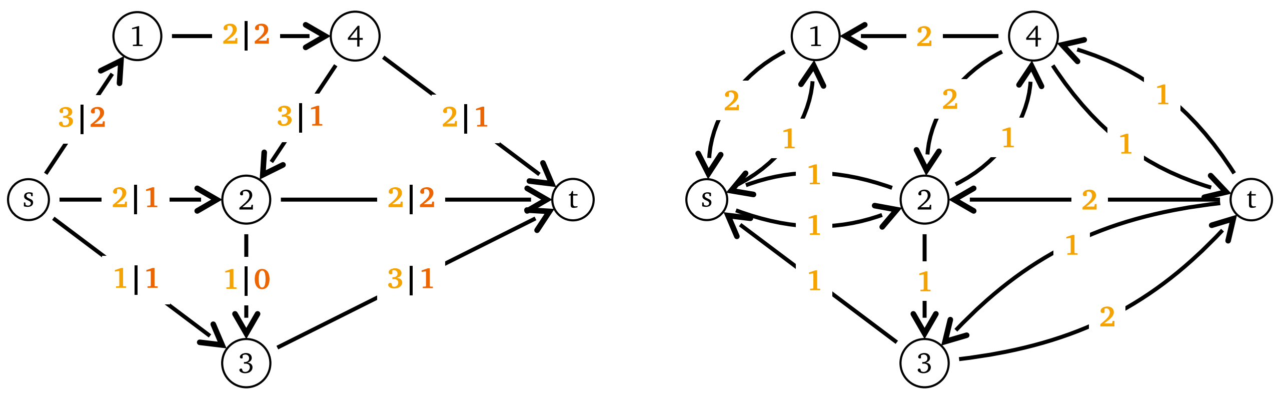


(e) Reduced capacities along $s-b-d-t$. No further s-t-path with positive capacitities exists.



(f) The final flow. It must be maximal, as no further flow can go into t.

Figure 8.4: Finding a maximal flow. We start with a flow of 0 along all arcs, choose a path where we can route flow, remember the value and reduce the capacities. Capacities are shown in yellow, the final flow in red.



(a) A network with a non-maximal flow

(b) The corresponding residual graph. The path s–2–4–t is s-t-augmenting.

Figure 8.5: A network with a flow and its residual graph.

Definition 8.8. Let $(G = (V, E), s, t, c)$ be a network with capacity function $c : E \to \mathbb{R}_{\geq 0}$ and let f be an s-t-flow in G.

For each arc $e = (u, v)$, we define a reverse (backward) arc $\overleftarrow{e} = (v, u)$. Let

$$\overleftarrow{E} := \{\overleftarrow{e} \mid e \in E\} \qquad \text{and} \qquad \overleftrightarrow{G} := (V, E \cup \overleftarrow{E}).$$

We define the *residual capacity [Residualkapazität]* of $e \in E$ with respect to f as

$$c_f(e) := c(e) - f(e) \qquad \text{and} \qquad c_f(\overleftarrow{e}) := f(e).$$

The *residual network [Residualnetzwerk]* is the network (G_f, s, t, c_f) that consists of the *residual graph [Residualgraph]* $G_f = (V, \{e \in \overleftrightarrow{G} \mid c_f(e) > 0\})$ (the directed graph with all arcs of positive residual capacity), capacity function c_f, source s and sink t.

Example. Figure 8.5 shows a non-maximal flow in a network and its residual network.

Note that some arcs appear with different orientation in the residual graph. We can think of this as *having the option to reduce the flow* along the original arc.

Definition 8.9. An s-t-path P in G_f is an *s-t-augmenting [s-t-vergrößernd]* path. The capacity of the path is $\gamma(P) = \min_{e \in P} c_f(e)$.

Augmenting [augmentieren] an s-t-flow f along path P means, for each $a \in P$, to increase $f(a)$ by $\gamma(P)$ if $a \in E$ and reducing $f(e)$ by $\gamma(P)$ if $a = \overleftarrow{e} \in \overleftarrow{E}$.

By construction the flow resulting from an augmentation along P is an s-t-flow. This gives the following optimality conditions:

Method 8.1: Ford-Fulkerson

```
Input  : network            $G = (V, E)$,
         capacity function  $c : E \to \mathbb{R}_{\geq 0}$,
         capacity function  $c : E \to \mathbb{R}_{\geq 0}$,
         source and target  $s, t \in V$
Output : maximal flow       $f : E \to \mathbb{R}_{\geq 0}$,
         flow value         $f_{\max}$
1  $f_{\max} \leftarrow 0$
2  foreach $e \in E$ do
3      $f(e) \leftarrow 0$
4  while there is an $s$-$t$-path $P$ in $G_f = (V, E_f)$ do
5      $\gamma(P) \leftarrow \min\{c_f(e) \mid e \in P\}$
6      $f_{\max} \leftarrow f_{\max} + \gamma(P)$
7      foreach $e = (u, v) \in P$ do
8          if $e \in E$ then
9              $f(e) \leftarrow f(e) + \gamma(P)$
10         else
11             $f(\overleftarrow{e}) \leftarrow f(\overleftarrow{e}) - \gamma(P)$
```

Theorem 8.10 (*MaxFlow-MinCut*)**.** *Let $G = (V, E)$ be a network with source s, target t, a capacity function c, and an s-t-flow f. Then the following are equivalent:*

(i) *f is maximal,*

(ii) *there is no s-t-augmenting path,*

(iii) $|f| = \min\{c(\delta^+(S)) \mid S \subset V \setminus \{t\},\ s \in S\}$.

Proof. It is clear that (i) implies (ii).

Assume that (ii) holds. Let $S \subseteq V$ be the nodes u that can be reached by a path from s to u in G_f. By assumption $t \notin S$. Consider an arc $e = (u, v) \in \delta^+(S)$ in G. Since, by definition $v \notin S$, we have $c_f(e) = c(e) - f(e) = 0$, *i.e.*, $f(e) = c(e)$. Now consider an arc $e = (v, u) \in \delta^-(S)$ in G. Then $f(e) = 0$, since otherwise $f(e) > 0$ would imply that $\overleftarrow{e} = (u, v) \in G_f$, *i.e.*, $c_f(\overleftarrow{e}) > 0$. This would imply that v can be reached by a path to u and then via arc $\overleftarrow{e}$, *i.e.*, $v \in S$.

Hence, by Proposition 8.6:

$$\begin{aligned} |f| &= \sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e) \\ &= \sum_{e \in \delta^+(S)} c(e) - 0 \\ &= c(\delta^+(S)). \end{aligned}$$

Therefore, (ii) implies (iii).

Finally, again by Proposition 8.6 it is clear that (iii) implies (i). □

These conditions suggest Method 8.1, the *Ford-Fulkerson Method*, that successively searches for s-t-augmenting paths in the graph and augments along such a path, as long as there is such a path left. Using Theorem 8.10, it is clear that it produces an optimal flow, if it terminates. However, the suprising observation is that, in general, *the method need not terminate*, and it *may even converge to some value smaller than the optimum*, if the capacities contain irrational numbers. The problem here is that we cannot directly control the amount by which we update the flow, and it may happen that the possible increment in each step drops too fast.

We give one example[1].

Example 8.11. Let $q := \frac{\sqrt{5}-1}{2}$. This is the inverse of the *golden ratio*, and satisfies several nice equalities. We will use the following identities:

$$\begin{aligned} 1 &= q + q^2 \\ q^k &= q^{k+1} + q^{k+2} && \text{for } k \geq 0 \\ \sum_{n=0}^{\infty} q^n &= \frac{\sqrt{5}+3}{2} < 4\,. \end{aligned}$$

The second equation shows that $a_k := q^k$ solves the recursion $a_{n+2} := a_n - a_{n+1}$ with initial values $a_0 = 1$ and $a_1 = q$.

Now consider the graph in Figure 8.6(a) for some $M \geq 4$. It is straightforward to check that its maximum flow is $2M + 1$ (find the corresponding cut and use Theorem 8.10). However, if we do the wrong choices for our augmenting paths we will not reach this value.

In a first augmenting step we can choose the path $s - b - c - t$, along which we can increase the flow by a value of 1, see Figure 8.6(b). The residual graph is shown in Figure 8.6(c). Now we do the following for augmentations:

(i) Along the path $s-d-c-b-a-t$ we can augment by a value of q. See Figure 8.6(d) for the new flow and Figure 8.6(e) for the resulting residual graph.

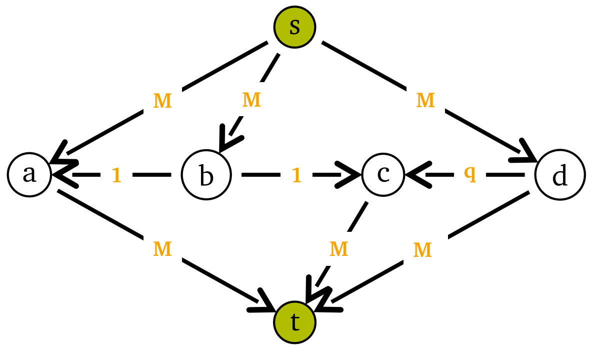
(ii) Now we can augment by a value of q along the path $s - b - c - d - t$ and obtain the flow shown in Figure 8.6(f). We omit the further residual graphs.

(iii) In this network we can now again augment along the path $s - d - c - b - a - t$, but only by a value of q^2. The new flow is in Figure 8.6(g)

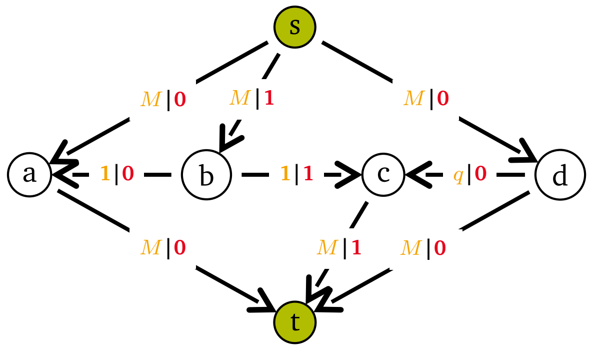
(iv) Finally ,we use the path $s - a - b - c - t$, along which we can also augment by a value of q^2. We end with the flow in Figure 8.6(h)

So far we have added a flow value of $1 + 2(q + q^2)$. Observe that by our choice M is large enough that it will not be the limiting capacity along any path. Further, if we compare the flows in Figure 8.6(b) and Figure 8.6(h) we observe that we have a similar situation, but the remaining free capacities on the arcs (d, c) and (b, a) are now $q^3 = a_3$ and $q^2 = a_2$ instead of $q = a_1$ and $1 = a_0$. We can thus continue with the same four augmentation steps, but this time add flow values of q^3, q^3, q^4, and q^4 to end with the same picture, but remaining free capacities $q^4 = a_5$ and $q^5 = a6$ on the arcs (b, a) and

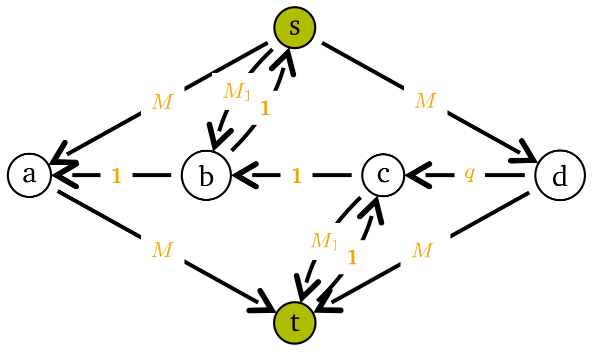
[1] Toshihiko Takahashi. “The Simplest and Smallest Network on Which theFord-Fulkerson Maximum Flow Procedure MayFail to Terminate”. In: *J. Inf. Proc.* 24.2 (2016), pp. 290–294. DOI: 10.2197/ipsjjip.24.390.



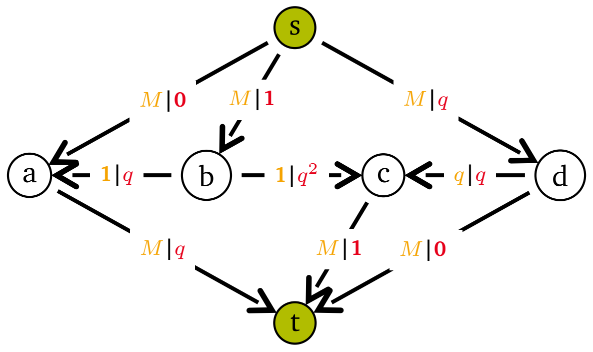
(a) A graph with capacities. Route flow 1 along $s - b - c - t$.



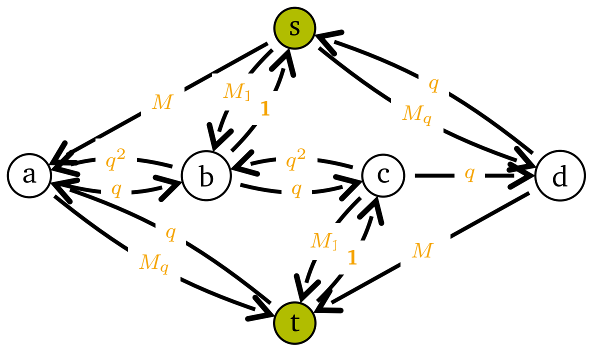
(b) The resulting flow.



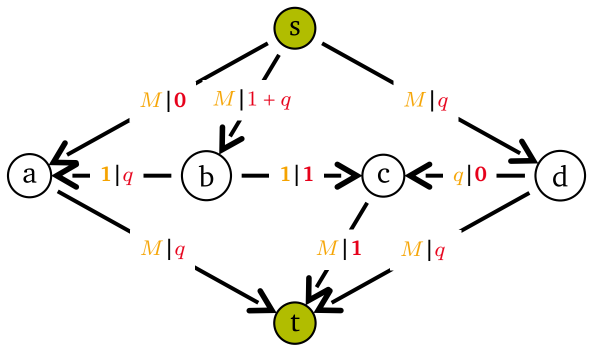
(c) The residual graph, with $M_1 := M - 1$. Route flow q along $s - d - c - b - a - t$.



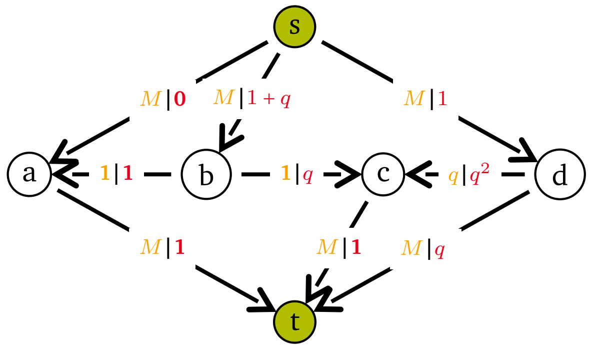
(d) The resulting flow.



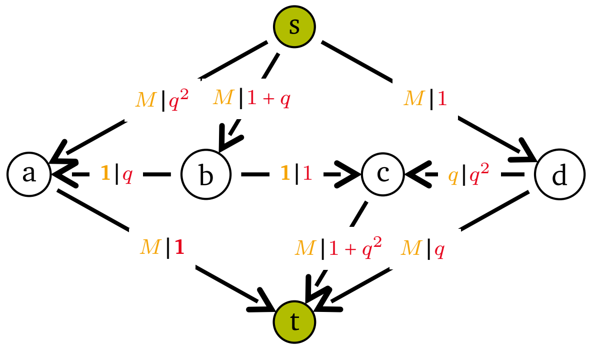
(e) The residual graph, with $M_1 := M - 1$ and $M_q := M - q$. Route flow q along $s - b - c - d - t$.



(f) Route flow q^2 along $s - d - c - b - a - t$.



(g) Route flow q along $s - a - b - c - t$.



(h) The resulting flow.

Figure 8.6: The first round of flow updates for Example 8.11

(d, c). The total flow is now $1 + 2(q + q^2 + q^3 + q^4)$.

In this way, we can repeat the four steps to add a value of $2(q^{2k-1} + q^{2k})$ for $k \geq 3$, while reducing the free capacities on (b, a) and (d, c) to q^{2k} and q^{2k+1}. So the total flow value for $k \geq 3$ is

$$|f| = 1 + 2\sum_{j\geq 1}^{2k} q^j .$$

Hence, by our third identity above, the flow converges to 3, which is less than the optimal flow of $2M + 1 \geq 9$. Observe that the arcs incident to s and t never limit the flow increment along one of our paths, as their capacity is larger than the flow we can reach with our augmenting paths.

8.4 Flow Algorithms

We have seen that the Ford-Fulkerson Method may not successfully produce a maximum flow in our network. In this section we will discuss two options to rectify this situation by making the method more specific. In Example 8.11 one of the capacities was irrational. We will see that this was essential for the example, as the method always terminates with rational capacities. The example also showed that, although our choice of augmenting paths did not lead to the maximum flow, there was also a choice of paths available that leads to the maximum flow. We will see below that a specific choice of augmenting paths will also guarantee termination with the maximum flow.

Let us first discuss the case of rational weights. Up to a total scaling of the capacities this is the same as integral weights, so we will look at weights in $\mathbb{Z}_{\geq 0}$ instead. This case has further nice properties that we introduce below.

Definition 8.12. An s-t-flow f on a network (G, s, t, c) is *integral [ganzzahlig]* if $f(e) \in \mathbb{Z}_{\geq 0}$ for all $e \in E$.

With this definition it is actually pretty simple to show that the Ford-Fulkerson Method must terminate on networks with integral capacities.

Corollary 8.13 (*Integral Flow Theorem*)**.** *Let $(G = (V, E), s, t, c)$ be a network with capacity function $c : E \to \mathbb{Z}_{\geq 0}$ and m arcs. Then G has an integral maximal flow $f^\star$ that can be found in $\mathcal{O}((m + n)\,|f^\star|)$ time.*

Proof. The flow $f \equiv 0$ is integral and feasible. Augmenting along an s-t-path changes f by integral values only and increases $|f|$ by at least 1. The value $|f^\star|$ is an upper bound, so we need at most $|f^\star|$ iterations.

Initialization (e.g., setting up $\overleftrightarrow{G}$) can be done in $\mathcal{O}(m)$ time. We can use BFS or DFS to find an s-t-augmenting path. In total, this gives an $\mathcal{O}((m + n)\,|f^\star|)$ running time. □

Observe again that the argument of the proof extends to rational capacities: we can scale with the least common multiple of all denominators to obtain integral capacities. So for rational capacities, the Ford-Fulkerson method terminates in finite time and returns a maximal flow.

Note that the running time also depends on the *output* of the algorithm. Moreover, there exist examples of networks where Method 8.1 actually uses $|f^\star|$ many iterations.

Example 8.14. Consider the network in Figure 8.7. The maximum flow value in this example is $2a$. Our choice of s-t-augmenting paths always uses the arc between 1 and 2, which has a capacity of 1. Hence, we need $2a$ steps to reach the maximum.

However, the example also suggests that we did a particularly poor choice of augmenting paths. We can achieve the maximum already in two steps by choosing the shorter augmenting paths s-1-t and s-2-t.

We show with the following theorem that the observation in the example, that we should have chosen the shorter augmenting paths, indeed leads to an output-independent algorithm, if we always use a *shortest* s-t-augmenting path, where *shortest* refers to the number of edges on the path, irrespective of their capacity. This specialization of the Ford-Fulkerson method is due to Edmonds and Karp[2].

Theorem 8.15 (Edmonds-Karp Algorithm)**.** *If we always choose a shortest s-t-augmenting path in Method 8.1 then it terminates with a maximal flow in time $\mathcal{O}(nm^2)$, even for irrational capacities.*

Proof. We use the Ford-Fulkerson method with the additional rule that in each iteration we choose the shortest (number of arcs) s-t-augmenting path and augment by the maximal possible amount of flow along this path.

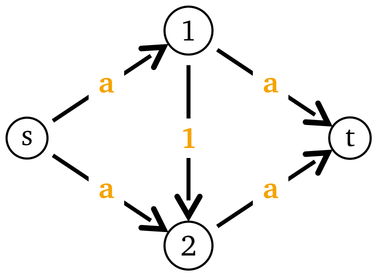
We compute s-t-paths in G_f with the least number of edges with Breadth-First Search. The algorithm assigns a *level* $\mathsf{level}(v)$ to each node of G_f, which, by Theorem 5.10, equals the path distance from s to v. This implies that $\mathsf{level}(v)$ is actually independent of the actual spanning tree obtained by BFS and if $s = v_0v_1 \ldots v_k = v$ is a shortest path from s to a node v at level k, then $\mathsf{level}(v_j) = j$. Further, by Proposition 5.14 any edge in the graph connects nodes whose levels differ by at most 1.

We follow changes in level when we augment along paths in G_f. Let p be a shortest s-t-augmenting path. If we augment by the maximum possible value, then at least one arc along p is saturated. Let $e = uv$ be such an arc (this may be an arc of G or a backward arc). In the update of G_f that is done when augmenting the flow the edge e is removed, and, unless already present, the arc vu is added. As p is a shortest path, we know that

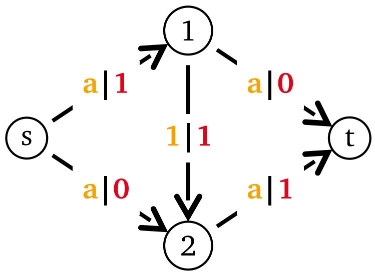
$$k \;:=\; \mathsf{level}(v) \;=\; \mathsf{level}(u) \;+\; 1\,,$$

i.e., it pointed from level $k-1$ to k. If we reverse it, then the level of a node w that used e in a shortest path from s to w can at most *increase* (it may stay the same if there is

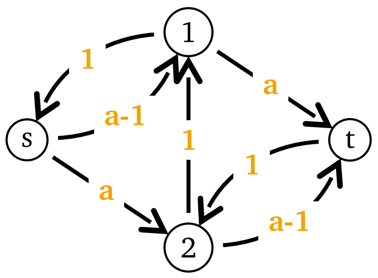
[2] Jack Edmonds and Richard M. Karp. “Theoretical improvements in algorithmic efficiency for network flow problems.” English. In: *J. Assoc. Comput. Mach.* 19 (1972), pp. 248–264. ISSN: 0004-5411. DOI: 10.1145/321694.321699.



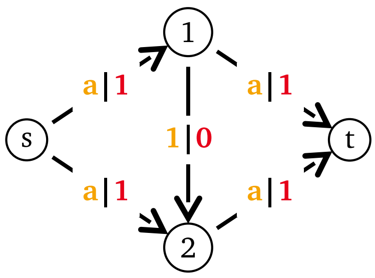
(a) A simple network. If we start with flow $f \equiv 0$, then this is also the residual graph and the path s–1–2–t is s-t-augmenting. It allows to increase the flow by 1.



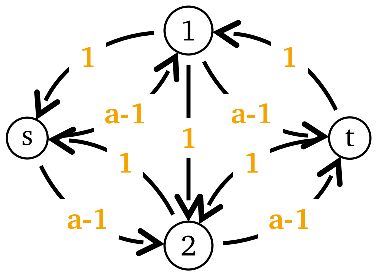
(b) The network with increased flow.



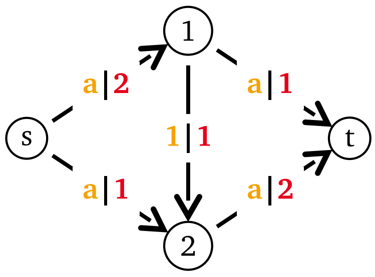
(c) The residual graph. We can increase along the path s-2-1-t by a value of 1.



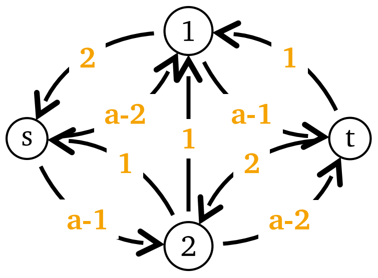
(d) The network with increased flow.



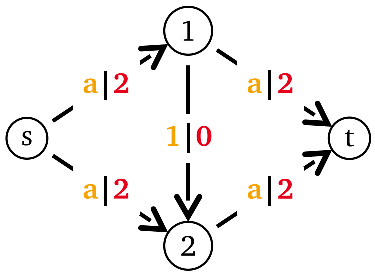
(e) The residual graph. We can increase along the path s-1-2-t by a value of 1.



(f) The network with increased flow.



(g) The residual graph. We can increase along the path s-2-1-t by a value of 1.



(h) The network with increased flow.

Figure 8.7: A network where we can find a sequence of s-t-augmenting paths that increase the value of the flow f by 1 in each step. Hence, we need $|f^*|$-many steps to reach the maximal flow. We do the first few steps. The continuation should be clear from those.
This shows that without further assumptions the Ford-Fulkerson method cannot produce a maximal flow in time independent of the size of the output.

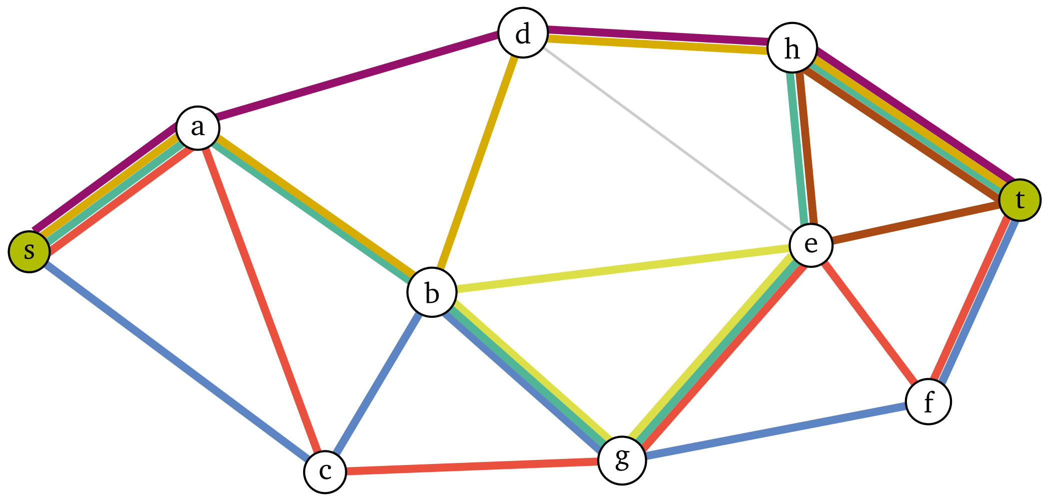


Figure 8.8: Path decomposition of the flow in Figure 8.2. All paths and cycles have value 1, except the blue path, which has a value of 2. We have two cycles, one with nodes b, g, e and one with nodes e, h, t. We have omitted the arc directions in the figure.

also a shortest path that does not use e). All other levels remain unchanged. The crucial observation here is that, if we augment along a shortest path, then the level of a node can at most increase. But it will never decrease.

As the level k of v never decreases, we can only use the arc vu in an s-t-augmenting path if the level of u increases from $k-1$ to $k+1$. As the level is at most $n-1$ this implies that the arc e or its backward arc can be the saturated arc of an s-t-augmenting path for at most $\frac{n}{2}$ times.

Now each s-t-augmenting path has at least one saturated arc, and we have at most $2m$ arcs in G_f, so we can do at most mn augmentations. We can find an s-t-augmenting path with Breadth-First Search. This can be done in time $\mathcal{O}(n+m) = \mathcal{O}(m)$ as we assume that there are no isolated nodes. Updating G_f can also be done in $\mathcal{O}(m)$, so that we obtain a total running time of $\mathcal{O}(m^2 n)$. $\square$

A modified version of this algorithm was earlier found by Y.A. Dinic[3] that runs in $\mathcal{O}(mn^2)$, who also described the Edmonds-Karp algorithm already in his paper.

8.5 Path Decomposition

In many applications one would like to route the flow along full paths from s to t, *e.g.*, if loading trucks one would like to avoid exchanging loads at intermediate nodes. Just using paths this cannot be done. But if we allow cycles, this is possible. We can decompose an s-t-flow in a network G into a sum of weighted s-t-paths and cycles. See Figure 8.8 for a path decomposition of the flow in Figure 8.2. We prove that this is always possible with the next theorem.

[3] Y. A. Dinic. "An algorithm for the solution of the problem of maximal flow in a network with power estimation". In: *Dokl. Akad. Nauk SSSR* 194.4 (1970), pp. 754–757.

Theorem 8.16. *Let G be a network and f an s-t-flow on G. Then there exist s-t-paths $P_1, \dots, P_k$ with weights $w_1, \dots, w_k$ and cycles $C_1, \dots, C_\ell$ with weights $\overline{w}_1, \dots, \overline{w}_\ell$ such that for all $e \in E$*

$$f(e) = \sum_{i: e \in P_i} w_i + \sum_{j: e \in C_j} \overline{w}_j,$$

and $k + \ell \leq |E|$.

If f is integral, all weights can be chosen to be integral as well.

Proof. Let (v_0, w_0) be an arc with nonzero flow. Unless $w_0 = t$, by flow conservation, there exists an arc (w_0, w_1) with positive flow. Repeating this argument, we obtain a path $(v_0, w_0, w_1, \dots, w_k)$ such that either $w_k = t$ or $w_k \in \{v_0, w_0, \dots, w_{k-1}\}$. Applying the same argument in the reverse direction, we obtain an arc (v_1, v_0) with positive flow. Repeating this, yields $(v_\ell, \dots, v_0, w_0, \dots, w_k)$ such that $v_\ell = s$ or $v_\ell \in \{v_{\ell-1}, \dots, v_0, w_0, \dots, w_k\}$. Thus, we either obtain an s-t-path P, and reduce the flow along P by $\gamma(P)$. In the other case, we found a cycle C and we reduce the flow by the minimal flow along C. In both cases, we again obtain an s-t-flow.

Since in each iteration, the flow is reduced to 0 on at least one arc, we have at most m many paths and cycles. Moreover, the weights are integral, if the flow is integral. □

8.6 Menger's Theorems

A famous and important consequence of the path decomposition are the two Theorems of Menger, which state that the number of disjoint paths equals the size of a separating set, where a separating set can ether be defined with nodes, or with edges. We state it for undirected graphs, but the same is true for directed graphs. Let us first formalize the notion of a *separating set* for edges.

Definition 8.17. Let $G = (V, E)$ be an undirected graph and $s, t \in V$.

(i) A subset $F \subseteq E$ is *s-t-separating [s-t-separierend]* if any s-t-path uses at least one arc of F.

(ii) A collection $P_1, \dots, P_k$ of s-t-paths in G is *edge-disjoint [kantendisjunkt]* if no pair $P_i, P_j, i \neq j$ have an arc in common.

With this we can formulate the first of the two Theorems of Menger.

Theorem 8.18 (Menger, edge version). *The maximal number of edge disjoint-paths in an* undirected *graph G equals the minimal size of an s-t-separating set.*

For the hard direction in the proof we construct a directed graph D and reduce to a flow problem. A path decomposition of the flow and the bound by any s-t-cut will then

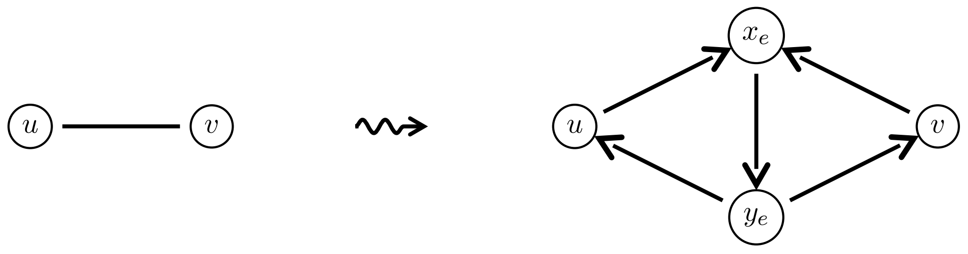


Figure 8.9: Construction in the proof of Theorem 8.18.

prove the result. The directed graph needed is $D := (V, A)$, where the arcs are created by replacing each edge $e = \{u, v\} \in E$ by arcs (u, x_e), (v, x_e), (x_e, y_e), (y_e, u), (y_e, v), where x_e and y_e are new nodes; see Figure 8.9. Moreover, we define the capacities $c(a) = 1$ for all $a \in A$.

Proof. We show both inequalities:

It is clear that the number of edge-disjoint paths is smaller or equal than the size of any s-t-separating set since each s-t-path uses at least one edge from every s-t-separating set.

Conversely, we use the directed graph $D := (V, A)$ introduced above. In this graph any maximal flow $f^\star$ in D is integral, and thus any s-t-cut as well. By Theorem 8.16, there exists an integral path decomposition that uses each arc at most once with at least $|f^\star|$ paths. On the other hand, any s-t-cut in D defines an s-t-separating set in G (of possibly smaller size, because of the construction in Figure 8.9).

Hence we obtain:

$$\begin{aligned}\text{maximal number of edge-disjoint } s\text{-}t\text{-paths} &\geq |f^\star| \\ &= \text{min cut} \\ &\geq \text{minimal } s\text{-}t\text{-separating set.}\end{aligned}$$

This completes the proof. $\square$

We switch to the node version and first define a corresponding notion of a *separating set* and then finish with the second Theorem of Menger.

Definition 8.19. Let G be an undirected graph and $s, t \in V(G)$.

- ▷ Two s-t-paths are *(internally) node-disjoint [kreuzungsfrei]* if they only share the nodes s and t.
- ▷ A set $W \subseteq V - \{s, t\}$ is called s-t-separating, if any s-t-path contains a node in W.

Theorem 8.20 (Menger, node version)**.** *Let G be an undirected graph, $s, t \in V(G)$, $s \neq t$, and $\{s, t\} \notin E(G)$. Then the maximal number of node-disjoint s-t-paths equals the minimal size of an s-t-separating node set.*

The proof is along the lines of the proof of the edge version of Menger's Theorem, however, after constructing the directed graph D we also have to split each node u from the original node set into two nodes u^- and u^+ together with an arc a_u from u^- to u^+ and replacing all arcs

- ▷ with their head in u by arcs with their head in u^-,
- ▷ with their tail in u by arcs with their tail in u^-.

Now paths using u in the original graph correspond to paths using a_u in the auxiliary graph and we can proceed as above.

Proof. Exercise. □

9 Matchings

Contents

In this chapter we turn to assignment problems that can be modelled with graphs. In these problems, we often have nodes of (at least) two different types together with some relation between these nodes. The relation may possibly weighted.

We have seen some examples in the very beginning of the course.

▷ Each semester the university has to schedule exams for various courses using some lecture halls. Clearly, in this problem we need enough seats for every participant, at any time we can schedule only one exam in a lecture hall. So for a given time slot we need to find an assignment between lecture halls and exams scheduled at this time.

▷ At a bicycle repair shop people turn in their bikes for fixing or maintenance, and you need to assign the different jobs to your colleagues, where jobs may have different priorities, as some customers have a service contract, or some jobs are better assigned to a specific worker, as she is specially trained.

▷ For your local sports league you want to create a training schedule and find a partner for each team, where you want to follow their time or travel constraints as much as possible.

The first two ask for assignments in a graph with two node classes with a different meaning (*e.g.*, bikes and colleagues), and all edges (the possible assignments) connect nodes in each class. Thus, the assignment lives in a bipartite graph, which is an important graph class for matching problems. The last one has only one type of nodes (the teams), so this is an assignment or matching problem in a general graph. Matchings in bipartite graphs are much easier. In this course we will concentrate on this subclass for algorithms to find a (weighted) matching. Algorithms for the general case are discussed in a subsequent class on *Discrete Optimization*.

In this chapter we again consider undirected graphs $G = (V, E)$. In the next Section 9.1 we give some definitions, before we consider matchings in bipartite graphs in Section 9.2 and a particular problem on a weighted bipartite graph in Section 9.3.

9.1 Basic Definitions and Properties

Definition 9.1. Let $G = (V, E)$ be a graph. A subset $M \subseteq E$ is a *matching [Paarung]* in G if $m \cap m' = \emptyset$ for all $m, m' \in M$, $m \neq m'$, *i.e.,* edges in M are pairwise non-incident.

A matching M is *maximal* if M is not a proper subset of a larger matching. The *matching number [Paarungszahl]* is

$$\nu(G) := \max\{|M| \mid M \text{ matching in } G\}\,.$$

A matching is *maximum* if $|M| = \nu(G)$. It is *perfect [perfekt]* if $2|M| = |V| = 2\nu(G)$.

Example 9.2. Consider the three matchings in the graph G shown in Figure 9.1. The ones in the center and on the right (*i.e.,* the red and the blue ones) are both maximal, since they are not contained in a larger matching. The one the right hand side (*i.e.,* the blue one) is also maximum. The yellow one on the left is neither maximal nor maximum since it is contained in the blue one.

This example shows that a maximal matching need not be maximum. Clearly, a perfect matching can only exist if the number $n = |V|$ of nodes is even. We will see further restrictions below.

In the following we discuss the problem to find a maximum matching in a graph.

Maximum Matching Problem.

input:	a graph $G = (V, E)$.
problem:	find a maximum matching M in G.

A first and simple attempt to construct matchings is the *greedy* approach to pick edges in the graph one by one as long as the choice still constitutes a matching. We stop if we cannot find any further edge we can add. This is formalized in Algorithm 9.1. We can see from the second (red) matching in Example 9.2 that this need not end with a maximum matching. Yet, we can show that the size of the resulting matching is always at least half of the optimum.

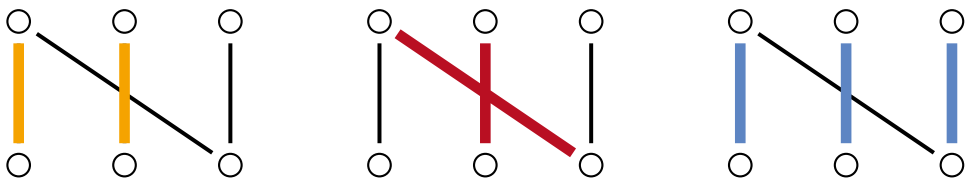


Figure 9.1: Three matchings in a graph. The one in the middle is maximal, the one on the right is a maximum matching

Algorithm 9.1: Maximal Matching

```
Input  : undirected graph $G = (V, E)$
Output : some maximal matching $M$
1 $M \leftarrow \emptyset$
2 foreach $e \in E$ do
3     if $M \cup \{e\}$ is a matching then
4         $M \leftarrow M \cup \{e\}$
```

Proposition 9.3. *Let M be obtained by Algorithm 9.1. Then $|M| \geq \frac{1}{2}\nu(G)$.*

Proof. Let M' be a matching realising $\nu(G)$ and $S := \{(m', m) \in M' \times M \mid m' \cap m \neq \emptyset\}$. Then $|S| \leq 2|M|$. Hence if $|M| < \frac{1}{2}|M'|$ then $|S| \leq |M'|$, so there is $e \in M'$ not incident to an edge in M. The algorithm would have added this edge to M. □

If we look at a path in the graph, then at most every second edge on the path can be an edge in the matching. If we find a path where indeed every second edge is in the matching, we can try to exchange along the path. We put an edge not in the matching into the matching, and take out the matching edges. We may create conflicts at the first and last node of the path.

Yet, this idea is powerful enough, that it will lead us to our first results on matchings in graphs. We formalize this in the next definition, together with a condition that ensures we do not get a conflict at the first and last node.

Definition 9.4. Let $G = (V, E)$ be a graph and M a matching in G. A path $P = v_0, v_1, \ldots, v_k$ is *M-alternating [M-alternierend]* if edges in M and not in M alternate along P.

The path is *M-augmenting [M-vergrößernd]* if it is M-alternating and v_0 and v_k are not contained in any edge of M.

See Figure 9.2 for an example. If we find an M-augmenting path in a graph then we can flip matching and non-matching edges in the path to obtain a larger matching.

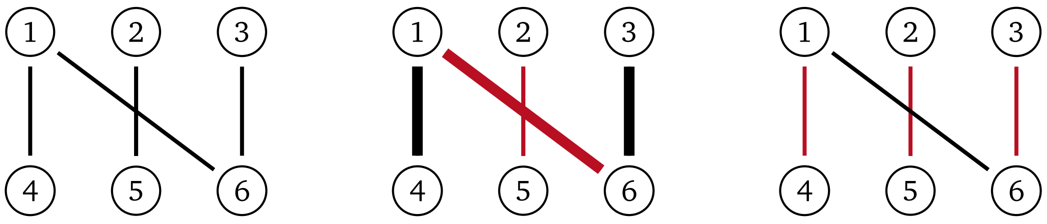


Figure 9.2: Let G be the above graph. Then the set $M = \{\{1, 6\}, \{2, 5\}\}$ (red) is a matching in G. The path $4, 1, 6, 3$ (drawn with thick edges in the center) is M-alternating. (Moreover, it is also M-augmenting and maximal.) If we flip the edges along this path we obtain the larger matching $M = \{\{1, 4\}, \{2, 5\}, \{3, 6\}\}$ drawn on the right.

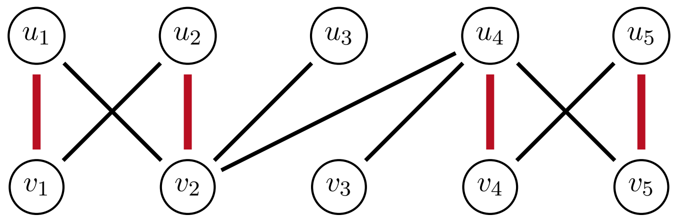


Figure 9.3: A bipartite graph with a maximum matching with red thick edges. The neighborhood of u_1, u_2, and u_3 contains only two nodes, so not all three nodes can be matched.

With the following theorem we prove that M-augmenting paths suffice to characterize a maximum matching.

Theorem 9.5 (*Theorem of Berge*). *Let G be a graph and M a matching. Then M is maximum if and only if there is no M-augmenting path.*

Proof. If M is maximum then it is clear that there is no M-augmenting path.

Conversely, assume that M is not maximum. We want to show that there exists an M-augmenting path. Let M' be a maximum matching.

Consider the subgraph $G' := M \cup M'$. All nodes in G' have degree at most 2. Hence G' is a disjoint union of paths, cycles, and isolated nodes. On cycles and paths in G', edges of M and M' must alternate (except for paths of length 1). Hence cycles have even size and contain the same number of edges from M and M'. As $|M| < |M'|$ there must be a path with more edges from M' than M. This is an M-augmenting path in G. □

Although Theorem 9.5 gives an easy characterization of maximum matchings, the corresponding famous algorithm by Jack Edmonds[1] goes beyond the scope of this lecture series. We therefore deal with bipartite graphs.

9.2 Matchings in Bipartite Graphs

We concentrate on unweighted bipartite graphs and first show that sizes of neighborhoods limit the size of a matching. This is known as the *Marriage Theorem*.

Example. Consider the graph in Figure 9.3. Certainly $\nu \leq 5$ and the red edges give a matching of size 4, so $\nu \geq 4$. But $\mathrm{N}(\{u_1, u_2, u_3\}) = \{v_1, v_2\}$, so we can match at most two of the nodes in $\{u_1, u_2, u_3\}$. Hence $\nu \leq 4$ and thus $\nu = 4$.

With the following theorem we show that for bipartite graphs the size of the neighborhoods of subsets of the nodes in one of the color classes is in fact the only obstruction to a perfect matching.

[1] Jack Edmonds. "Paths, trees, and flowers". In: *Canadian J. Math.* 17 (1965), pp. 449–467. DOI: 10.4153/CJM-1965-045-4.

Theorem 9.6 (*Hall's Theorem / Marriage Theorem [Heiratssatz]*). *Let $G = (A \cup B, E)$ be a bipartite graph. There is a matching M of size $|M| = |A|$ if and only if $|\mathrm{N}(S)| \geq |S|$ for all $S \subseteq A$.*

Proof. It is clear that $|M| = |A|$ implies $|\mathrm{N}(S)| \geq |S|$ for all $S \subseteq A$.

Conversely, let $|\mathrm{N}(S)| \geq |S|$ hold for all $S \subseteq A$. Assume that M is a maximum matching with $|M| < |A|$. Then there is an unmatched node $a \in A$. Let $U \subseteq A \cup B$ be the set of nodes that can be reached from a via an M-alternating path. Then $a \in U$. Let $X := A \cap U$ and $Y := B \cap U$.

By construction, following a path starting in a we use edges not in M to get from A to B and matching edges from B to A. By Theorem 9.5 there is no M-alternating path of maximal length starting in a that ends in B. Thus, each node in $X \setminus \{a\}$ is matched by an edge in M between Y and X. Thus, $|X| - 1 = |Y|$. Moreover, $\mathrm{N}(X) = Y$, so

$$|\mathrm{N}(X)| = |Y| = |X| - 1 < |X|,$$

which violates the assumption on the sizes of neighborhoods in the theorem. $\square$

With this theorem we have completely characterized those bipartite graphs that have a matching whose size coincides with the cardinality of the smaller color class. We can easily generalize the theorem in the following way.

Corollary 9.7. *Let $G = (A \cup B, E)$ be a bipartite graph such that $|\mathrm{N}(S)| \geq |S|d$ for all $S \subseteq A$, and some fixed integer $d \in \mathbb{Z}_{\geq 0}$. Then G contains a matching of size at least $|A| - d$.*

Proof. Let $G' := (V', E')$ be the graph with node set $A \cup (B \cup C)$ for a set C of size d and edges

$$E' \;=\; E \cup \{(a, c) \mid a \in A, c \in C\} \,.$$

Then G' is bipartite. If N_G denotes the neighborhood in G and $\mathrm{N}_{G'}$ the one in G', then for all $S \subseteq A$

$$|\mathrm{N}_{G'}(S)| \;=\; |\mathrm{N}_G(S)| \;+\; d \;\geq\; |S| \;-\; d \;+\; d \geq |S|\,,$$

so Hall's Theorem (Theorem 9.6) implies that G' has a matching M of size $|A|$. At most d edges of M have one endpoint in C, so G has a matching of size at least $|A| - d$. $\square$

Corollary 9.8. *A k-regular bipartite graph has a perfect matching.*

Proof. For all bipartite graphs $G = (A \cup B, E)$ and $S \subseteq A$ we have that

$$\sum_{u \in \mathrm{N}(S)} \deg(u) \;\geq\; \sum_{u \in S} \deg(u)\,.$$

Now in a k-regular graph the left side is $k|\mathrm{N}(S)|$ and the right side is $k|S|$. This implies the condition of Hall's Theorem (Theorem 9.6). Further,

$$\sum_{u\in B} \deg(u) = \sum_{u\in A} \deg(u)\,,$$

so A and B have the same size. □

From this corollary we can deduce even more. Namely, if we have a perfect matching M in a k-regular bipartite graph G, then we can define a new bipartite graph $G' := (V, E - M)$, that is $(k-1)$-regular. Hence, this graph has a perfect matching and we can continue this process until all edges are distributed into k disjoint perfect matchings.

Corollary 9.9. *A k-regular bipartite graph has k pairwise edge disjoint perfect matchings.* □

We may try to use the Theorem of Berge (Theorem 9.5) to construct a maximum matching in a bipartite graph. While this is possible, it is conceptually much simpler to reduce this problem to a flow problem and apply one of the maximum flow algorithms we have already seen in Chapter 8.

For the reduction to the Flow Problem let $G = (U_1 \cup U_2, E)$ be a bipartite graph. We construct a directed graph $D = (V, A)$ for this in the following way. Its node set V is $U_1 \cup U_2$ with two auxiliary nodes s and t. for each edge $e \in E$ we add a directed arc a_e from the node in U_1 to the node in U_2 to the set A. Further we add a directed arc $a_u^{(s)}$ from s to all nodes $u \in U_1$ and an arc $a_u^{(t)}$ from all nodes $u \in U_2$ to t.

On this new graph D we define a weight function $w : A \to \mathbb{R}$ that assigns a capacity of 1 to each arc and compute a maximum integral flow in this graph. We now add all edges $e \in E$ into our matching M whose corresponding directed arc a_e in the graph D has positive flow value (which then is necessarily 1). This approach is formalized in Algorithm 9.2.

Theorem 9.10. *Let $G = (U_1 \cup U_2, E)$ be a bipartite graph with n nodes and m edges. Then Algorithm 9.2 returns a maximum matching in time $\mathcal{O}(nm^2)$.*

Proof. If M is a matching in G, then setting the flow to 1 on all arcs a_e, $a_u^{(s)}$, and $a_v^{(t)}$ for $e = (u, v) \in M$ with $u \in U_1$ and $v \in U_2$ defines a valid flow.

Conversely, if all capacities are integral, then the Edmonds-Karp Algorithm returns an integral flow. As all capacities are one, the flow on an arc is either 0 or 1. As the inflow of a node $u \in U_1$ is either 0 or 1, at most one arc with tail u has a flow value of 1. Similarly, the outflow of a node $v \in U_2$ is either 0 or 1, so at most one arc with head v has a flow value of 1. This implies that the edges e corresponding to an arc a_e with positive flow value produce a matching.

Now in this equivalence the size of the matching and the flow value coincide, so a maximum flow must correspond to a maximum matching.

Algorithm 9.2: Bipartite Matching

```
Input  : undirected bipartite graph $G = (S \dot\cup T, E)$
Output : maximal matching $M$
// Construct a flow network:
$V \leftarrow S \cup T \cup \{s,t\}$
$A_1 \leftarrow \{(s,v) \mid v \in S\}$
$A_2 \leftarrow \{(v,w) \mid \{v,w\} \in E\}$
$A_3 \leftarrow \{(w,t) \mid w \in T\}$
$A \leftarrow A_1 \cup A_2 \cup A_3$
$D \leftarrow (V, A)$
foreach $a \in A$ do
    $c(a) \leftarrow 1$
// Use Edmonds-Karp:
$f \leftarrow$maximal flow on $(D, s, t, c)$
// Find the matching:
$M \leftarrow \emptyset$
foreach $a \in A_2$ do
    if $f(a) > 0$ then
        $M \leftarrow M \cup \{a\}$
```

The running time is immediate from the running time of the Edmonds-Karp Algorithm, which we determined in Theorem 8.15, as the initialization can be done in $\mathcal{O}(m + n) \subseteq \mathcal{O}(nm^2)$. □

We can use this correspondence between matchings in bipartite graphs and flows also to discover some structural results of bipartite graphs.

Definition 9.11. Let $G = (V, E)$ be an undirected graph. A *node cover* $C \subseteq V$ in G is a subset of the nodes so that each edge $e \in E$ is incident to at least one node of C, *i.e.*, $|e \cap C| \geq 1$ for all $e \in E$.

A node cover is *minimum* if it has the smallest size among all vertex covers of the graph.

Note that also here we could make a similar distinction between minimal and minimum as we did for matchings. The following theorem is a consequence of the node version of Menger's Theorem (Theorem 8.20).

Theorem 9.12 (König's Theorem). *In a bipartite graph the size of a maximum matching is the same as the size of a minimum node cover.*

Proof. Let A and B be the color classes of the bipartite graph G. Add a node s with an edge to all nodes in A, and a node t with edges to all nodes in B. Now a node cover of G is s-t-separating, while a matching in G defines a set of node disjoint paths from s to t. □

9.3 Minimum Cost Perfect Matching

In this section we consider another problem for bipartite graphs and look at the following task. We need an additional cost function on the edges.

Minimum Cost Perfect Matching Problem.

input: A bipartite graph $G = (A \cup B, E)$ and a cost function $c : E \to \mathbb{R}$.

problem: find a perfect matching in G (if it exists) of minimum total cost.

In the following let $G = (A \cup B, E)$ be a bipartite graph and $c : E \to \mathbb{R}$ be a cost function. We want to compute a perfect matching M in G such that

$$c(M) := \sum_{m \in M} c(m)$$

is minimal among all perfect matchings in G. The basic principle for our algorithm is to inductively construct the matching using appropriately chosen M-augmenting paths. For the construction of an M-augmenting path we define an auxiliary directed graph and do a shortest path computation. The auxiliary graph is defined as follows.

For some matching $M \subseteq E$ in the bipartite graph $G = (A \cup B, E)$, we define $G_M = (A \cup B, E_M)$ with weights $c_M : E_M \to \mathbb{R}$, where $E_M = E_1 \cup E_2$ with

$$\begin{aligned}
E_1 &:= \{(a,b) \in A \times B \mid \{a,b\} \in E \setminus M\},\\
E_2 &:= \{(b,a) \in B \times A \mid \{a,b\} \in E \cap M\},\\
c_M(a,b) &:= c(\{a,b\}) \qquad \text{if } (a,b) \in E_1,\\
c_M(b,a) &:= -c(\{a,b\}) \quad \text{if } (b,a) \in E_2.
\end{aligned}$$

Moreover, we define A_M and B_M to be the nodes in A and B, respectively, that are not contained in a matching edge.

Thus, we have forward arcs (*i.e.*, from A to B) for edges not in M and backward arcs for edges in M. The cost function is negated for edges in M. An alternating path in G corresponds to a directed path in $G(M)$. Its cost is the sum of the costs for edges not in M minus the costs of edges in M. Thus, this corresponds to the net cost of augmenting along this path. Again, the idea of Algorithm 9.3 is to iteratively augment along the shortest path.

Lemma 9.13. *If in algorithm Algorithm 9.3, P does not exist, there does not exist a perfect matching. Otherwise, P is M-augmenting. The new matching has cost $c(M) + c_M(P)$.*

Proof. A directed path starting in A_M and ending in B_M corresponds to an M-augmenting path, since “forward” and “backward” edges have to alternate, which corresponds to alternating edges not in M and edges in M. If there exists a perfect matching M', we have $|M'| > |M|$. Thus, there exists an M-augmenting path by Theorem 9.5.

Algorithm 9.3: Hungarian Algorithm

```
Input  : undirected bipartite graph $G = (A \cup B, E)$ with $|A| = |B|$, $c : E \to \mathbb{R}$
Output : cost minimal perfect matching $M$
1 $M = \emptyset$
2 while $|M| < |A|$ do
3     find shortest path $P$ between $A_M$ and $B_M$ w.r.t. $c_M$
4     if $P$ does not exist then
5         stop (no perfect matching exists);
6     augment $M$ along $P$
7 return $M$
```

Moreover, the augmented matching M' has cost:

$$c(M') = \sum_{e \in M'} c(e) = c(M) + \sum_{(a,b) \in P} c(a,b) - \sum_{(b,a) \in P} c(a,b) = c(M) + c_M(P).$$

This concludes the proof. □

Theorem 9.14. *Algorithm 9.3 computes a cost-minimal perfect matching, if it exists, in time $\mathcal{O}(n^2 m)$.*

Proof. By Lemma 9.13, the algorithm correctly detects whether there exists a perfect matching. Thus, we assume that it exits.

The matching M in iteration i has size i, since the size increases by one in each iteration. We now prove that M is a cost-minimal matching among all matchings of size i by induction with respect to i. For $i = 0$, $M = \emptyset$ and the claim is true.

Let M' be a matching with $|M'| = i+1$. By Theorem 9.5 there exists an M-augmenting path P' such that $P' \subseteq M' \cup M$, see the proof of Theorem 9.5. Consider

$$M' \triangle P' := \{e \in E \mid e \in M' \setminus P' \text{ or } e \in P' \setminus M'\},$$

which corresponds to an augmentation along P'. Thus, it is a matching and $|M' \triangle P'| = i$. By construction of P, we have $c_M(P) \le c_M(P')$ and by the induction hypothesis $c(M) \le c(M' \triangle P')$. Consequently, by Lemma 9.13

$$c(M') = c(M' \triangle P') + c_M(P') \ge c(M) + c_M(P).$$

Thus, the new matching $M \triangle P$ has smallest cost among all matching of size $i + 1$.

The shortest paths can be found using the Bellman-Ford algorithm and there can be at most n iterations, which yields an $O(n^2 m)$ running time. □

In fact, using one Bellman-Ford computation as a preprocessing the costs can be transformed to be non-negative. This allows to use Dijkstra's algorithm instead and the running time can be improved (to $\mathcal{O}(nm + n^2 \log n)$).

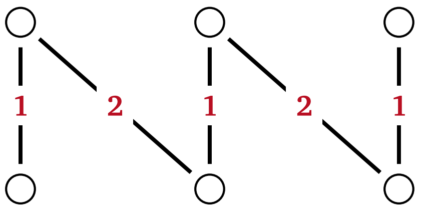


Figure 9.4: A bipartite graph with a weight function. It has a perfect matching of maximum weight 3, but a matching of maximal weight 4.

Observe that we can directly use the Hungarian method also to compute a perfect matching of maximum weight in a bipartite graph. This will solve the following problem.

Maximum Weight Perfect Matching Problem.

input:	A bipartite graph $G = (A \cup B, E)$ and a weight function $c : E \to \mathbb{R}$.
problem:	find a perfect matching in G (if it exists) of maximum total weight.

Indeed, if G is a bipartite graph with n nodes, matching number $m := \nu(G)$, weight function $c : E \to \mathbb{R}$ and $R > \max(c(e) \mid e \in E)$, then $\sum_{e \in M} c(e)$ is maximal among all perfect matchings in G if and only if $\sum_{e \in M} R - c(e) = mR - \sum_{e \in M} c(e)$ is minimal among all perfect matchings in G.

9.4 Maximum Weight Matching Problem

In the previous section we where concerned with perfect matchings. Sometimes, it is not necessary to match all nodes, if there is a matching of smaller size but larger weight. This may clearly happen, see Figure 9.4.

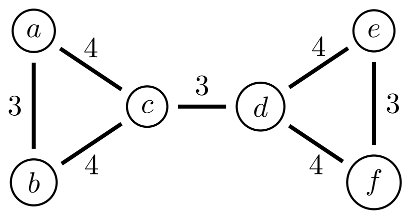
Maximum Weight Matching Problem.

input:	A bipartite graph $G = (A \cup B, E)$ and a weight function $c : E \to \mathbb{R}$.
problem:	find a matching in G of maximum total weight.

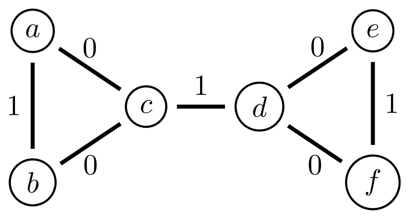
We show that this can be reduced to the previous problem and thus can be solved with the Hungarian method.

For this, let A and B the two color classes of the graph. We construct a new bipartite graph G' in two steps. First, we add nodes to the smaller color class to obtain two color classes A' and B' of the same size. In a second step, we add all missing edges with a weight of 0.

If we now have matching M of maximum weight in G, then we can extend this to a *perfect* matching of the same weight by adding some of the new edges of cost 0. Conversely, we can use the Hungarian method to compute a maximum weight perfect matching in G'. Removing edges of weight 0 from this matching reduces to a matching of the same weight in G. This leads to the following theorem.



(a) Maximum weight matching: use the three edges of weight 3.



(b) The same graph, but with all weights subtracted from 4. A maximal matching of minimal weight uses two of the edges of weight 0.

Figure 9.5: An example that the simple transfer between minimum and maximum weight matchings does not work for general matchings instead of perfect matchings

Theorem 9.15. *We can solve the maximum weight matching problem on a connected bipartite graph G in time $O(n^2m)$ using the Hungarian algorithm.* □

Observe that the correspondence used to transfer a maximum cost perfect matching into a maximum weight perfect matching does not work for this problem. For this consider the graphs in Figure 9.5.

10 Complexity

Contents

All of the algorithms discussed so far were “efficient” in the sense that their worst case running time (*i.e.,* number of elementary operations) can be bounded by a polynomial in the input size, which is usually given by the number n of nodes and the number m of edges in our graph. However, there are problems for which no such algorithm could be found so far – although considerable investment of research. In this chapter, we will present a mathematical theory that tries to explain this behavior.

Recall the following definitions from Chapter 3:

- ▷ A *problem* Π is given by a set $\mathcal{I}$ of instances and a family of sets S_I for every $I \in \mathcal{I}$, the set of *feasible solutions* of the instance I. See Definition 3.1.
- ▷ For the following we say that the *input size [Eingabegröße]* of a problem is the number of bits in a binary representation
- ▷ An *algorithm* $\mathcal{A}$ is given by finite list of instructions that operate on given data (Definition 3.3).
- ▷ The algorithm $\mathcal{A}$ solves a problem Π if for any input $I \in \mathcal{I}$ it determines in finite time an output from S_I (or stops without if there is none). See Definition 3.3 for the full statement.
- ▷ We have discussed one computational model in Section 3.3, the *RAM-Model*. There are others, as the *Turing model [Turingmodell]*, but for our purposes the actual choice is not really important.
- ▷ By Definition 3.4 the *running time* of an algorithm is given by a function

$$f_{\mathcal{A}}(k) := \max \left\{ \begin{array}{c} \text{number of elementary operations} \\ \text{performed by } \mathcal{A} \text{ on input } I \end{array} \;\middle|\; I \in \mathcal{I}, \text{size of } I \le k \right\}$$

10.1 P and NP

We will now introduce a classification of problems Π. We distinguish two types of problems

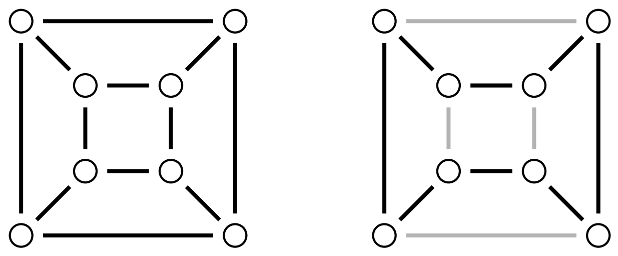


Figure 10.1: A graph and a Hamilton cycle.

- ▷ *decision problems [Entscheidungsprobleme]*: $S_I = \{0, 1\}$ ($= \{\text{yes}, \text{no}\}$) for all I
- ▷ *optimization problems [Optimierungsprobleme]*: There is an *objective function [Zielfunktion]* $c_I : S_I \to \mathbb{Q}$. The goal is: Find $s \in S_I$ that maximizes (minimizes) c_I.

Example. Here are two examples, the first is a decision problem, the second an optimization problem.

(i) *Hamilton cycle [Hamilton-Kreis]*

Instance:	graph G
Question:	Is there a cycle that uses every node of G?

See Figure 10.1 for an example.

(ii) *Travelling Salesman Problem (TSP) [Problem des Handlungsreisenden]*

Instance:	graph G (often $G = K_n$) with weight function $\ell : E(G) \to \mathbb{Q}$
Question:	What is the shortest (sum of the weights) Hamilton cycle in G? Such a tour is called a *TSP-Tour*.

The basic statements of Complexity theory use decision problems. Optimization problems can be transformed into decision problems via an additional parameter, *e.g.* "Is there a TSP-tour of length $\le K$"? This decision problem is denoted by (TSP_DEC).

We define the following *complexity classes [Komplexitätsklassen]*:

Definition 10.1. A decision problem Π belongs to the *complexity class*

- ▷ P if it can be solved in polynomial time. (P: **p**olynomial time)
- ▷ NP if there exists a polynomial $p(n)$ and an (certifying) algorithm $\mathcal{A}$, such that
  - For each "yes" instance I there exists a *certificate* $C(I)$ with $\langle I \rangle \le p(\langle I \rangle)$.
  - Given I and $C(I)$, algorithm $\mathcal{A}$ certifies the answer "yes" in a running time of at most $p(\langle I \rangle)$. (NP: **n**on-deterministic **p**olynomial time)

The class co-NP exchanges "yes" and "no".

Remark.

- ▷ The definition of NP is asymmetric with respect to "yes" and "no". This reflects the fact that very often proving one is easy, while the other requires more effort.
- ▷ The *complementary* problem $\overline{\Pi}$ of Π exchanges "yes" and "no". Thus $\overline{\Pi} \in$ co-NP $\Leftrightarrow$ $\Pi \in$ NP.

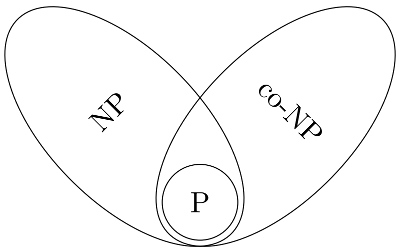


Figure 10.2: Relations between P, NP, and co-NP

.

▷ The class NP can equivalently be defined by a modified (non-deterministic) Turing machine in which at certain steps the next step is not deterministically defined.

Proposition 10.2. $\mathrm{P} \subseteq \mathrm{NP} \cap \text{co-NP}$

Proof. If $\Pi \in \mathrm{P}$, there exists a polynomial time algorithm $\mathcal{A}$ that verifies both "yes" or "no" answer (no certificate is needed). □

One of the fundamental open problem in mathematics/theoretical computer science is the question whether P = NP. The Clay institute has set a prize of one million dollars for its solution[1]. It is widely believed that P ≠ NP, *i.e.,* that the two sets differ, but despite many attempts no proof is known, the problem is still wide open.

A possible "map" of the complexity world might look like given in Figure 10.2. Note however that this is still a very simplistic picture. In the attempt to understand the fundamental differences in the complexity of various problems many more categories have been introduced that help to distinguish between problems.

Example. Here are some problems together with the complexity class they belong to.

(i) *Hamilton cycle problem* $H \in \mathrm{NP}$: A certificate is given by a cycle of size $|V|$. Checking whether this is a Hamilton cycle can be done in linear time in $|V|$. It is unknown whether H is in co-NP.

(ii) *Travelling Salesman Problem*: The associated decision problem "Is there a tour of length at most K?" is in NP.

(iii) *Linear Programming*: Given a matrix $A \in \mathbb{Z}^{m\times n}$, vectors $c \in \mathbb{Z}^n$, $b \in \mathbb{Z}^m$, and a number $z \in \mathbb{Q}$, does there exist a vector x such that $Ax \le b$ and $c^T x \ge z$? This problem is in P (but *not* via the simplex method). See the lecture "Introduction to Optimization".

Integer Programming: Same as linear programming, but x is required to be integral. This problem is in NP (but not known to be in P).

(iv) A quite famous problem in NP is the *Satisfiability Problem (SAT)*. In this problem we are given a set of variables $x_1, \dots, x_n$ that can take values in {`true, false`} and a set of *clauses*. We need to find an assignment of `true` or `false` to each

[1] see `http://www.claymath.org/millennium-problems/p-vs-np-problem`

variable in such a way that each clause is *satisfied*. A clause in the variables $x_1, \ldots, x_n$ is an expression of the form

$$x_1 \vee x_4 \vee \neg x_7 \vee x_9$$

where $\vee$ is the logical `or` and $\neg$ is negation. A *literal* is a variable or its negation. Thus, clauses can be identified with the set of literals it contains. A clause $x_i \vee x_j$ is satisfied if at least one variable is `true`, and $\neg$ exchanges `true` and `false`. So the above clause is, *e.g.*, satisfied for (x_1, x_4, x_7, x_9) = (`false`, `false`,`false`,`false`).

A satisfiability problem consists of a finite list of such clauses and we want to know whether we can assign the variables in such a way that all clauses in the list are satisfied simultaneously. For example, the left of the following systems is satisfiable, while the right is not.

$$\begin{array}{c} x_1 \vee x_2 \vee x_3 \\ \neg x_1 \vee x_2 \vee x_3 \\ x_1 \vee \neg x_2 \vee \neg x_3 \end{array} \qquad\qquad \begin{array}{c} x_1 \vee x_2 \\ x_1 \vee \neg x_2 \\ \neg x_1 \end{array}$$

The SAT problem is in NP.

10.2 NP-Completeness

Among the problems in NP some problems have proved to be particularly important in the sense that they are characteristic for the distinction between P and NP: If we can prove for one such problem that it is polynomial time solvable then we immediately have shown this for all others as well. In a way those are the “hardest” problems in NP. Let us formalize this.

Definition 10.3. Let Π_1, Π_2 be decision problems. We say that Π_1 is *polynomial time (Karp) reducible* to Π_2, written $\Pi_1 \propto \Pi_2$, if there is a polynomial time algorithm that returns for each instance I_1 of Π_1 an instance I_2 of Π_2 such that

$$I_1 \text{ is a yes-instance} \iff I_2 \text{ is a yes-instance.}$$

Lemma 10.4.

(i) $\Pi_1 \propto \Pi_2$, $\Pi_2 \propto \Pi_3 \Rightarrow \Pi_1 \propto \Pi_3$.

(ii) $\Pi_1 \propto \Pi_2$ *and* $\Pi_2 \in \mathrm{P} \implies \Pi_1 \in \mathrm{P}$.

(iii) $\Pi_1 \propto \Pi_2$ *and* $\Pi_2 \in \mathrm{NP} \implies \Pi_1 \in \mathrm{NP}$.

Example 10.5. We can reduce H to TSP_DEC: Given $G = (V, E)$ let K_n be the complete graph on V. Define

$$w(e) = \begin{cases} 0 & \text{if } e \in E, \\ 1 & \text{if } e \notin E. \end{cases}$$

Then G has a Hamilton cycle if and only if K_n with weight w has a TSP tour of length ≤ 0.

Definition 10.6. A decision problem Π is NP-*complete [*NP-*vollständig]* if

(i) $\Pi \in \mathrm{NP}$ and

(ii) $\Pi' \propto \Pi$ for all $\Pi' \in \mathrm{NP}$.

Let NPC be the set of all NP-complete problems.

There exists at least one NP-complete problem:

Theorem 10.7 (Cook, 1971)**.** *SAT is* NP*-complete.*

The proof idea is to show that every non-deterministic Turing machine can be modelled as a SAT instance.

Lemma 10.8.

(i) $\mathrm{P} \cap \mathrm{NPC} \neq \varnothing \implies \mathrm{P} = \mathrm{NP}$.

(ii) $\Pi \in \mathrm{NPC}, \Pi' \in \mathrm{NP}, \Pi \propto \Pi' \implies \Pi' \in \mathrm{NPC}$.

Proof.

(i) Let $\Pi \in \mathrm{P} \cap \mathrm{NPC}$. Then $\Pi \in \mathrm{NP}$. Let $\Pi' \in \mathrm{NP}$ any other problem. Then by definition $\Pi' \propto \Pi$ and thus $\Pi' \in \mathrm{P}$ by Lemma 10.4.

(ii) This follows from transitivity. □

Thus, the standard way to prove NP-completeness of a new problem is to reduce some NP-complete problem to it. For example, H, TSP_DEC, and Integer Programming are NP-complete.

We here show that the following problem is NP-complete:

Vertex Cover Problem
Given: undirected graph $G = (V, E)$ and positive integer K.
Question: Does there exist a node set $V' \subseteq V$ such that $|V'| \leq K$ such that each edge of G is incident to at least one node in V'?

We will use the following NP-complete problem:

3SAT
Given: Set of variables $X = \{x_1, \ldots, x_n\}$ and clauses $C = \{c_1, \ldots, c_m\}$ such that each clause contains exactly three literals.

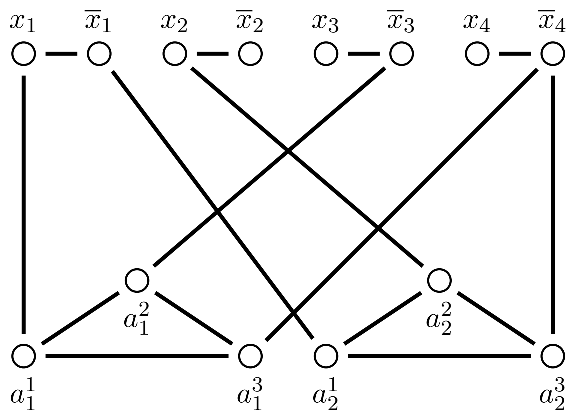


Figure 10.3: Example: Reduction from 3SAT to Vertex Cover: $c_1 = \{x_1, \overline{x}_3, \overline{x}_4\}$, $c_2 = \{\overline{x}_1, x_2, \overline{x}_4\}$.

Question: Does there exist a truth assignment to the variables such that each clause contains at least one true literal?

It can be shown – in a bit technical proof – that 3SAT is NP-complete by reduction of the SAT problem. We now obtain:

Theorem 10.9. *The Vertex Cover Problem is* NP-*complete.*

Proof. We first note that vertex cover is in NP.

We proceed by reduction from the 3SAT problem. So, given a 3SAT instance with variables $X = \{x_1, \dots, x_n\}$ and clauses $C = \{c_1, \dots, c_m\}$, we have to construct a graph $G = (V, E)$ and K in polynomial time such that the answer to the 3SAT instance is "yes" if and only if the answer to the Vertex Cover instance is "yes".

The node set is

$$V := \{x, \overline{x} \mid x \in X\} \cup \{a_j^1, a_j^2, a_j^3 \mid j = 1, \dots, m\}.$$

We have edges $\{x, \overline{x}\}$ for each $x \in X$. Moreover, for each clause c_j, we add edges forming a triangle:

$$\{a_j^1, a_j^2\}, \{a_j^2, a_j^3\}, \{a_j^1, a_j^3\}.$$

The three nodes a_j^1, a_j^2, a_j^3 represent the three literals in c_j.

Moreover, let c_j consist of the literals x_j, y_j, and z_j. Then, we add the following edges:

$$\{a_j^1, x_j\}, \{a^2, y_j\}, \{a^3, z_j\}. \tag{10.1}$$

Finally, we define $K := n + 2m$. See Figure 10.3 for an illustration. The construction can clearly be carried out in polynomial time.

We now show that if $V' \subseteq V$ is a vertex cover with $|V'| \le K$, then the 3SAT instances has answer "yes". Each vertex cover must either contain x_i or $\overline{x}_i$ for all i. Moreover, it must contain at least two nodes among $\{a_j^1, a_j^2, a_j^3\}$ for all j. Thus, each vertex cover has size at least $n + 2m = K$. Since $|V'| \leq K$, V' must in fact contain exactly one node in the first and exactly two nodes in the second case. We then simply set variable x_i to be true if $x_i \in V'$ and to false otherwise.

Now consider the three edges in (10.1). At most two of these edges can be covered by nodes in $\{a_j^1, a_j^2, a_j^3\} \cap V'$. To cover the third, we need to cover the third literal node by using on of x_j or $\overline{x}_i$. Thus, a least one literal is true for each clause.

Conversely, assume that there exists a satisfying assignment for the 3SAT instance. We then include x_i or $\overline{x}_i$ in V' depending on the truth value of x_i. Then at least one of the three edges in (10.1) is covered. We then pick two nodes among each $\{a_j^1, a_j^2, a_j^3\}$ to cover the remaining edges. This gives a vertex cover V' of size K.

This shows a Karp-reduction from 3SAT to vertex cover. By Lemma 10.8, we have shown that vertex cover is NP-complete. □

Corollary 10.10. *The problem to decide whether there exists a clique of size at least K is* NP*-complete. Moreover, the problem to decide whether there exists a* stable set *of size at least K is* NP*-complete. (A stable set is a set of nodes such that no two are connected by an edge.)*

Proof. It is quite easy to show: $V' \subset V$ is a vertex cover in a graph $G = (V, E)$ if and only if $V \setminus V'$ is a clique in the complement graph $\overline{G} := \{V, \{\{u, v\} \mid u, v \in V,\ \{u, v\} \notin E\}\}$. Moreover, V' is a vertex cover if and only if $V \setminus V'$ is a stable set in G. Thus, all three problems are essentially the same. Moreover, the reductions can be performed in polynomial time (in particular, construction of $\overline{G}$). □

Since P and NP only contain decision problems, we want to extend complexity theory, *e.g.*, to handle optimization problems.

Definition 10.11. A problem Π is *Touring-reducible* on another problem Π', if there exists a polynomial time algorithm for Π that can call an "oracle" that solves Π' and costs $\mathcal{O}(1)$ time. Notation: $\Pi \propto_T \Pi'$.

Remark 10.12. $\Pi \propto \Pi' \implies \Pi \propto_T \Pi'$. (Solve Π' and return result.)

Definition 10.13. A problem Π is NP-*hard*, if each problem $\Pi' \in$ NP can be Touring-reduced on Π, i.e., $\Pi' \propto_T \Pi$.

NP-hard problems are at least as hard to solve as NP-complete problems.

Index