3 Basics on Graphs

Graphs are a central structure in Discrete Mathematics with numerous real-world applications. This comes at no surprise, as graphs and networks are ubiquitous in everyday life and can be used to model transport networks, social networks, telecommunications networks, and neural networks, just to name a few examples. By exploiting the underlying graph-theoretical structures, problems in such networks can often be solved fast even for enormously large instances.

The aim of this chapter is to provide a very brief introduction to some of the basics linked to graphs and networks. After some motivational examples, we first introduce terminology and notation commonly used in Graph Theory. Moreover, we talk about the most common data structures to store graphs. This allows us to understand the input size of a problem that involves graphs and also clarifies how much time certain simple graph operations take. Finally, we discuss one of the most basic graph algorithms, namely breadth-first search, and some of its implications. This first algorithmic example also helps us to combine and better understand the previously introduced concepts, and is a first non-trivial example of how to analyze a graph algorithm.

We would like to emphasize that the terms "graph" and "network" are used interchangeably in many contexts. Sometimes, however, a network is used to describe graphs with additional information, for example on its edges or vertices.

3.1 Some motivational examples

In this introductory section we describe some classical problems that can be modeled and solved using graphs.

Example 3.1: Road network

Five cities, A, B, C, D, and E, are linked by the road network shown in Figure 3.1. The cities are represented by *vertices*, the streets as connections between two vertices, which are also called *edges*. The edges are labeled with the respective distances (in km) between the connected cities.

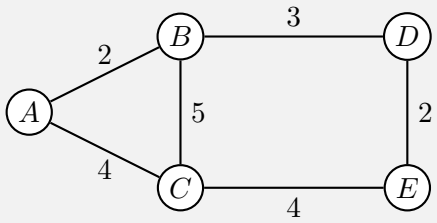


Figure 3.1: A road network represented as a graph.

One typical problem is to find the distance (i.e., the length of a shortest path) between any two cities as well as a corresponding path of that length. For example, the distance between city A and city E is 7 kilometers, and A-B-D-E is a possible path between A and E with this length.

The schematic representation of the road network in Figure 3.1, consisting of vertices and edges, is a *graph*. The edges here are labeled with numbers—representing the distances in the example— so we are talking about a *weighted graph*. If Example 3.1 contained a one-way street, we could illustrate this in the graph by adding an arrow pointing in the allowed direction. Such an edge is called *directed edge*, or simply *arc*. If every edge is directed in a graph, we are talking about a *directed graph*.

Example 3.2: Utility network

Between five locations, a supply network is to be set up (e.g., for water, gas, electricity, or internet access). The goal is that each pair of locations is connected, where a connection can go through other intermediate stations. In Figure 3.2, the locations are represented as vertices and the possible connections as edges between the vertices. The numbers on the edges correspond to the installation cost for the respective connection. The problem is to find a minimum cost set of edges that connect all locations. The highlighted edges in Figure 3.2 show an optimal solution for this particular example.

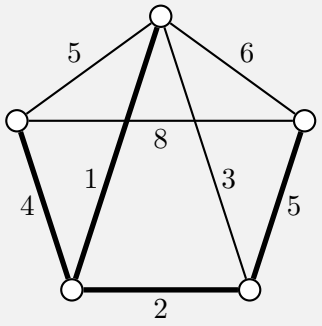


Figure 3.2: A utility network respresented as a graph.

Example 3.3: Assignment problem

A set of n jobs $\{j_1, \ldots, j_n\}$ are to be completed by m workers $\{a_1, \ldots, a_m\}$. Not every job can be completed by every worker: for each job, there is a given list of workers who are qualified to perform it. The question is whether it is possible to assign to each job one worker who is qualified for it. Each worker can be assigned to at most one job. Figure 3.3 shows a toy problem of this type with five jobs and six workers. The problem is represented as a graph. The top row of vertices corresponds to the jobs and the bottom one to the available workers. There is an edge between a job and a worker if and only if the job can be performed by the worker. The highlighted edges indicate a possible allocation assigning to each job one worker.

job	qualified workers
j_1	a_1, a_3, a_4
j_2	a_1, a_2, a_3, a_5
j_3	a_4, a_5, a_6
j_4	a_3, a_4
j_5	a_4, a_5, a_6

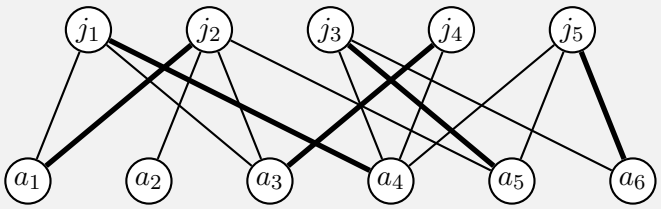


Figure 3.3: An assignment problem and its representation as a graph.

Example 3.4: Flow problem

In a system of water pipes there is a source s and a sink t. The system consists of junctions and pipes connecting these points. Each pipe can be used to let water flow in either one of the two directions. Moreover, each pipe has a maximum throughput, i.e., a maximum amount of water that can flow through the pipe per time unit. Junctions cannot store water. Hence, for each junction, the water inflow must be equal to its outflow. A canonical optimization problem in this setting is to determine the largest so-called s-t flow, which is the water flow—in terms of water per time unit—from s to t.
Figure 3.4 shows an example of a pipe network. The edges represent pipes and the vertices junctions, except for the two labelled vertices, which represent the source s and the sink t. The black numbers on the edges, i.e., the ones on the right-hand side, are the known maximum throughput rates for the pipes. This concrete example allows a maximum s-t flow of 17 units. The blue numbers, which are to the left of the throughput values, indicate a possible flow in the direction of the blue arrows that realizes this maximum.

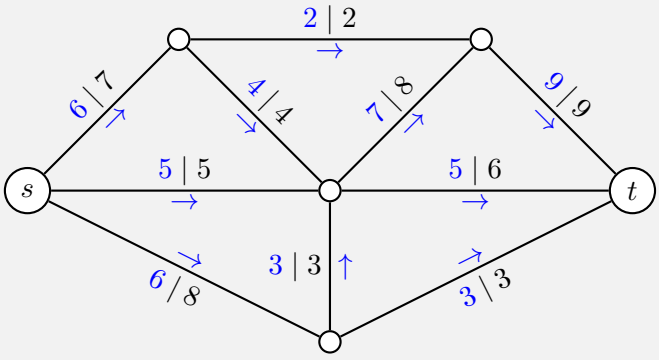


Figure 3.4: A flow problem together with a maximum s-t flow represented as a graph.

Example 3.5: Connection problem

In a telecommunications network, there is a central server in a network of locations. There are connections between the locations of different bandwidths. At certain stations are customers who would like to each have a connection of bandwidth 1 to the server. The task is to determine a maximum number of customers that can simultaneously be served by the server with a unit bandwidth each. The left part of Figure 3.5 shows such a problem where the server and locations are represented as vertices of a graph and the edges correspond to the connections. The numbers on the edges indicate the bandwidth of the respective connections. The tokens above vertices represent the clients. As highlighted on the right-hand side of Figure 3.5, a maximum of 5 clients can be connected to the server simultaneously.

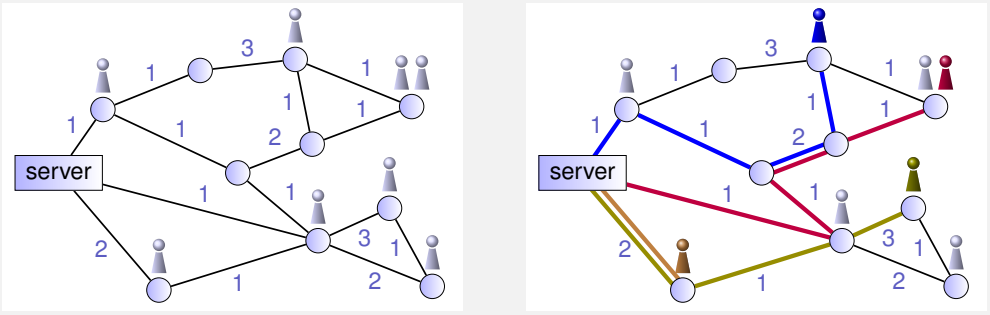


Figure 3.5: A connection problem and an optimal solution for it.

Example 3.6: Round trip

Our goal is to plan a road trip through the USA going through 20 given cities. The road trip should be a round trip, i.e., start and endpoint must coincide. The task is to determine a road trip that minimizes the driving time. The driving time from any of the 20 cities to any other one is known upfront. Figure 3.6 shows an example.

Figure 3.6: A possible road trip through 20 given cities in the USA.

Here, the problem is presented as a graph with 20 vertices, one per city, and all possible edges between these vertices. Each edge is assigned a value, representing the travel time between its endpoints. With this abstraction, the shortest round trip problem, also famously known as the Traveling Salesman Problem, can be formulated as follows: Find a shortest cycle in the graph that goes through all vertices.

Example 3.7: Drilling circuit boards

One of the steps in the production process of an electronic circuit board is the drilling of holes. This is performed by automatic drilling machines, whose drill head wanders over the circuit board and drills one hole after the other. In order to optimize the time required for this task, the distance traveled by the drill head should be minimized. This problem is very similar to the round trip problem depicted in Example 3.6 and can be modeled in an analogous way. However, in contrast to the round trip problem, we do not need that the drill head returns back to the starting point after completion of the drilling. Figure 3.7 shows a section of a circuit board and a possible drilling order for that area.

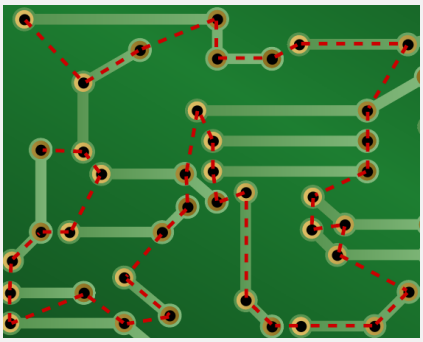
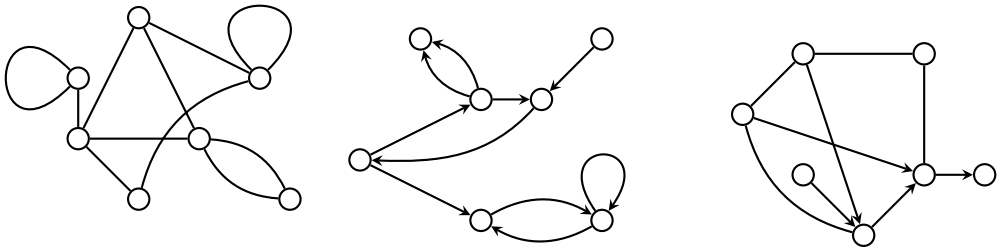


Figure 3.7: Section of a circuit board with a possible drilling order.

3.2 Basic terminology and notation

A graph consists of a set of *vertices* and connections between pairs of vertices, which are called *edges*. Edges can be *undirected* or *directed*, that is, oriented from one endpoint to another. A graph containing only undirected edges is an *undirected graph* (Figure 3.8a). If all edges are directed, we speak of a *directed graph* (Figure 3.8b). It is also possible that undirected and directed edges occur simultaneously. In this case we are talking about a *mixed graph* (Figure 3.8c).

Our main focus will be on directed and undirected graphs, which we will formally introduce shortly. In the context of graphs, various notations and terminology are used throughout the literature. We focus on one widespread formalization in the following.



(a) An undirected graph. (b) A directed graph. (c) A mixed graph.

Figure 3.8: Different types of graphs.

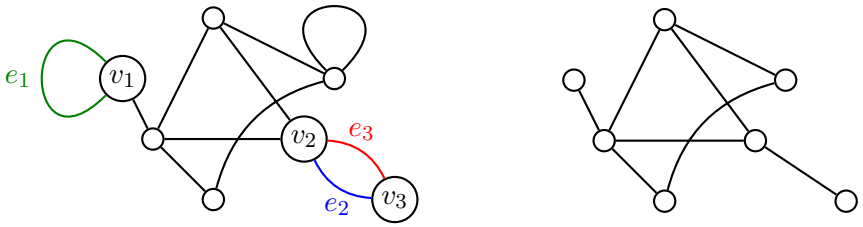
Undirected graphs

An *undirected* graph $G = (V, E)$ is a pair consisting of a finite set V of *vertices* and a finite set E of *edges*. Each edge $e \in E$ has two endpoints $u, v \in V$, which may be identical. The endpoints of an edge are denoted by $\text{endpoints}(e) = \{u, v\}$.

An edge $e \in E$ is called a *loop* if its two endpoints are identical, or in other words if $|\text{endpoints}(e)| = 1$. Two different edges $e_1, e_2 \in E$ are called *parallel* if $\text{endpoints}(e_1) = \text{endpoints}(e_2)$. An (undirected) graph without loops and parallel edges is called a *simple graph*. For example, in Figure 3.9a, we have

$$\text{endpoints}(e_1) = \{v_1\} \quad \text{and} \quad \text{endpoints}(e_2) = \text{endpoints}(e_3) = \{v_2, v_3\} \ ;$$

hence e_1 is a loop and the edges e_2 and e_3 are parallel. Thus, the graph in Figure 3.9a is not a simple graph. The graph in Figure 3.9b, on the other hand, has neither loops nor parallel edges and is thus simple.



(a) A non-simple graph. (b) A simple graph.

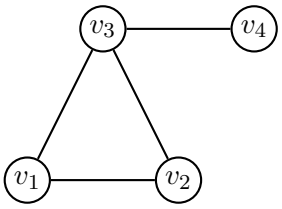
Figure 3.9: Non-simple and simple graphs.

In simple graphs, we often describe an edge as the set of its endpoints, i.e.,

$$E \subseteq \binom{V}{2} \coloneqq \{\{u, v\} \colon u, v \in V,\ u \neq v\} \ .$$

Figure 3.10 illustrates this notation, which is unambiguous in simple graphs, because for any set of endpoints there is at most one edge corresponding to it. In non-simple graphs, however,

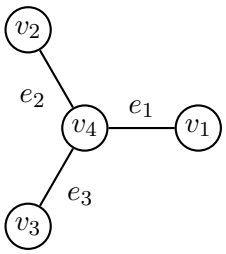
ambiguities may appear when referring to one of several parallel edges by its endpoints. Nevertheless, we also frequently use the above notation in the context of non-simple graphs in cases where there is little danger of confusion.



$$G = (V, E)$$
$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}, \{v_3, v_4\}\}$$

Figure 3.10: Notation for simple graphs. Due to its convenience, this notation is also often used in the context of non-simple graphs.

An edge $e \in E$ and a vertex $v \in V$ are called *incident* if v is one of the endpoints of e, i.e., if $v \in \text{endpoints}(e)$. Two distinct vertices $u, v \in V$ are called *adjacent* (or *neighboring*) if there is an edge e with $\text{endpoints}(e) = \{u, v\}$. With the notation introduced above, u and v are adjacent if and only if $\{u, v\} \in E$. Figure 3.11 shows examples of incident and adjacent vertices.



v_1 is incident to e_1, but not to e_2 or e_3.

v_4 is adjacent to v_1, v_2, and v_3.

v_1 and v_2 are not adjacent.

Figure 3.11: Incidence and adjacency in the context of vertices.

Directed graphs

An *directed graph* $G = (V, A)$ consists of a finite set V of vertices and a finite set A of *directed edges*, which we call *arcs* to more easily distinguish them from their undirected counterparts. Every arc $a \in A$ has a *tail* $\text{tail}(a) \in V$ and a *head* $\text{head}(a) \in V$ and points from tail to head. In particular, the endpoints are given by $\text{endpoints}(a) = \{\text{tail}(a), \text{head}(a)\}$.

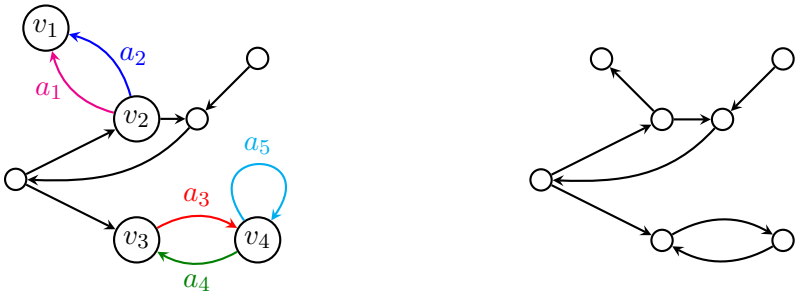
An arc a is a *loop* if $\text{tail}(a) = \text{head}(a)$. Two distinct arcs a_1 and a_2 are called *parallel* if $\text{tail}(a_1) = \text{tail}(a_2)$ and $\text{head}(a_1) = \text{head}(a_2)$. They are called *antiparallel* if $\text{tail}(a_1) = \text{head}(a_2)$ and $\text{head}(a_1) = \text{tail}(a_2)$. As for undirected graphs, a directed graph $G = (V, A)$ is called *simple* if it contains neither loops nor parallel arcs. However, a simple directed graph may contain antiparallel arcs. Figure 3.12 shows examples of a non-simple and a simple directed graph. In Figure 3.12a, we have

$$\left.\begin{aligned} \text{head}(a_1) = \text{head}(a_2) = v_1 \\ \text{tail}(a_1) = \text{tail}(a_2) = v_2 \end{aligned}\right\} \quad \Longrightarrow \quad a_1 \text{ and } a_2 \text{ are parallel,}$$

$$\left.\begin{aligned} \text{tail}(a_3) = \text{head}(a_4) = v_3 \\ \text{head}(a_3) = \text{tail}(a_4) = v_4 \end{aligned}\right\} \quad \Longrightarrow \quad a_3 \text{ and } a_4 \text{ are antiparallel,}$$

$$\left.\begin{aligned} \text{head}(a_5) = \text{tail}(a_5) = v_4 \\ \text{or, equivalently: } \text{endpoints}(a_5) = \{v_4\} \end{aligned}\right\} \quad \Longrightarrow \quad a_5 \text{ is a loop.}$$

Hence, because the directed graph in Figure 3.12a has both parallel edges and moreover a loop, it is not simple. The directed graph in Figure 3.12b is simple.



(a) A non-simple directed graph.

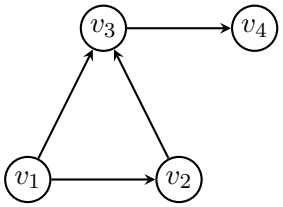
(b) A simple directed graph.

Figure 3.12: Non-simple and simple directed graphs.

Analogously to the undirected case, we can use a simplified notation for arcs that is unambiguous in simple graphs. More precisely, we can describe an arc $a \in A$ as an ordered pair $a = (u, v)$ of vertices, where $u = \text{tail}(a)$ and $v = \text{head}(a)$. With this notation it holds that

$$A \subseteq \{(u, v) \colon u, v \in V,\ u \neq v\} \ ,$$

where A is the arc set of a simple directed graph $G = (V, A)$. As before, this notation is unambiguous only if no parallel arcs exist. Similar to the undirected case, we nevertheless use this convenient notation also for directed graphs with parallel arcs, as long as there is no danger of ambiguity. Figure 3.13 illustrates this notation.



$$G = (V, A)$$
$$V = \{v_1, v_2, v_3, v_4\}$$
$$A = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4)\}$$

Figure 3.13: Notation for simple directed graphs. Due to its convenience, this notation is also used in the context of non-simple graphs.

Basic notation and terminology

We continue with some very common and basic notation and terminology in the context of graphs. Because of the close relation between undirected and directed graphs, when it comes to

terminology and notation, we discuss both cases together. In parenthesis, we indicate whether the introduced notation is commonly used for undirected (undir.) graphs, directed (dir.) graphs, or both (undir./dir.).

- (undir./dir.) Edges that *cross* the subset of vertices $S \subseteq V$:
$$\delta(S) := \begin{array}{l} \{\{u,v\} \in E \colon |\{u,v\} \cap S| = 1, u \neq v\} \ , \\ \{(u,v) \in A \colon |\{u,v\} \cap S| = 1, u \neq v\} \ . \end{array}$$
- (dir.) Arcs leaving the vertex set $S \subseteq V$:
$$\delta^+(S) := \{(u,v) \in A \colon u \in S,\ v \notin S\} \ .$$
- (dir.) Arcs entering the vertex set $S \subseteq V$:
$$\delta^-(S) := \{(u,v) \in A \colon u \notin S,\ v \in S\} \ .$$

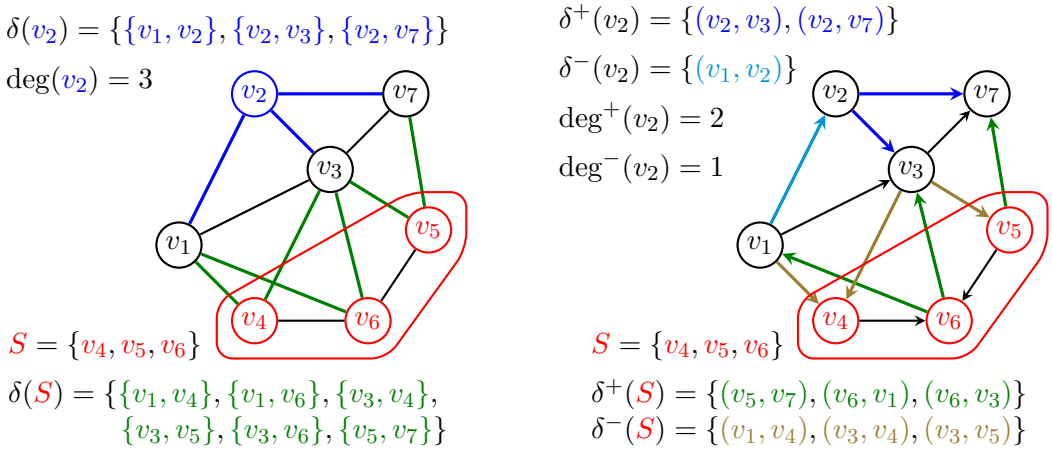
In particular, for a directed graph $G = (V, A)$ and a subset of vertices $S \subseteq V$, it always holds that $\delta(S) = \delta^+(S) \cup \delta^-(S)$.

For a single vertex $v \in V$, we use the following shorthands of the above notation:

$$\delta(v) := \delta(\{v\}) \ , \quad \delta^+(v) := \delta^+(\{v\}) \ , \quad \text{and} \quad \delta^-(v) := \delta^-(\{v\}) \ .$$

- (undir./dir.) *Degree* of v: $\deg(v) := |\delta(v)| + 2 \cdot |\{\text{loops at } v\}|$.
- (dir.) *Outdegree* of v: $\deg^+(v) := |\delta^+(v)| + |\{\text{loops at } v\}|$.
- (dir.) *Indegree* of v: $\deg^-(v) := |\delta^-(v)| + |\{\text{loops at } v\}|$.

Figure 3.14 exemplifies the introduced notation.



(a) Undirected graph. (b) Directed graph.

Figure 3.14: Crossing edges and vertex degrees.

Having introduced the most important notations for dealing with graphs, we now present two simple yet very useful properties of the vertex degrees occurring in a graph.

Property 3.8: Sum of all vertex degrees

In an undirected graph $G = (V, E)$, the sum of the degrees of all vertices is even.

Proof of Property 3.8. If we sum up the vertex degrees, each edge is counted twice, once for each of its endpoints. Consequently

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

implying that the sum of all vertex degrees is even. □

The proof of Property 3.8 is an example of double counting. A direct implication of Property 3.8 is the so-called *handshaking lemma*.

Property 3.9: Handshaking lemma

In every undirected graph, the number of odd-degree vertices is even.

Proof of Property 3.9. We partition the vertex set $V = V_1 \dot\cup V_2$, where

$$V_1 \coloneqq \{v \in V \colon \deg(v) \text{ odd}\} = \{\text{odd-degree vertices}\} \ , \text{ and}$$
$$V_2 \coloneqq \{v \in V \colon \deg(v) \text{ even}\} = \{\text{even-degree vertices}\} \ .$$

We now have

$$\sum_{v \in V_1} \deg(v) = \sum_{v \in V} \deg(v) - \sum_{v \in V_2} \deg(v) \ .$$

On the right-hand side of the last equation, the first sum is even by Property 3.8, and the second one is even as well because each summand is even. Thus it follows that the sum on the left-hand side is even. Notice that each term in the sum on the left-hand side is odd by definition of V_1; hence, the number of summands must be even, i.e., $|V_1|$ is even. □

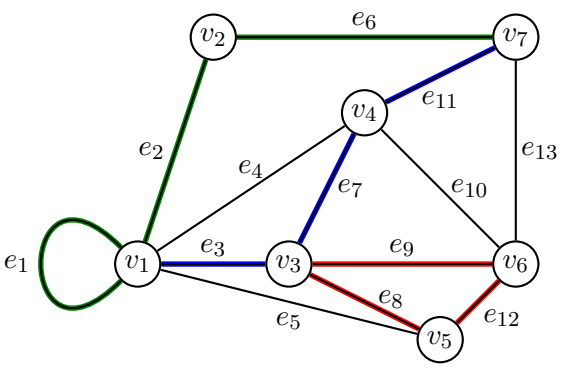
The reason why Property 3.9 is also known as the handshaking lemma is the following: Property 3.9 states that at every party the number of people who shake hands with an odd number of guests has to be even. To see this, consider a graph that has the party's guests as its vertices. Two people are connected by an edge if and only if they shake hands. Thus, the degree of a vertex is the number of guests with whom the corresponding person shook hands. The statement now follows immediately from Property 3.9.

A *walk* in an undirected graph $G = (V, E)$ is a sequence $v_1, e_1, v_2, \dots, v_{k-1}, e_{k-1}, v_k$ (possibly with repetitions) consisting of $k \in \mathbb{Z}_{\geq 1}$ vertices $v_1, \dots, v_k \in V$ and $k - 1$ edges $e_1, \dots, e_{k-1} \in E$ with $e_i = \{v_i, v_{i+1}\}$ for all $i \in [k-1]$, where we recall that $[\ell] \coloneqq \{1, \dots, \ell\}$ for $\ell \in \mathbb{Z}_{\geq 1}$, and we use the convention $[0] = \emptyset$. We call such a walk a *walk between* v_1 *and* v_k.

For an undirected walk, we consider the description $v_1, e_1, v_2, \dots, v_{k-1}, e_{k-1}, v_k$ and the description $v_k, e_{k-1}, v_{k-1}, \dots, v_2, e_1, v_1$ in the opposite direction as equivalent, i.e., they describe the same walk.

In a directed graph $G = (V, A)$, a walk is a sequence $v_1, a_1, v_2, \dots, v_{k-1}, a_{k-1}, v_k$ (possibly with repetitions) consisting of vertices $v_1, \dots, v_k \in V$ and arcs $a_1, \dots, a_{k-1} \in A$ with $a_i = (v_i, v_{i+1}) \in A$ for all $i \in [k-1]$. In the directed case we are talking about a *walk from* v_1 *to* v_k or a v_1-v_k *walk*. This notation is also used for undirected walks, where in undirected graphs, each v_1-v_k walk is also a v_k-v_1 walk.

In both cases we define the *length* of a given walk with k vertices—where a vertex is counted as many times as it appears in the walk—as $k-1$, which is the number of edges in the walk, also counted with multiplicities if an edge appears multiple times. A walk is called a *path* if it contains no vertex more than once, i.e., $v_i \neq v_j$ for all $i, j \in [k]$ with $i \neq j$. A walk is called *closed*, if $v_1 = v_k$. A closed walk, in which all vertices except for start and endpoint are pairwise distinct, is called a *cycle*. Figure 3.15 illustrates the new terminology.



$v_1, e_1, v_1, e_2, v_2, e_6, v_7, e_6, v_2$ is a walk of length 4, but not a path, since v_1 and v_2 appear multiple times.

$v_1, e_3, v_3, e_7, v_4, e_{11}, v_7$ is a v_1-v_7-path of length 3.

$v_3, e_8, v_5, e_{12}, v_6, e_9, v_3$ is a closed walk with pairwise distinct vertices (up to start and endpoint), hence it is a cycle of length 3.

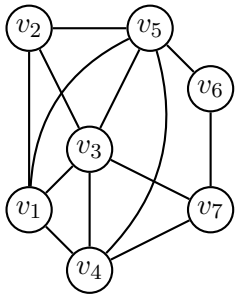
Figure 3.15: Walks, paths, closed walks, cycles, and their lengths.

To simplify the notation, we often denote a path or cycle simply by the set of edges it traverses, for example, a cycle C in a graph $G = (V, E)$ is represented as an edge set $C \subseteq E$. In addition, '$+$' and '$-$' are used for adding or deleting a single element to or from a set, i.e., $S + u - w := (S \cup \{u\}) \setminus \{w\}$.

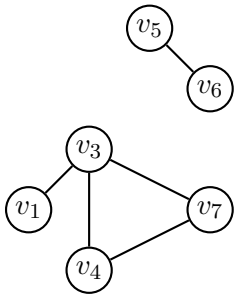
A *subgraph* of an undirected graph $G = (V, E)$ is a graph $H = (W, F)$ with $W \subseteq V$ and $F \subseteq \{e \in E \colon e \subseteq W\}$, i.e., F is a subset of those edges of G that have both endpoints in W. For a vertex set $W \subseteq V$, the subgraph *induced* by W is the subgraph of $G = (V, E)$ defined by

$$G[W] = (W, E[W]) \ ,$$

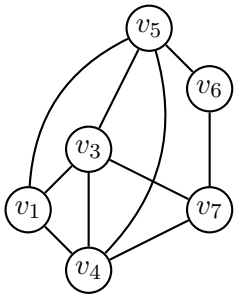
where $E[W] := \{e \in E \colon e \subseteq W\}$ is the set of all edges of G with both endpoints in W. The definitions of subgraph and induced subgraph are analogous for directed graphs. Figure 3.16 exemplifies these notions for undirected graphs.



(a) A graph $G = (V, E)$.



(b) A subgraph of G.



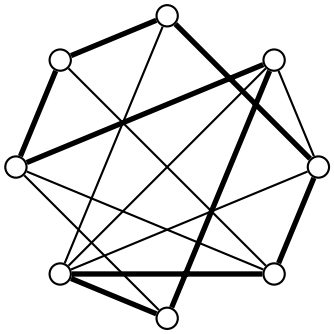
(c) Induced subgraph $G[V \setminus \{v_2\}]$.

Figure 3.16: Graphs, subgraphs, and induced subgraphs.

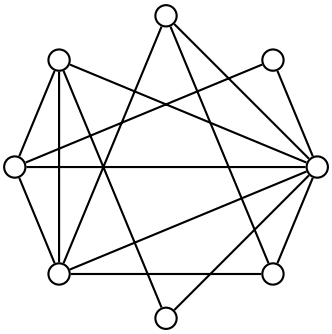
Exercise 3.10: Seating arrangements

You invite n guests over for dinner. For each pair of guests it is known whether they are friends or not. The task is to create a seating plan for a round table such that guests sitting next to each other are friends. Formulate this problem as the graph-theoretic problem of finding a certain cycle in a well-defined graph.

Solution
We construct a graph $G = (V, E)$ in which each vertex corresponds to a guest. Two vertices are connected by an edge if and only if the two corresponding guests are friends. A desired seating plan now corresponds to a cycle in G that contains all the vertices in V exactly once (see Figure 3.17a). Such a cycle is called a *Hamilton cycle*. Of course, not every graph contains a Hamilton cycle. For example, for the graph in Figure 3.17b, it can be shown that it does not contain a Hamilton cycle.



(a) A graph with a Hamilton cycle.

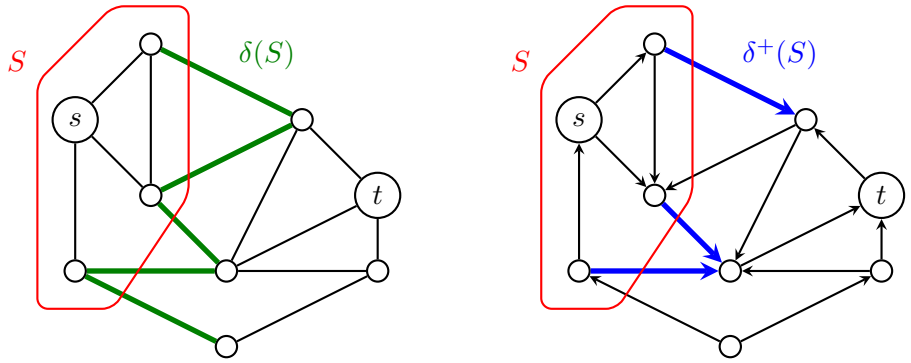


(b) A graph with no Hamilton cycle.

Figure 3.17: Hamilton cycles in graphs.

A *cut* in a graph $G = (V, E)$ is a nonempty subset $S \subsetneq V$ of vertices, where $S \subsetneq V$ is a shorthand notation for $S \subseteq V$ and $S \neq V$. One can also imagine a cut as a non-trivial partition of the vertex set V into the sets S and $V \setminus S$. This partition is non-trivial because both S and

$V \setminus S$ are nonempty. For this reason, a cut is sometimes also written as the pair $(S, V \setminus S)$. Cuts are defined analogously in directed and undirected graphs. For a cut S in an undirected graph, the edges in $\delta(S)$ are also referred to as the *edges in the cut* S or as the edges *crossing* S. In the case of a directed graph, we denote the edges in $\delta^+(S)$ as the *edges in the cut* S. For two distinct vertices $s, t \in V$, an *s-t cut* in G is a cut $S \subseteq V$ such that $s \in S$ and $t \notin S$. In other words, an s-t cut separates s from t. Figure 3.18 illustrates the difference between the undirected and the directed case.



(a) Undirected graph. (b) Directed graph.

Figure 3.18: An s-t cut S and the edges in the cut $\delta(S)$ (for the undirected case) and $\delta^+(S)$ (for the directed case).

Exercise 3.11

Let $G = (V, A)$ be a directed graph, S a cut in G, and $C \subseteq A$ a closed directed walk in G. Show that $|\delta^+(S) \cap C| = |\delta^-(S) \cap C|$.

Solution

Let $H = (V, C)$ be the subgraph of G only containing the arcs in C. For each vertex set $W \subseteq V$, let $\delta_H^+(W) := \delta^+(W) \cap C$ and $\delta_H^-(W) := \delta^-(W) \cap C$. Using this notation, we therefore need to prove $|\delta_H^+(S)| = |\delta_H^-(S)|$. First we show

$$|\delta_H^+(S)| - |\delta_H^-(S)| = \sum_{v \in S} \left(\deg_H^+(v) - \deg_H^-(v)\right) \quad . \tag{3.1}$$

This equality can be verified by checking that each arc $a \in C$ contributes to both sides of the equation equally. An easy way to perform this check is by distinguishing the following three types of arcs: (i) $a \in \delta_H^+(S)$, (ii) $a \in \delta_H^-(S)$, and (iii) $a \notin \delta_H^+(S) \cup \delta_H^-(S)$.
Because C is a closed directed walk, every vertex in H has the same indegree as outdegree, i.e., $\deg_H^+(v) = \deg_H^-(v)$ for all $v \in V$. Consequently, every single summand on the right-hand side of (3.1) is equal to zero. Hence, the difference on the left-hand side is zero, too. It follows that $|\delta_H^+(S)| = |\delta_H^-(S)|$, as desired.

3.3 Data structures for graphs

One of our main goals is to develop and analyze algorithms for graph optimization problems. To this end, we need to clarify how a graph is stored in a computer and how graph-related data can be accessed. This will both clarify the input size of a graph and the time we need to perform basic graph operations, both of which are key to talk about the running time of graph algorithms. In this section we address these points by talking about data structures for graphs.

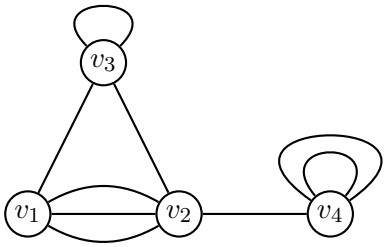
Two of the most widely used data structures for graphs are the representation as an *adjacency matrix* (sometimes called *neighborhood matrix*) and the representation as an *incidence list*. Unless explicitly indicated otherwise, we will always assume that the underlying data structure is an incidence list. However, we start by presenting the adjacency matrix before expanding on the incidence list, due to its simplicity. The choice of data structures is a crucial and often highly non-trivial step in algorithmics that comes with various trade-offs. Introducing another graph data structure besides the incidence list allows us to discuss and compare advantages and disadvantages of these two data structures in comparison to each other, and to highlight some example trade-offs faced when choosing a graph data structure.

The adjacency matrix

The *adjacency matrix* of an undirected graph $G = (V, E)$ is the symmetric matrix $M \in \mathbb{Z}_{\geq 0}^{V \times V}$ whose entries for all $u, v \in V$ are defined by

$$M(u, v) \coloneqq \begin{cases} |\{e \in E \colon \text{endpoints}(e) = \{u, v\}\}| & \text{if } u \neq v \ , \\ 2 \cdot |\{\text{loops at } v\}| & \text{if } u = v \ . \end{cases}$$

Figure 3.19 shows the adjacency matrix of an undirected graph with 4 vertices.



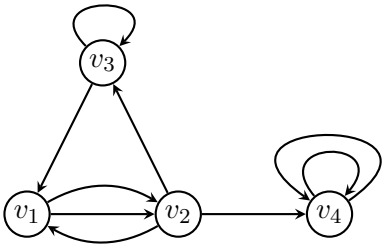
$$M = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 1 & 1 \\ 1 & 1 & 2 & 0 \\ 0 & 1 & 0 & 4 \end{pmatrix} \end{array}$$

Figure 3.19: The adjacency matrix of an undirected graph.

The adjacency matrix of a directed graph $G = (V, A)$ is the matrix $M \in \mathbb{Z}_{\geq 0}^{V \times V}$ whose entries for all $u, v \in V$ are given by

$$M(u, v) \coloneqq |\{a \in A \colon \text{tail}(a) = u, \text{head}(a) = v\}| \ .$$

Figure 3.20 shows the adjacency matrix of a directed graph with 4 vertices. As the example in Figure 3.20 highlights, the adjacency matrix of a directed graph is not necessarily symmetric.



$$M = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 2 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \end{array}$$

Figure 3.20: The adjacency matrix of a directed graph.

For simple graphs, the corresponding adjacency matrices are $\{0,1\}$-matrices, i.e., matrices where each entry is either 0 or 1. In addition, for simple graphs, all entries on the diagonal of the adjacency matrix are 0.

The incidence list

An *incidence list* contains for each vertex a doubly linked list of those (directed or undirected) edges incident to the corresponding vertex. To access these lists, we can use an array indexed by the vertices of the graph, which contains the pointers to the respective lists. More precisely, these pointers point to the first element of the respective lists. For each vertex v, accessing the first element of the list corresponding to v requires constant time. Figure 3.21 schematically shows the incidence list of a directed graph.

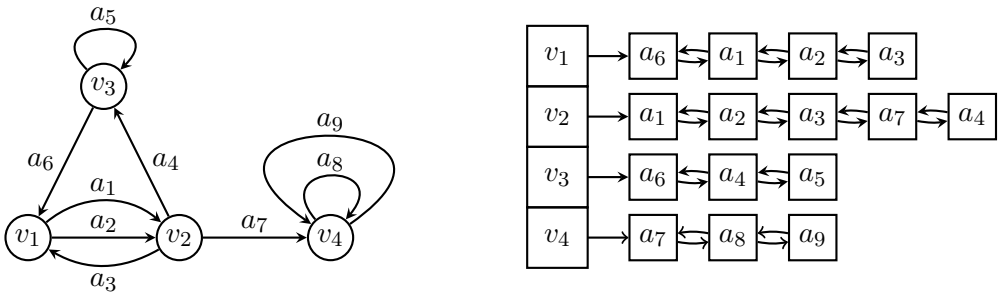


Figure 3.21: The incidence list of a directed graph.

For undirected edges e we use a data structure in which the set $\mathrm{endpoints}(e)$ is stored and can be queried. In the case of an arc a, it is possible to query $\mathrm{tail}(a)$ and $\mathrm{head}(a)$ in constant time. Further information about the edges, for example weights, can also be stored in the data structure for the edges.

Comparison: adjacency matrix and incidence list

Space requirements We consider an undirected graph $G = (V, E)$ (the directed case is analogous). Let $n := |V|$ be the number of vertices and $m := |E|$ the number of edges of G.

An important difference between the adjacency matrix and the incidence list of G is the space required:

- Adjacency matrix: $\Theta(n^2)$.
- Incidence list: $\Theta(m+n)$.

Thus, for simple graphs the incidence list is typically a more compact encoding of the graph, because in this case $m = O(n^2)$. The reason for the significantly larger space requirement of the adjacency matrix is that even if two vertices are not connected by an edge, it stores a 0 to capture this information. This difference is particularly pronounced if the graph is *sparse*, that is, if it has only few edges, for example $m = O(n)$. Here, the adjacency list needs quadratic space in n whereas the incidence list only requires linear space. This difference is crucial when dealing with graphs with millions of vertices but constant average degree—a regime that is very common in many applications.

Algorithmic complexity of some operations the Table 3.1 lists, for both the adjacency matrix and the incidence list, the running times of several simple computational tasks in an undirected graph $G = (V, E)$, namely: calculating the degree of a vertex or listing all incident edges, deciding whether a particular edge is part of the graph, and computing the number of edges $|E|$.

	$\deg(v)$ or $\delta(v)$	$\exists ?\{u, v\} \in E$	$\lvert E\rvert$
adjacency matrix	$O(n)$	$O(1)$	$O(n^2)$
incidence list	$O(\deg(v))$	$O(\min\{\deg(u), \deg(v)\})$	$O(m+n)$

Table 3.1: Running time of simple computations in graphs.

Exercise 3.12

Let $G = (V, E)$ be a graph given by an incidence list and let $u, v \in V$ two vertices. Show that in time $O(\min\{\deg(u), \deg(v)\})$ it can be decided whether G contains an edge $\{u, v\}$.

We recall that, unless explicitly stated otherwise, we will always assume that graphs are given as incidence lists. The size of a graph with n vertices and m edges is therefore $\Theta(m+n)$. Thus, an algorithm whose input is a single such graph is polynomial if and only if its running time is upper bounded by a polynomial in $m + n$.

3.4 Breadth-first search (BFS): shortest paths and more

We now discuss one of the best-known search methods on graphs, namely breadth-first search, or BFS for short. To be exact, breadth-first search is an algorithmic idea that can be adapted to solve many different problems. We introduce it in the context of finding shortest paths in undirected graphs. Later we will see how the algorithm can be modified for directed graphs.

Let $G = (V, E)$ be an undirected graph. We define a distance function $d\colon V \times V \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ as follows:

$$d(u, v) \coloneqq \min\{|P|\colon P \subseteq E,\ P \text{ is a walk between } u \text{ and } v\} \qquad \forall u, v \in V\ , \tag{3.2}$$

i.e., $d(u, v)$ is the length of the shortest u-v-walk in G. If there is no walk between u and v, then $d(u, v)$, as defined above, is the minimum over an empty set. For this case we set by convention $d(u, v) = \infty$. Notice that one can impose in (3.2) the additional requirement that P must be a path (instead of a walk) without changing the definition. Indeed, this would not change the value of $d(u, v)$ because a shortest walk is always a path. Figure 3.22 shows a graph $G = (V, E)$ that contains next to each vertex v the distance to the fixed vertex v_1, i.e., the length of a shortest path to v_1.

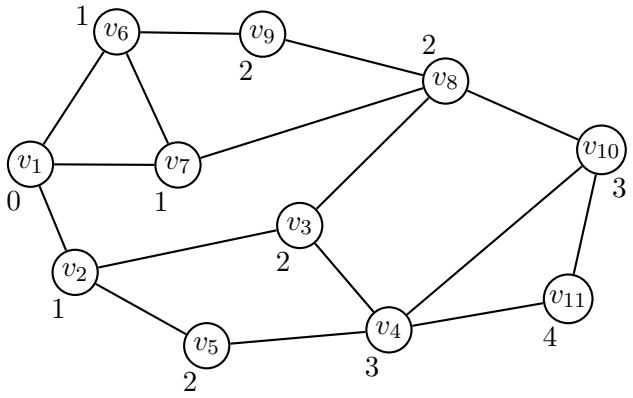


Figure 3.22: The number next to each vertex gives the distance to v_1.

Breadth-first search is an algorithm that efficiently calculates the distances $d(v_1, v)$ for all vertices $v \in V$ for a fixed vertex $v_1 \in V$. Algorithm 2 describes BFS in pseudocode.

The idea of breadth-first search is simple: We first consider the vertex v_1. For this vertex, we know that $d(v_1, v_1) = 0$, because an empty walk has no edges and therefore has length 0. In the following, we consider all vertices in the *neighborhood* $N(\{v_1\})$ of v_1, that is, all vertices connected to v_1 by an edge. For these vertices v we set $d(v_1, v) = 1$. In the next step we consider all vertices that are adjacent to one of the vertices of distance 1 and have not yet been considered. For these vertices v we set $d(v_1, v) = 2$, and so on. Table 3.2 shows the sets L and $N(L)$ for each iteration of the while-loop Algorithm 2 runs through for the graph in Figure 3.22. The set L contains the vertices newly considered in the previous iteration, $N(L)$ their neighbors.

In the following we show that the function d computed by breadth-first search indeed corresponds to the distances from v_1. In addition, we show that the runtime of BFS is $O(m + n)$, where $n \coloneqq |V|$ and $m \coloneqq |E|$, which is linear in the size of the graph.

Algorithm 2: Breadth-first search: computation of distances from a fixed vertex v_1

Input: $G = (V, E)$, $v_1 \in V$.
Output: $d(v_1, v)$ for all $v \in V$.

1. **Initialization:**
   $d(v_1, v) = \begin{cases} \infty & \text{if } v \in V \setminus \{v_1\}, \\ 0 & \text{if } v = v_1. \end{cases}$
   $L = \{v_1\}$. `// vertices to be processed`
   $k = 1$. `// shortest possible assignable distance`
2. **while** $(L \neq \emptyset)$ **do:**
   $L_{\text{new}} = \emptyset$.
   for $v \in N(L) \coloneqq \{u \in V : \exists w \in L \text{ with } \{w, u\} \in E\}$ **do:**
   if $d(v_1, v) = \infty$ **then:**
   $d(v_1, v) = k$.
   $L_{\text{new}} = L_{\text{new}} \cup \{v\}$.
   $k = k + 1$.
   $L = L_{\text{new}}$.
3. **return** $d(v_1, v)$ for all $v \in V$.

k	L	$N(L)$
1	$\{v_1\}$	$\{v_2, v_6, v_7\}$
2	$\{v_2, v_6, v_7\}$	$\{v_1, v_3, v_5, v_6, v_7, v_8, v_9\}$
3	$\{v_3, v_5, v_8, v_9\}$	$\{v_2, v_3, v_4, v_6, v_7, v_8, v_9, v_{10}\}$
4	$\{v_4, v_{10}\}$	$\{v_3, v_4, v_5, v_8, v_{10}, v_{11}\}$
5	$\{v_{11}\}$	$\{v_4, v_{10}\}$
6	$\{\}$	$\{\}$

Table 3.2: Breadth-first search table for the graph in Figure 3.22.

Theorem 3.13: Correctness of breadth-first search

For every graph $G = (V, E)$ and every vertex $v_1 \in V$, Algorithm 2 calculates the values $d(v_1, v)$ correctly for all $v \in V$.

Proof. To avoid confusion, let $d'(v_1, v)$ for $v \in V$ be the distances computed by Algorithm 2, and d the true distances in G as defined in (3.2). Furthermore, let $D'_\ell = \{v \in V : d'(v_1, v) = \ell\}$ and $D_\ell = \{v \in V : d(v_1, v) = \ell\}$ for all $\ell \in \mathbb{Z}_{\geq 0}$.

We show for every $k \in \mathbb{Z}_{\geq 0}$ that Algorithm 2, after k iterations of the while-loop, computed the set of all vertices $v \in V$ satisfying $d(v_1, v) \leq k$ as well as their distances correctly, i.e., $D_\ell = D'_\ell$ for all $\ell \leq k$. For this we use induction on k. For $k = 0$ the statement is certainly true, because only $v = v_1$ satisfies $d(v_1, v) = 0$ and only $v = v_1$ satisfies $d'(v_1, v) = 0$, So,

suppose the statement holds for iteration k and consider iteration $k+1$ of the while-loop. In this iteration, we set $d'(v_1, v) = k+1$ for all v that are adjacent to at least one vertex in $D'_k = D_k$ and have not yet been considered, hence $d'(v_1, v) = \infty$. Consequently,

$$D'_{k+1} = \{v \in V : d(v_1, v) > k, v \in N(D_k)\} \ .$$

We need to show that $D'_{k+1} = D_{k+1}$. For this we prove both inclusions $\subseteq$ and $\supseteq$.

$D'_{k+1} \subseteq D_{k+1}$: Let $v \in D'_{k+1}$. Because $v \in N(D_k)$, it holds that $d(v_1, v) \leq k+1$: At least one neighbor w of v has distance k from v_1, thus there exists a v_1-w walk of length at most k. This walk can be extended to a v_1-v walk of length at most $k+1$, which yields $d(v_1, v) \leq k+1$. In addition, we have $d(v_1, v) > k$ because, due to the inductive assumption, all vertices v with $d(v_1, v) \leq k$ have already been considered. Together it follows that $d(v_1, v) = k+1$, so $v \in D_{k+1}$.

$D_{k+1} \subseteq D'_{k+1}$: Let $v \in D_{k+1}$. Hence, there is a v_1-v walk of length $k+1$ with vertices $v_1, \ldots, v_{k+1}, v$. It follows that $v_{k+1} \in D_k$ needs to hold; for otherwise we could construct a v_1-v-path of length $< k+1$. Thus, $v \in N(D_k)$, and consequently $v \in D'_{k+1}$.

This completes the inductive step and thus implies the statement. □

Theorem 3.14: Running time of breadth-first search

For every graph $G = (V, E)$ and every vertex $v_1 \in V$, Algorithm 2 has a running time bounded by $O(m+n)$.

Proof. The initialization in the first step of Algorithm 2 requires $O(n)$ time. For the second step, we first determine the time needed to compute the neighborhoods $N(L)$ calculated at the beginning of the for-loop. Let $L_1 = \{v_1\}, L_2, \ldots, L_{k-1}, L_k = \emptyset$ be the sets L occurring during the algorithm. Then L_i contains exactly those vertices of V that have distance $i-1$ from v_1. Thus, $L_1, \ldots, L_k$ is a partition of all vertices that are in the same connected component as v_1.

The calculation of $N(L)$ takes $O\left(|L| + \sum_{v \in L} \deg(v)\right)$ time, because all neighbors of vertices in L are explored. Thus, the running time to determine all neighborhoods $N(L_1), \ldots, N(L_k)$ is bounded by

$$O\left(\sum_{i=1}^{k} \left(|L_i| + \sum_{v \in L_i} \deg(v)\right)\right) = O\left(|V| + \sum_{v \in V} \deg(v)\right) = O(n+m) \ ,$$

where we use that $L_1, \ldots, L_k$ is a partition of V, and that the sum of all vertex degrees is exactly $2m$ (see proof of Property 3.8). Furthermore, each vertex $v \in V$ is contained in at most 3 different neighborhoods $N(L_j)$, namely in the iteration i such that $v \in L_i$, as well as in the one before and after. Indeed, a vertex can only be a neighbor of vertices with either the same distance to v_1 or with a distance to v_1 differing from $d(v_1, v)$ by exactly one unit. Thus, $\sum_{i=1}^{k} |N(L_i)| = O(n)$, and because every operation within the for-loop only requires constant time per iteration, the total running time of all operations within the for-loop is bounded by $O(n)$. The output of the distances $d(v_1, v)$ in the last step requires $O(n)$ time. Overall, the running time is therefore bounded by $O(m+n)$, as desired. □

Algorithm 2 can easily be adapted for directed graphs $G = (V, A)$. For this we define the *out-neighborhood* $N^+(L)$ as

$$N^+(L) = \{u \in V : \exists w \in L \text{ with } (w, u) \in A\} \ .$$

It suffices to replace $N(L)$ by $N^+(L)$ in Algorithm 2 to adapt it to the directed case.

Connectivity and connected components

A canonical application of BFS is to study the connectivity of a graph.

Definition 3.15: Connectivity in undirected graphs

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

(i) Two vertices $s, t \in V$ are called *connected* in G if G contains an s-t walk.
(ii) The graph G is callled *connected* if each pair of vertices in G is connected.

With breadth-first search it is easy to check if a graph $G = (V, E)$ is connected. It suffices to perform a single breadth-first search with an arbitrary starting vertex $v_1 \in V$, because G is connected if and only if $d(v_1, v) < \infty$ holds for all $v \in V$.

In directed graphs $G = (V, A)$, connectivity can be defined analogously by "forgetting" the directions of the arcs. Moreover, there is a stronger notion of connectivity for directed graphs.

Definition 3.16: Connectivity in directed graphs

Let $G = (V, A)$ be a directed graph.

(i) G is called *connected*, if the undirected graph G', obtained from G by ignoring arc directions, is connected.
(ii) Let $s, t \in V$. The vertex t can be *reached* from s in G if G contains a directed s-t walk.
(iii) The graph G is called *strongly connected* if every vertex in G can be reached from every other vertex.

Also strong connectivity (of a directed graph) can be checked through BFS, using only two BFS calls. To this end, we fix an arbitrary vertex v_1 and start BFS with v_1 as starting vertex. This run allows us to determine whether every vertex can be reached from v_1, that is, if $d(v_1, v) < \infty \ \forall v \in V$. If a vertex cannot be reached from v_1, then G is not strongly connected. Otherwise, we call BFS a second time with starting vertex v_1 but this time on the graph $\overline{G}$ obtained from G by reversing all arc directions. This allows us to determine whether in G, the vertex v_1 can be reached from all other vertices. If not, then G is not strongly connected. Otherwise, G is strongly connected. Indeed, if there is a vertex that can reach any other vertex and can be reached by any other vertex then the graph must be strongly connected. This follows from the fact that for any two vertices $u, w \in V$ there is a u-w walk because such a walk can be obtained by concatenating a u-v_1 walk with a v_1-w walk.

Definition 3.17: Connected components in undirected graphs

Let $G = (V, E)$ be an undirected graph. A *connected component* of G is an induced subgraph $G[W]$ such that $G[W]$ is connected and $W \subseteq V$ is maximal with respect to this property, i.e., for every $X \subseteq V$ with $X \supsetneq W$ the graph $G[X]$ is not connected.

We emphasize the crucial notion of maximality, which is ubiquitous in Mathematical Optimization. Namely, a set W is *maximal* with respect to a particular property if there is no strict superset of W that also fulfills the property. This does not necessarily mean that W is a set of maximum cardinality among all the sets with the property. The converse, however, is true: If a set among all sets with a given property has maximum cardinality, then it is also maximal with respect to that property.

Each graph can be decomposed into its connected components. Figure 3.23 shows a graph with four connected components.

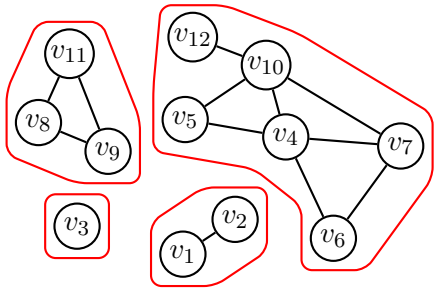


Figure 3.23: The above graph G has four connected components: $G[\{v_1, v_2\}]$, $G[\{v_3\}]$, $G[\{v_4, v_5, v_6, v_7, v_{10}, v_{12}\}]$, and $G[\{v_8, v_9, v_{11}\}]$.

Using BFS, one can also find the connected components of a graph. Starting from a starting vertex v_1, breadth-first search reaches exactly those vertices that are in the same connected component as v_1. Indeed, the vertices in the same connected component are precisely those that can be reached from v_1, i.e., those vertices v satisfying $d(v_1, v) < \infty$. So, if V_1 is the set of vertices reached by the breadth-first search, then $G[V_1]$ is a connected component of G. We can repeat this procedure for another starting vertex $v_2 \in V \setminus V_1$ to find a second connected component $G[V_2]$, and so on. A crude way to bound the running time of this procedure is by observing that we make $O(n)$ calls to BFS—because a graph has at most $n := |V|$ connected components—each of which uses $O(n + m)$ time by Theorem 3.14. This leads to an overall running time bound of $O(n(m + n))$.

Exercise 3.18: Running time improvement

Show that the connected components of an undirected graph $G = (V, E)$ can be found in running time $O(m + n)$. To achieve this, improve the running time analysis presented above and, if necessary, adapt BFS (Algorithm 2) for this setting.

We can analogously define connected components in directed graphs by disregarding arc directions. Again, for directed graphs, there is moreover a natural stronger notion of connected components.

Definition 3.19: Connected components in directed graphs

Let $G = (V, A)$ be a directed graph.

(i) The *connected components* of G are the connected components of the undirected graph G', which results from G by ignoring arc directions.

(ii) A *strongly connected component* of G is an induced directed subgraph $G[W]$ such that $G[W]$ is strongly connected and $W \subseteq V$ is maximal with respect to this property.

Figure 3.24 shows a directed graph with five strongly connected components.

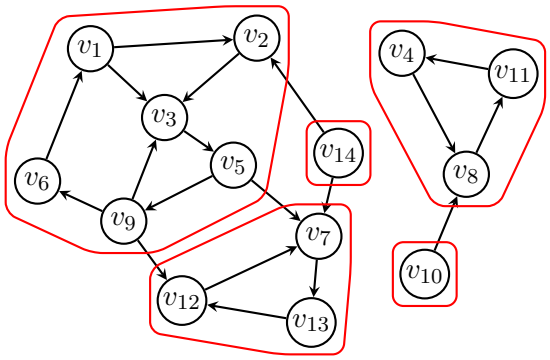


Figure 3.24: A graph G with two connected components and five strongly connected components (in red): $G[\{v_{10}\}]$, $G[\{v_1, v_2, v_3, v_5, v_6, v_9\}]$, $G[\{v_4, v_8, v_{11}\}]$, $G[\{v_7, v_{12}, v_{13}\}]$, and $G[\{v_{14}\}]$.