# MTH6105 Algorithmic Graph Theory
# (Spring 2024)

Felix Fischer
felix.fischer@qmul.ac.uk

May 7, 2024

# Contents

1 Introduction and Basic Definitions 1
1.1 Graphs . . . 2
1.2 Representations of Graphs . . . 3
1.3 Graph Isomorphism . . . 4
1.4 Subgraphs . . . 6
1.5 Neighborhoods and Degrees . . . 7
1.6 Digraphs and Networks . . . 8

2 Paths, Cycles, and Trees 11
2.1 Connectivity . . . 11
2.2 Trees . . . 15
2.3 Characterizations of Trees . . . 16
2.4 Counting Trees . . . 17
2.5 Spanning Trees . . . 23

3 Complexity of Algorithms and Problems 25
3.1 Input and Running Time . . . 26
3.2 Asymptotic Upper Bounds . . . 27
3.3 Complexity of Problems: P and NP . . . 30

4 Graph Traversal 33
4.1 Breadth-First and Depth-First Search . . . 33
4.2 Connected Components . . . 36
4.3 Paths and Cycles . . . 37
4.4 Strongly Connected Components . . . 38
4.5 Directed Cycles . . . 39

5 Minimum Spanning Trees and Shortest Paths in Networks 43
5.1 Minimum Spanning Trees . . . 43
5.2 Shortest Paths for Non-Negative Weights . . . 48
5.3 Shortest Directed Paths . . . 50
5.4 Directed Cycles . . . 53
5.5 Longest Paths in Directed Acyclic Networks . . . 54

6 Network Flows 57
6.1 Maximum Flows . . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.2 Minimum Cuts . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.3 Residual Capacities and Augmenting Paths . . . . . . . . . . . . . 60
6.4 The Ford-Fulkerson Algorithm . . . . . . . . . . . . . . . . . . . . 63

7 Matchings 65
7.1 Bipartite Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . 66
7.2 Maximimum Matchings in Bipartite Graphs . . . . . . . . . . . . . 67
7.3 Augmenting Paths . . . . . . . . . . . . . . . . . . . . . . . . . . 69
7.4 Saturating Matchings in Bipartite Graphs . . . . . . . . . . . . . . 72

8 Euler Trails and Tours 75

# Chapter 1

# Introduction and Basic Definitions

This module provides an introduction to the theory of graphs, with a focus on algorithms that work with graphs. A *graph* is a mathematical model for any situation involving a set of entities that may or may not be in a pairwise relation with each other. Examples of such situations include the following:

- a road network, where the entities are cities, and two cities are in relation if there is a direct road between them;
- the World Wide Web, where entities are web pages, and two web pages are in relation if one links to the other;
- a network of acquaintances, where the entities are individuals, and two individuals are in relation if they know each other.

Graph theory is a relatively young area of mathematics that has developed very rapidly. Its origins can be traced back to the work of Leonhard Euler on the problem of the Seven Bridges of Königsberg, but a comprehensive theory has only developed in the last hundred years. The first book on graph theory, *Theorie der Endlichen und Unendlichen Graphen* by Dénes König, was published in 1936. Since the 1940s, a major stimulus for the development of graph theory has been the search for efficient algorithms for network optimization in operations research.

An *algorithm* is a set of rules or instructions followed to solve a problem. We will prove results about graphs and develop algorithms for solving problems modeled by a graph. The two will go hand in hand: sometimes a theorem concerning graphs will help us in developing an algorithm and show that it works correctly and efficiently, sometimes an algorithm will help us in proving a theorem. An appealing aspect of graph theory is that it lends itself to visualization, as abstract ideas can be illustrated by drawing the entities and their relationships as dots and lines on a piece of paper.

## 1.1 Graphs

**Definition 1.1** (graph, vertex, edge, endpoints, incidence, adjacency, loop, multiple edge). A *graph* $G$ is given by a finite set $V$ of *vertices* and a finite set $E$ of *edges* such that $V \cap E = \emptyset$. Each edge in $E$ is associated with two vertices in $V$ called its *endpoints*. We say that an edge is *incident* to both its endpoints, or *between* its endpoints. We say that two vertices $u$ and $v$ are *adjacent* or *neighbors* if there is an edge between $u$ and $v$. An edge is a *loop* if both of its endpoints are the same, and a *multiple edge* if there is another edge with the same endpoints.

Given a graph $G$, we will write $V(G)$ for its set of vertices and $E(G)$ for its set of edges. Often we will use integers to represent vertices, such that for a graph $G$ with $n$ vertices $V(G) = [n]$.

We will focus mainly on so-called simple graphs, which do not have any loops or multiple edges.

**Definition 1.2** (simple graph). A graph $G$ is *simple* if (i) every edge in $E(G)$ has two distinct endpoints and (ii) for every pair of vertices $u, v \in V(G)$ there is at most one edge incident to both $u$ and $v$.

When discussing simple graphs, it is convenient to label each edge by its endpoints and identify the edge by its label. From now on, we will say "the edge $\{u, v\}$" or "the edge $uv$" to refer to an edge with endpoints $u$ and $v$. Note that in this terminology, the edge $uv$ and the edge $vu$ are in fact the same edge. Note further that by the two properties of simple graphs, the terminology will never cause confusion: every edge has a label with two vertices, and every label refers to at most one edge. In other words, there is a bijection between the set of possible edges of a simple graph with vertex set $V$ and the set of $\binom{|V|}{2}$ unordered pairs of vertices. This immediately implies our first two results.

**Theorem 1.3.** A simple graph with $n$ vertices has at most $\binom{n}{2}$ edges.

**Corollary 1.4.** For any finite set $V$, there are $2^{\binom{|V|}{2}}$ distinct simple graphs with vertex set $V$.

**Example 1.5** (special graphs). Some graphs show up again and again and have been given a name. These include the following:

- $E_n$, the *empty graph* on $n$ vertices, with $V(E_n) = [n]$ and $E(E_n) = \emptyset$;

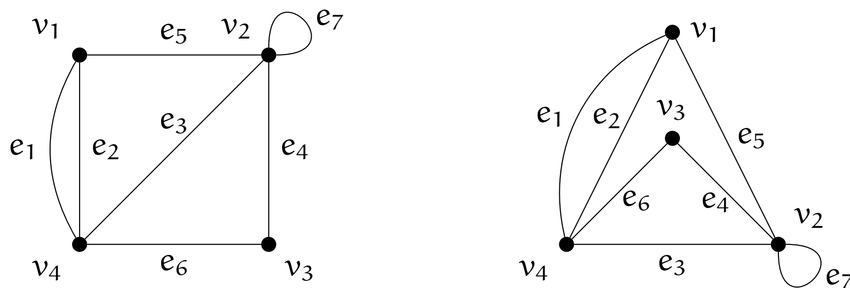



Figure 1.1: Two different drawings of the graph $G$ with $V(G) = \{v_1, v_2, v_3, v_4\}$ and $E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ where $e_1 = v_1v_4$, $e_2 = v_1v_4$, $e_3 = v_2v_4$, $e_4 = v_2v_3$, $e_5 = v_1v_2$, $e_6 = v_3v_4$, and $e_7 = v_2v_2$.

- $K_n$, the *complete graph* on $n$ vertices, with $V(K_n) = [n]$ and $E(K_n) = \{uv : u, v \in [n]\}$;
- $P_n$, the *path* on $n$ vertices, with $V(P_n) = [n]$ and $E(P_n) = \{uv : u, v \in [n], |u - v| = 1\}$;
- $C_n$, the *cycle* on $n \geqslant 3$ vertices, with $V(C_n) = [n]$ and $E(C_n) = \{uv : u, v \in [n], |u - v| = 1\} \cup \{1n\}$.

## 1.2 Representations of Graphs

A convenient way to specify and discuss graphs is to draw them in the plane. For a graph $G$, we do this by drawing a dot for each vertex in $V(G)$ and then place a line or curve between the two dots corresponding to vertices $u$ and $v$ if and only if $uv \in E(G)$. Figure 1.1 shows two different drawings of the same graph with four vertices and seven edges. Note that to determine whether the two drawings show the same graph, only incidences between vertices and edges matter and not the exact positions of the vertices and edges.

For larger and more complicated graphs, and in cases where we want to manipulate graphs using a computer, a more useful representation is the incidence matrix. As the name suggests, this matrix specifies the incidence relation between vertices and edges.

**Definition 1.6** (incidence matrix). The *incidence matrix* of a graph $G$ is a matrix with $|V(G)|$ rows corresponding to the vertices of $G$ and $|E(G)|$ columns corresponding to the edges of $G$. The entry in row $v \in V(G)$ and column $e \in E(G)$ is equal to 2 if $e$ is a loop from $v$ to itself, equal to 1 if $v$ is one of two distinct endpoints of $e$, and equal to 0 otherwise.

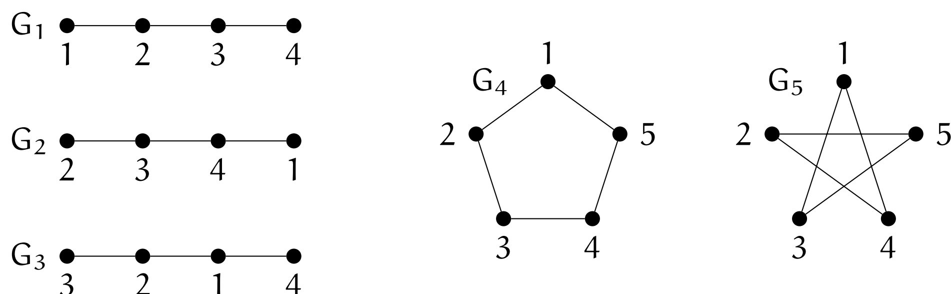



Figure 1.2: Five different graphs. The three graphs on the left are isomorphic to one another, and the two graphs on the right are isomorphic.

For example, the incidence matrix of the graph in Figure 1.1 is equal to

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

A more convenient representation for simple graphs is the so-called adjacency matrix.

**Definition 1.7** (adjacency matrix). The *adjacency matrix* of a simple graph $G$ is a matrix with $|V(G)|$ rows and columns corresponding to the vertices of $G$. The entry in row $u \in V(G)$ and column $v \in V(G)$ is equal to $1$ if $u$ is adjacent to $v$ in $G$, and equal to $0$ otherwise.

For example, the adjacency matrix of $P_4$ is equal to

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

Note that all diagonal entries are zero and the matrix is symmetric, corresponding to the absence of loops and the symmetry of the adjacency relation.

## 1.3 Graph Isomorphism

Two graphs are the same if and only if they have the same set of vertices and the same set of edges. As we have seen, the two drawings in Figure 1.1 depict the same graph.

Figure 1.2, on the other hand, shows five graphs that are all distinct. To see that this is true, first observe that the three graphs on the left all have four vertices, while both graphs on the right have five vertices. A graph on the left can thus not be the same as a graph on the right. Now consider the three graphs on the left. The edge $\{1, 4\}$ appears in $G_2$ and $G_3$ but not in $G_1$, while the edge $\{1, 2\}$ appears in $G_1$ and $G_3$ but not in $G_2$. Thus the three graphs on the left are all distinct. Finally the edge $\{1, 2\}$ appears in $G_4$ but not in $G_5$, so these graphs must be also be distinct.

You may be thinking at this point that the three graphs on the left look completely identical, apart from the fact that their vertices have been labeled differently. This leads us to the notion of isomorphic graphs.

**Definition 1.8** (graph isomorphism). Two simple graphs $G_1$ and $G_2$ are *isomorphic* if there exists a bijection $\phi : V(G_1) \to V(G_2)$ such that $uv \in E(G_1)$ if and only if $\phi(u)\phi(v) \in E(G_2)$.

Informally two graphs are isomorphic if we can obtain one from the other by relabeling the vertices, which is exactly what the function $\phi$ in Definition 1.8 is doing. A sensible way to extend this definition to graphs that are not simple would be to require that the bijection $\phi$ preserves the number of loops for each vertex and the number of edges between each pair of vertices.

To see that graphs $G_1$ and $G_2$ in Figure 1.2 are isomorphic, consider the bijection $\phi : V(G_1) \to V(G_2)$ with $\phi(1) = 2$, $\phi(2) = 3$, $\phi(3) = 4$, and $\phi(4) = 1$. It is straightforward to verify that for all $u, v \in V(G_1)$, $\phi(u)\phi(v) \in E(G_2)$ if and only if $uv \in E(G_1)$. For example, $12 \in E(G_1)$ but $14 \notin E(G_1)$, and $\phi(1)\phi(2) = 23 \in E(G_2)$ but $\phi(1)\phi(4) = 21 \notin E(G_2)$. Similarly, we can show isomorphism of $G_1$ and $G_3$, of $G_2$ and $G_3$, and of $G_4$ and $G_5$, by finding an appropriate bijection and verifying that it satisfies the conditions of Definition 1.8.

A necessary condition for two graphs to be isomorphic is that they have the same number of vertices and the same number of edges. However, deciding whether two graphs are isomorphic is not straightforward because we have to find an appropriate bijection or show that none exists.

An alternative way to think about graph isomorphism is by considering *unlabeled* graphs. Suppose you draw a graph but do not label any of the vertices. We can then think of this unlabeled graph as representing the *set* of graphs that can be obtained by labeling the vertices in some way. Every pair of graphs $G_1$ and $G_2$ in this set are isomorphic, since one can be obtained from the other by relabeling vertices: just switch the labeling used to construct $G_1$ from the unlabeled graph to the labeling used to construct $G_2$. We can in fact show that graph isomorphism is an equivalence relation, where each unlabeled graph corresponds to an equivalence class of this relation containing exactly the set of labeled graphs we have just described. As an equivalence relation, graph isomorphism in particular

satisfies transitivity. To show that the graphs $G_1$, $G_2$, and $G_3$ in Figure 1.2 are isomorphic to one another, it would thus suffice to show that $G_1$ is isomorphic to $G_2$ and that $G_2$ is isomorphic to $G_3$.

Some properties of graphs we will consider are invariant under isomorphism, and we will consider unlabeled graphs in these cases.

## 1.4 Subgraphs

Sometimes we want to talk about a smaller part of a given graph. To this end we introduce the notion of a *subgraph*.

**Definition 1.9** (subgraph). Let $G$ be a graph. Then a graph $H$ is a *subgraph* of $G$ if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

Note that this definition requires the subgraph $H$ to be a graph, so that the edges in $E(H)$ must be between vertices in $V(H)$.

To get some intuition for Definition 1.9 it is useful to think about the following two-step process for generating a subgraph of $G$:

1. Remove some vertices from $V$, together with all edges incident to these vertices.
2. Remove any edges from the remaining set of edges.

It is easy to see that this process produces a subgraph of the original graph, and that any subgraph can be obtained in this way. In particular, the way in which the first step is performed guarantees that all remaining edges are between vertices that were not removed.

By limiting ourselves to only the first step of the process, we obtain the more restrictive notion of an induced subgraph.

**Definition 1.10** (induced subgraph). Let $G$ be a graph and $X \subseteq V(G)$. The *induced subgraph* of $G$ on vertex set $X$, denoted $G[X]$, is the subgraph $H$ of $G$ where $V(H) = X$ and $E(H) \subseteq E(G)$ is the set of all edges in $E(G)$ between vertices of $X$.

In Figure 1.3, each of the four graphs at the bottom is a subgraph of $G$. Only the rightmost of them is also an induced subgraph, specifically the induced subgraph $G[\{1, 2, 3, 4\}]$. It is worth noting that every graph is both a subgraph and an induced subgraph of itself.

Given a graph $G$ and two subgraphs $G_1$ and $G_2$ of $G$, we will write $G_1 \cup G_2$ for the subgraph of $G$ with $V(G_1 \cup G_2) = V(G_1) \cup V(G_2)$ and $E(G_1 \cup G_2) = E(G_1) \cup E(G_2)$, and $G_1 \cap G_2$ for the subgraph of $G$ with $V(G_1 \cap G_2) = V(G_1) \cap V(G_2)$ and $E(G_1 \cap G_2) = E(G_1) \cap E(G_2)$. It is obvious from Definition 1.9 that $G_1 \cup G_2$ and $G_1 \cap G_2$ are indeed subgraphs of $G$.

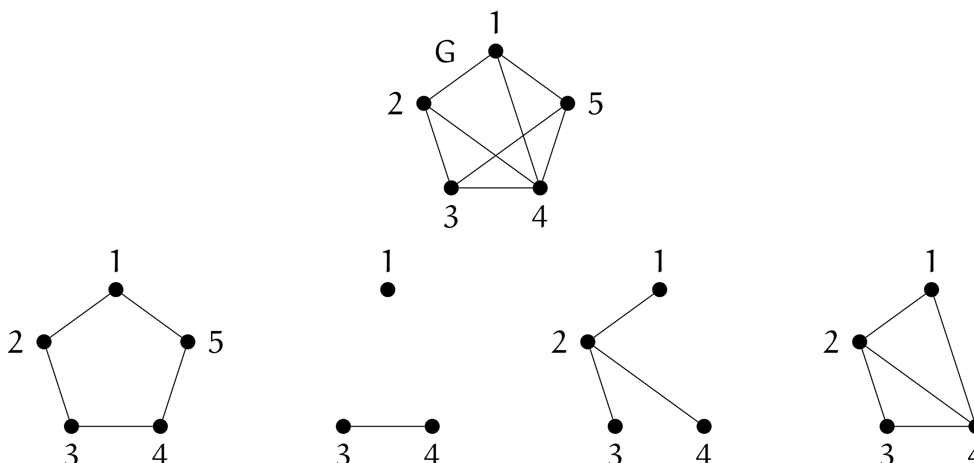



Figure 1.3: A graph G and four of its subgraphs. The right-most subgraph is an induced subgraph of G, whereas the other three are not.

## 1.5 Neighborhoods and Degrees

If we view a graph from the perspective of a particular vertex, two objects of obvious interest are the identity and the number of its neighbors.

**Definition 1.11** (neighborhood, degree). Let $G$ be a graph and $v \in V(G)$. Then the *neighborhood* $N_G(v)$ of $v$ in $G$ is the set of vertices adjacent to $v$. The *degree* $d_G(v)$ of $v$ in $G$ is the number of edges incident to $v$, where loops are counted twice.

When the graph $G$ is clear from the context, we will write $N(v)$ instead of $N_G(v)$ and $d(v)$ instead of $d_G(v)$. Note that in simple graphs, $d(v) = |N(v)|$.

Note that we can easily determine the degree of a vertex in a graph both from a drawing of the graph and from its incidence or adjacency matrix. Given a drawing we can simply count the edges incident to a particular vertex. Given an incidence or adjacency matrix, the degree of a vertex is equal to the sum of the entries in the row corresponding to that vertex.

The following simple result about the sum of degrees in a graph is also called the handshaking lemma, because it relates the number of handshakes at a dinner party to the number of hands each guest has shaken and implies that an even number of guests must have shaken an odd number of hands.

**Lemma 1.12.** For any graph $G$, $\sum_{v\in V(G)} d(v) = 2|E(G)|$.

*Proof.* Consider an edge $e \in E(G)$ with endpoints $u, v \in V(G)$. If $e$ is a loop, it contributes two to $\sum_{v\in V(G)} d(v)$. Otherwise it contributes one each to $d(u)$ and $d(v)$, and thus again two to $\sum_{v\in V(G)} d(v)$. This is true for every edge, so

$\sum_{v \in V(G)} d(v) = 2|E(G)|$. □

**Corollary 1.13.** In any graph, the number of vertices of odd degree is even.

## 1.6 Digraphs and Networks

In some situations it may not be enough to say that two vertices are related, the direction of the relationship may also be important. This motivates the definition of a directed graph.

**Definition 1.14** (digraph, arc, tail, head)**.** A directed graph or *digraph* $D$ is a graph in which every edge $e$ has been given a direction from one endpoint $u$ to the other endpoint $v$. In this case we call $e$ as an *arc* from $u$ to $v$, and respectively refer to $u$ and $v$ as the *tail* and *head* of $e$.

Given a digraph $D$, we write $A(D)$ for its set of arcs and note that each element of $A(D)$ corresponds to an ordered pair of vertices.

**Definition 1.15** (outdegree, indegree)**.** Let $D$ be a digraph and $v \in V(D)$. Then the outdegree $d^+_D(v)$ of $v$ in $D$ is the number of arcs with tail $v$, and the indegree $d^-_D(v)$ of $v$ in $D$ is the number of arcs with head $v$.

When the graph $G$ is clear from the context, we will write $d^+(v)$ instead of $d^+_G(v)$ and $d^-(v)$ instead of $d^-_G(v)$.

The following is a variant of the handshaking lemma. We leave its proof as an exercise.

**Lemma 1.16.** For any digraph $D$, $\sum_{v \in V(D)} d^+(v) = \sum_{v \in V(D)} d^-(v) = |A(D)|$.

**Definition 1.17** (network, weight)**.** A *network* is a graph or digraph $G$ together with a function $w : E(G) \to \mathbb{R}$ that assigns a *weight* $w(e)$ to each edge or arc.

Digraphs and networks can be drawn in the plane with arrows to indicate arcs, and labels on lines or arrows to indicate the weight of the corresponding edges or arcs. For larger digraphs or networks, or storage in a computer, representation by a matrix will again be useful. The adjacency matrix of a simple digraph $D$ for example, is the $|V(D)| \times |V(D)|$ matrix with entry 1 in row $u$ and column $v$ if $uv \in A(D)$. Unlike adjacency matrices of graphs, adjacency matrices of digraphs

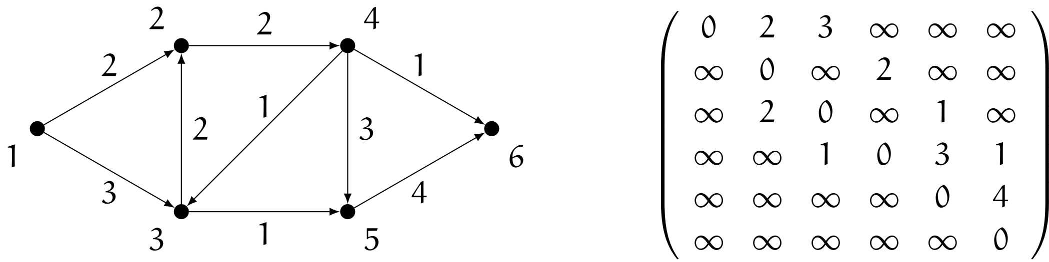



Figure 1.4: A (directed) network and its distance matrix.

are thus not typically symmetric. For networks, we can also use the entries of the matrix to represent weights. However, when using the equivalent of an adjacency matrix to represent a simple network, we have to be careful to distinguish between edges or arcs with weight zero and edges and arcs that are not part of the network. In cases where weights correspond to distances, it makes sense to represent the latter by an entry of $\infty$ in a distance matrix, following the intuition that it would take infinitely long to traverse an edge or arc that does not exist. An example of a directed network and its distance matrix are shown in Figure 1.4.

# Chapter 2

# Paths, Cycles, and Trees

We will now move beyond the neighborhood of a vertex to vertices that can be reached by following a sequence of edges. This idea is captured by the notion of a walk, and by various specializations.

**Definition 2.1** (walk, trail, path, tour, cycle)**.** A *walk* in a graph is an alternating sequence of vertices and edges in which each edge is preceded by one of its endpoints and followed by the other. The length of a walk is equal to the number of edges in the sequence. A walk is a $u$–$v$-walk if it starts with vertex $u$ and ends with vertex $v$, and closed if it starts and ends with the same vertex. A *trail* is a walk in which all edges are distinct. A *path* is a walk in which all vertices are distinct. A *tour* is a closed trail. A *cycle* is a tour containing at least one edge in which all vertices except the first and last are distinct. A *directed walk* (trail, path, tour, cycle) in a digraph is a walk (trail, path, tour, cycle) where each arc is preceded by its tail and followed by its head.

Note that we could have alternatively defined a path of length $k$ in a graph $G$ as a subgraph of $G$ isomorphic to $P_{k+1}$, and a cycle of length $k$ in $G$ as a subgraph of $G$ isomorphic to $C_k$.

## 2.1 Connectivity

Intuitively, existence of a $u$–$v$-path in a graph $G$ means that there is some way to reach $v$ from $u$ by traversing the edges of $G$, and the length of the path is the number of edges it traverses.

The following definition captures those graphs for which we can travel from any vertex to any other vertex.

**Definition 2.2** (connected)**.** A graph $G$ is *connected* if for every pair of vertices $u, v \in V(G)$ there exists a $u$—$v$-walk in $G$.

As the following result shows, we could in fact replace walks by paths without affecting the definition of connectivity.

**Lemma 2.3.** Let $G$ be a graph and $u, v \in V(G)$. Then a $u$—$v$-path exists in $G$ if and only if a $u$—$v$-walk exists in $G$.

*Proof.* The direction from left to right is obvious because every path is also a walk.

For the direction from right to left, let $W = v_0 e_1 v_1 \dots e_m v_m$ be a $u$—$v$-walk in $G$ that has minimum length among all $u$—$v$-walks in $G$. Assume for contradiction that $W$ is not a path. Then some vertex appears more than once in $W$, i.e., $v_i = v_j$ for $0 \leqslant i < j \leqslant m$. Let $W' = v_0 e_1 v_1 \dots e_i v_i e_{j+1} v_{j+1} \dots e_m v_m$. Then $W'$ is a $u$—$v$-walk in $G$, and it is shorter than $W$, which contradicts the assumption that $W$ has minimum length. Hence $W$ must be a path. □

**Corollary 2.4.** A graph $G$ is connected if and only if for every pair of vertices $u, v \in V(G)$ there exists a $u$—$v$-path in $G$.

Intuitively, if a graph is *not* connected, then it must have different sets of vertices that cannot be reached from one another. This notion is formalized in the following definition.

**Definition 2.5** (connected component)**.** A *connected component* of a graph $G$ is a maximal connected subgraph of $G$, i.e., a connected subgraph of $G$ that is not itself a subgraph of any other connected subgraph of $G$.

Note that a connected graph has a single connected component, equal to the graph itself. Figure 2.1 shows a connected graph $G_1$, and a graph $G_2$ that is not connected along with its connected components. Note that the connected components of $G_2$ partition $G_2$, i.e., they do not overlap and together form the whole graph. We will see that this is in fact true for the connected components of any graph. We need the following auxiliary lemma.

**Lemma 2.6.** Let $G$ be a graph, and consider connected subgraphs $G_1$ and $G_2$ of $G$ such that $V(G_1) \cap V(G_2) \neq \emptyset$. Then $G_1 \cup G_2$ is connected.

*Proof.* Let $x \in V(G_1) \cap V(G_2)$ and consider two arbitrary vertices $u, v \in V(G_1 \cup G_2)$. If $u, v \in V(G_1)$, then by connectivity of $G_1$ there exists a $u$—$v$-walk in $G_1$

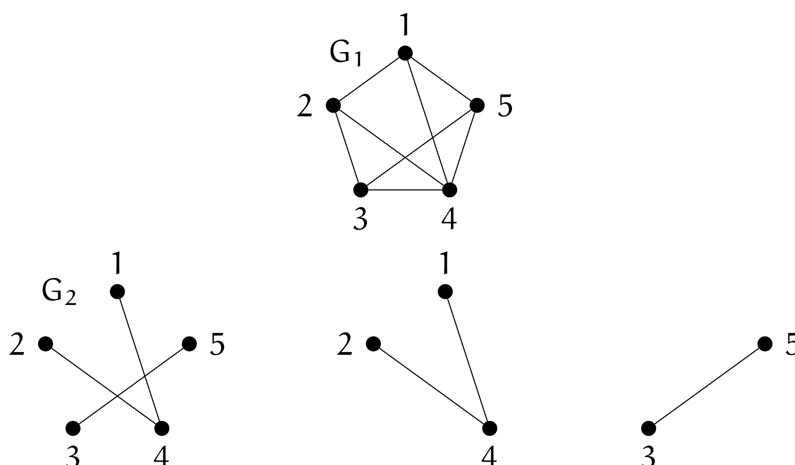



Figure 2.1: A connected graph $G_1$, and a graph $G_2$ with its two connected components

and thus in $G_1 \cup G_2$. If $u, v \in V(G_2)$, then by connectivity of $G_2$ there exists a $u$—$v$-walk in $G_2$ and thus in $G_1 \cup G_2$. Finally consider the case where $u$ and $v$ are in different connected components, and assume without loss of generality that $u \in V(G_1)$ and $v \in V(G_2)$. Since $x \in V(G_1)$, and by connectivity of $G_1$, there exists a $u$—$x$-walk in $G_1$ and thus in $G_1 \cup G_2$. Since $x \in V(G_2)$, and by connectivity of $G_2$, there exists an $x$—$v$-walk in $G_2$ and thus in $G_1 \cup G_2$. These two walks can be combined to form a $u$—$v$-walk in $G_1 \cup G_2$. We have shown for arbitrary $u, v \in V(G_1 \cup G_2)$ that there exists a $u$—$v$-path in $G_1 \cup G_2$, which means that $G_1 \cup G_2$ is connected. □

**Lemma 2.7.** Consider a graph $G$ with connected components $G_1, \ldots, G_m$. Then $\{V(G_1), \ldots, V(G_m)\}$ is a partition of $V(G)$, and $\{E(G_1), \ldots, E(G_m)\}$ is a partition of $E(G)$.

*Proof.* Since every vertex of $G$ belongs to a connected subgraph of $G$ and thus to a connected component, $\cup_{i=1}^{m} V(G_i) = V(G)$. Now assume for contradiction that $V(G_i) \cap V(G_j) \neq \emptyset$ for some $1 \leqslant i < j \leqslant m$. Then $G_i \cup G_j$ is connected by Lemma 2.6, which contradicts the assumption that $G_i$ and $G_j$ are connected components and thus maximal connected subgraphs of $G$. Thus $V(G_i) \cap V(G_j) = \emptyset$, and $\{V(G_1), \ldots, V(G_m)\}$ is a partition of $V(G)$.

Since every edge of $G$ belongs to a connected subgraph of $G$ and thus to a connected component, $\cup_{i=1}^{m} E(G_i) = E(G)$. Assume for contradiction that $E(G_i) \cap E(G_j) \neq \emptyset$ for some $1 \leqslant i < j \leqslant m$. Let $e \in E(G_i) \cap E(G_j)$ and let $v$ be one of the endpoints of $e$. Then $v \in V(G_i) \cap V(G_j)$, which contradicts the fact that $V(G_i) \cap V(G_j) = \emptyset$. Thus $E(G_i) \cap E(G_j) = \emptyset$, and $\{E(G_1), \ldots, E(G_m)\}$ is a partition of $E(G)$. □

For digraphs we can obtain stronger notions of connectivity and connected

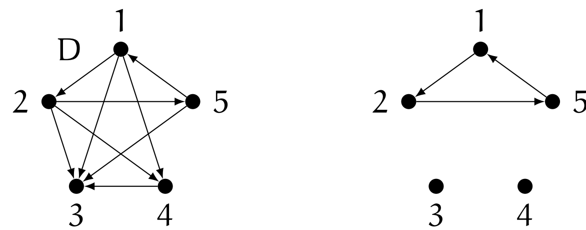



Figure 2.2: A digraph $D$ and its three strongly connected components

components by requiring the existence of a directed path between any pair of vertices.

**Definition 2.8** (strongly connected). A digraph $D$ is *strongly connected* if for every pair of vertices $u, v \in V(D)$ there exists a directed $u$—$v$-walk in $D$.

Again, this definition would not be affected if walks were replaced by paths. The proof of the following result is similar to that of Lemma 2.3 and left as an exercise.

**Lemma 2.9.** Let $D$ be a digraph and $u, v \in V(G)$. Then a directed $u$—$v$-path exists in $D$ if and only if a directed $u$—$v$-walk exists in $D$.

**Corollary 2.10.** A digraph $D$ is strongly connected if and only if for every pair of vertices $u, v \in V(D)$ there exists a directed $u$—$v$-path in $D$.

**Definition 2.11** (strongly connected component). A *strongly connected component* of a digraph $D$ is a maximal strongly connected subdigraph of $D$.

Figure 2.2 shows a digraph and its strongly connected components.

Like connected components, strongly connected components partition the set of vertices of any graph. The same is no longer true for the set of arcs, as the digraph in Figure 2.2 illustrates. The proof of the following result is similar to that of Lemma 2.7 and left as an exercise.

**Lemma 2.12.** Consider a digraph $D$ with strongly connected components $D_1, \ldots, D_m$. Then $\{V(D_1), \ldots, V(D_m)\}$ is a partition of $V(D)$.

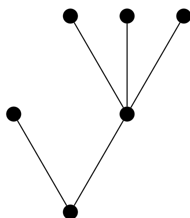


Figure 2.3: A tree with six vertices and four leaves

## 2.2 Trees

And interesting and useful class of graphs can be obtained by combining connectedness and the absence of cycles.

**Definition 2.13** (acyclic, tree, leaf). A graph $G$ is a *acyclic* if it does not contain a cycle, and a *tree* if it is both connected and acyclic. A vertex $v \in V(G)$ of a tree $G$ is a *leaf* if $d(v) = 1$.

As Figure 2.3 illustrates, trees and leafs can be drawn to resemble their natural counterparts.

Trees will be useful in particular because we will obtain them by visiting all vertices of a connected graph or connected component of a graph, and will in turn allow us to answer various questions about the original graph. But let us first spend some time looking at trees themselves.

Our first result about trees captures the intuition that the removal of a leaf from a tree produces a smaller tree.

**Lemma 2.14** (Tree induction). Every tree with two or more vertices has at least two leaves. Removing any leaf from a tree with $n$ vertices, along with its single incident edge, yields a tree with $n - 1$ vertices.

*Proof.* Consider a tree $T$ with at least two vertices, and let $v_0 v_1 \dots v_{m-1} v_m$ be a path of maximum length in $T$. Then $v_0 \neq v_m$, because there are at least two distinct vertices and there is a path between any two vertices. Moreover, since the path has maximum length and $T$ is acyclic, the edge $v_0 v_1$ is the unique edge incident to $v_0$ and the edge $v_{m-1} v_m$ is the unique edge incident to $v_m$. Thus there are at least two leaves, $v_0$ and $v_m$.

Now consider any leaf $v$ of $T$, and let $u$ be its unique neighbor. Clearly, removing vertex $v$ and the edge $uv$ from $T$ results in a graph $T'$ with $n - 1$ vertices. We need to argue that $T'$ is connected and acyclic. It is easy to see that $T'$ is acyclic, because $T$ is acyclic and $T'$ is a subgraph of $T$. To see that $T'$ is connected, consider two arbitrary vertices $s, t \in V(T')$. Note that $s, t \in V(T)$ and, because $T$ is connected, there exists an $s$—$t$-path in $T$. Note further that $v$ is

not contained in this $s$—$t$-path: $v \notin V(T')$, so $v \neq s$ and $v \neq t$; any other vertex on the path must have degree at least two, while $v$ has degree one. Hence the $s$—$t$-path is also contained in $T'$, which means that $T'$ is connected. □

The lemma can be used to inductively prove various results concerning trees, including the following on the number of edges in a tree.

**Theorem 2.15.** If $G$ is a tree, then $|E(G)| = |V(G)| - 1$.

*Proof.* We prove the result by induction on $|V(T)|$. The result clearly holds when $|V(T)| = 1$, since the only acyclic graph with one vertex does not have any edges. Now consider a tree $T$ with $n + 1 \geqslant 2$ vertices. By Lemma 2.14, $T$ has a leaf $v$, and removing $v$ from $T$ along with its single incident edge results in a tree $T'$ with $n$ vertices. By the induction hypothesis, $T'$ has $n - 1$ edges. Since $T$ has exactly one more edge than $T'$, namely the single edge we removed along with $v$, $T$ must have $n - 1 + 1 = n$ edges, as claimed. □

## 2.3 Characterizations of Trees

To gather some evidence that trees are interesting, we can convince ourselves that they can be characterized in a number of different ways. We will consider three characterizations that are particularly useful. The first two characterizations show that trees have a minimal set of edges subject to connectedness and a maximal set of edges subject to acyclicity.

**Theorem 2.16.** A graph $G$ is a tree if and only if $G$ is a *minimal connected graph*, i.e., if $G$ is connected but removing any edge from $E(G)$ yields a graph that is not connected.

*Proof.* For the direction from left to right, assume that $G$ is a tree, i.e., that it is connected and acyclic. Assume for contradiction that $G$ is not minimal. Then there exists an edge $e \in E(G)$ with endpoints $u, v \in V(G)$ such that the graph $G'$ with $V(G') = V(G)$ and $E(G') = E(G) \setminus \{e\}$ is connected, and thus a $u$—$v$-path in $G$ that does not contain the edge $e$. This $u$—$v$-path and the edge $e$ form a cycle in $G$, which contradicts the assumption that $G$ is acyclic.

For the direction from right to left, assume that $G$ is a minimal connected graph. Assume for contradiction that $G$ contains a cycle. Let $e \in E(G)$ be an arbitrary edge contained in this cycle, let $u, v \in V(G)$ be its endpoints, and let $P$ be the $u$—$v$-path obtained by removing $e$ from the cycle. We claim that the graph $G'$ with $V(G') = V(G)$ and $E(G') = E(G) \setminus \{e\}$ is connected, contradicting minimality of $G$. To see that $G'$ is connected, consider arbitrary vertices $s, t \in V(G')$. Since $G$ is connected, there exists an $s$—$t$-path $Q$ in $G$. If $Q$ does not contain edge $e$, then it is also an $s$—$t$-path in $G'$. If $Q$ does contain edge $e$, there exists an

$s$—$t$-walk in $G$ that first follows $Q$ from $s$ to $u$, then follows $P$ from $u$ to $v$, and finally follows $Q$ from $v$ to $t$. This walk does not contain edge $e$ and thus is an $s$—$t$-walk in $G'$. In both cases there exists an $s$—$t$-path in $G'$, and since $s$ and $t$ were chosen arbitrarily $G'$ is connected. □

**Theorem 2.17.** A graph $G$ is a tree if and only if $G$ is a *maximal acyclic graph*, i.e., if $G$ does not contain a cycle but adding any edge not in $E(G)$ creates a cycle.

*Proof.* For the direction from left to right, assume that $G$ is a tree, i.e., that it is connected and acyclic. We want to show that the addition of any edge not in $E(G)$ creates a cycle. Let $e$ be an edge that is not contained in $E(G)$, and let $s, t \in V(G)$ be the endpoints of this edge. Let $G'$ be the graph with $V(G') = V(G)$ and $E(G') = E(G) \cup \{e\}$. Since $G$ is connected, there exists a path $v_0 e_1 v_1 \dots e_m v_m$ in $G$ such that $v_0 = s$ and $v_m = t$. Then $v_0 e_1 v_1 \dots e_m v_m e s$ is a cycle in $G'$.

For the direction from right to left, assume that $G$ acyclic and that the addition of any edge not in $E(G)$ would create a cycle. We want to show that $G$ is connected. Consider arbitrary vertices $u, v \in V(G)$. If $G$ contains an edge with endpoints $u$ and $v$, it clearly also contains a $u$—$v$-path. If $G$ does not contain edge $e$ with endpoints $u$ and $v$, then by assumption the graph $G'$ with $V(G') = V(G)$ and $E(G') = E(G) \cup \{e\}$ contains a cycle $u e_1 v_1 \dots e_m v e u$, and the path $u e_1 v_1 \dots e_m v$ is a $u$—$v$ path in $G$. In both cases there exists a $u$—$v$-path in $G$, and since $u$ and $v$ were chosen arbitrarily $G$ is connected. □

The third characterization works in terms of the existence of a unique path between any pair of vertices. We leave its proof as an exercise.

**Theorem 2.18.** A graph $G$ is a tree if and only if it contains no loops and a unique $u$—$v$-path for every $u, v \in V$.

## 2.4 Counting Trees

Recall that one of our first results, which followed almost immediately from the definition of a simple graph, gave us the number of distinct simple graphs with a given set of vertices. It is natural to ask in the same way for the number of distinct trees, but here the problem seems much more complicated at least if we approach it directly. A direct approach to determine the number of trees with a given number of vertices would be to determine first the set of unlabeled trees, corresponding to the different shapes a tree can take, and then count the number of distinct labelings for each of the unlabeled trees. Our goal in the next paragraph will be to convince ourselves that we do not want to count trees directly, so if you find it somewhat hard to follow just keep reading.

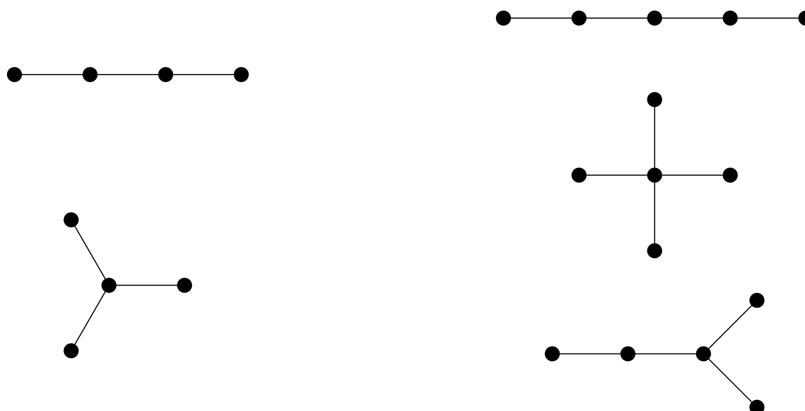


Figure 2.4: Unlabeled trees on 4 and 5 vertices

It is not difficult to see that there is a single tree each with one and two vertices. For three vertices there is a single unlabeled tree, a path. This tree has 3 distinct labelings, corresponding to the difference choice of label for the vertex at the center, so there are 3 distinct trees with three vertices. For four vertices there are two distinct unlabeled trees, shown on the left of Figure 2.4. The one at the top has $4!/2$ distinct labelings, since each left-to-right labelling and its reverse yield the same graph. The one at the bottom has only 4 distinct labelings, since for each possible choice of label for the vertex at the center all labelings of the outer vertices yield the same graph. There are thus $12 + 4 = 16$ distinct trees with four vertices. Things get even more complicated for five vertices, since there are now three unlabeled trees, shown on the right of Figure 2.4. We can convince ourselves that these trees respectively have $5!/2 = 60$, 5, and $5!/2$ distinct labelings, which implies that there are $5!/2 + 5 + 5!/2 = 125$ distinct trees with four vertices.

By now you are probably convinced that trees are difficult to count directly, but given the above numbers we may conjecture that the number of labeled trees with $n \geqslant 2$ vertices is $n^{n-2}$. Fortunately a number of elegant ways have been found to prove this. One of them works by mapping each tree to a distinct code, specifically a sequence of labels, and then counting the number of codes.

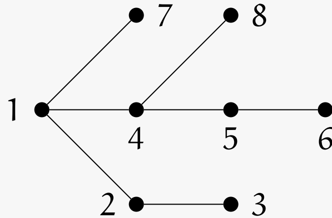
**Algorithm 2.19** (Prüfer code)**.** Let $T$ be a tree with $V(T) = [n]$. The *Prüfer code* for $T$ is a sequence of length $n - 2$ over $[n]$ constructed as follows. Start with the empty sequence, then repeat the following steps until only two vertices remain in $T$.

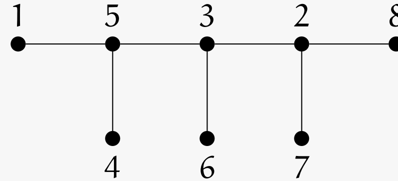
1. Let $v$ be the leaf in $T$ with *smallest* label.
2. Let $u$ be the unique vertex adjacent to $v$ in $T$. Write down $u$ as the next number in the sequence.
3. Delete $v$ from $T$, along with its single incident edge $uv$.

Observe that by Lemma 2.14 the algorithm is guaranteed to produce a code for every tree with $n \geqslant 2$ vertices, and that the length of this code will indeed be

$n-2$.

**Example 2.20.** Consider the following trees:

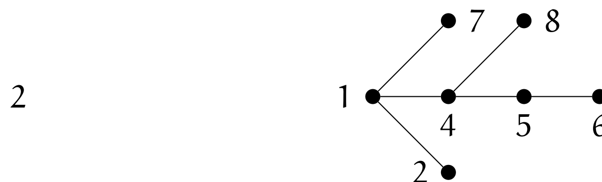




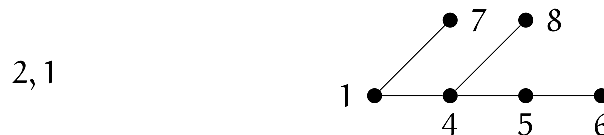

The Prüfer codes of these trees respectively are $2, 1, 5, 4, 1, 4$ and $5, 5, 3, 3, 2, 2$.

We determine the Prüfer code for the tree on the left and leave the tree on the right as an exercise.

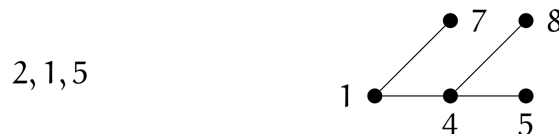
1. The smallest leaf is 3, its unique neighbor 2. Add 2 to the sequence and delete 3 from the tree to obtain




2. The smallest leaf is 2, its unique neighbor 1. Add 1 to the sequence and delete 2 from the tree to obtain



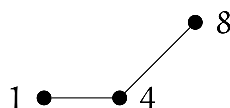

3. The smallest leaf is 6, with an edge to 5. Write down 5, delete 6 from the graph to obtain




4. The smallest leaf is 5, with an edge to 4. Write down 4, delete 5 from the graph to obtain

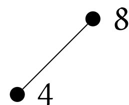
5. The smallest leaf is 7, with an edge to 1. Write down 1, delete 7 from the graph to obtain

$2, 1, 5, 4, 1$




6. The smallest leaf is 1, with an edge to 4. Write down 4, delete 1 from graph to obtain

$2, 1, 5, 4, 1, 4$




7. Only two vertices remain, so we are done.

It turns out that distinct trees produce distinct codes and that every sequence is the code of some tree. This allows us to count trees by counting codes.

**Theorem 2.21.** For any $n \geqslant 2$, there are $n^{n-2}$ distinct trees with $n$ vertices.

*Proof.* Algorithm 2.19 defines a function from the set of trees with $n$ vertices to the set of sequences of length $n-2$ over $[n]$. We will show that this function is injective, i.e., that the algorithm yields different codes for different trees, and surjective, i.e., that every sequence of length $n-2$ over $[n]$ is obtained as the code of some tree with $n$ vertices. The algorithm thus defines a bijective function between the two sets, which means that their cardinalities must be the same. The claim then follows from the fact that there are $n^{n-2}$ distinct sequences of length $n-2$ over $[n]$.

We will use two simple observations about Algorithm 2.19. First, each vertex $v \in V(T)$ appears exactly $d(v)-1$ times in the Prüfer code of $T$. This is because the degree of a vertex decreases by one each time its label is written down and the label of a vertex with degree one is never written down. Second, once the first leaf $v$ of $T$ has been deleted and its unique neighbor $u$ written down, the rest of the Prüfer code will be equal to the Prüfer code of the tree $T'$ with $V(T') = V(T) \setminus \{v\}$ and $E(T') = E(T) \setminus \{uv\}$. This is because the algorithm proceeds with $T'$ once $v$ has been written down.

Let us first show that the algorithm yields different codes for different trees. We do this by induction on $n$. The claim trivially holds when $n = 2$, in which case there is a unique labeled tree and a unique code, the sequence of length zero. Now assume that $n \geqslant 3$ and consider two distinct trees $T_1$ and $T_2$ with vertex set $[n]$. Let $v_1$ be the leaf with smallest label in $T_1$, and $u_1$ its unique neighbor. Let $v_2$ the leaf with smallest label in $T_2$, and $u_2$ its unique neighbor. If $v_1 \neq v_2$, then $d_{T_1}(v_1) \neq d_{T_2}(v_1)$ and, since $v_1$ appears $d_{T_1}(v_1) - 1$ times in the Prüfer code for $T_1$ and $d_{T_2}(v_2)-1$ times in the Prüfer code for $T_2$, $T_1$ and $T_2$ have different Prüfer codes. If $v_1 = v_2$ and $u_1 \neq u_2$, then the Prüfer codes for $T_1$ and $T_2$ start with different numbers and are thus distinct. Finally, if $v_1 = v_2$ and $u_1 = u_2$, then the Prüfer codes for $T_1$ and $T_2$ start with the same number and continue with the

respective Prüfer codes of the trees $T_1[[n] \setminus \{v_1\}]$ and $T_2[[n] \setminus \{v_1\}]$. These trees are distinct, and by the induction hypothesis their Prüfer codes are distinct as well.

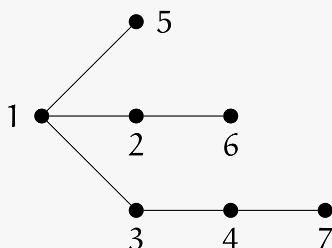
Let us now show that that every sequence is the code of some tree. We again do this by induction on $n$. As before, the claim trivially holds when $n = 2$. Now assume that $n \geqslant 3$ and consider a sequence $S$ of length $n-2$ over $[n]$. Let $v$ be the smallest element of $[n]$ that does not appear in $S$, $u$ the first element of $S$, and $S'$ the sequence obtained by removing the first element from $S$. $S'$ is a sequence of length $n-3$ over $[n] \setminus \{v\}$, so by the induction hypothesis there exists a tree $T'$ with $V(T') = [n] \setminus \{v\}$ whose Prüfer code is $S'$. Observe that none of the elements in $S'$ is a leaf in $T'$: when an element of $S'$ was written down by the algorithm, it was adjacent to a leaf in a tree with at least three vertices and thus not itself a leaf. Let $T$ be the tree obtained from $T'$ by adding vertex $v$ and edge $vu$, and observe that none of the elements of $S$ is a leaf in $T$. Then the Prüfer code of $T$ starts with $u$, which is the smallest leaf in $T$, and continues with $S'$. It is thus equal to $S$, so we have found a tree with Prüfer code $S$. □

Since Algorithm 2.19 defines a bijection it must have an inverse, i.e., a procedure that converts a code into a tree. Given a sequence of length $n-2$ over $[n]$, this inverse procedure starts by adding $*$ to the end of the sequence to help us remember this position. Then it repeats the following until $*$ is at the beginning of the sequence:

1. Let $s$ be the number at the start of the sequence, and $x$ the smallest number not currently in the sequence.
2. Add the edge $sx$ to the tree, along with any vertices not already present.
3. Remove $s$ from the start of the sequence, and add $x$ to the end of the sequence.

Finally, when $*$ is at the start of the sequence, two numbers $x, y \in [n]$ are missing from the sequence. We add the edge $xy$ to the tree, along with any vertices not already present.

**Example 2.22.** The tree with Prüfer code $1, 2, 1, 3, 4$ is the following:

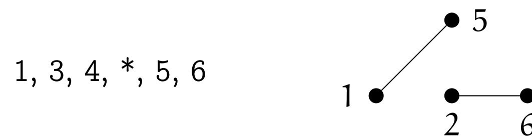



The Prüfer code has 5 elements and thus corresponds to a tree with vertex set $[7]$.

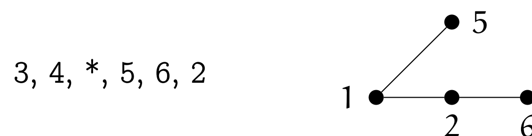
1. Add $*$ to the end of the Prüfer code to obtain the sequence $1, 2, 1, 3, 4, *$.
2. The first number in the sequence is 1, the smallest number not in the sequence is 5. Add the edge $\{1, 5\}$ to the tree, remove 1 from the beginning of

the sequence and add 5 to the end to obtain

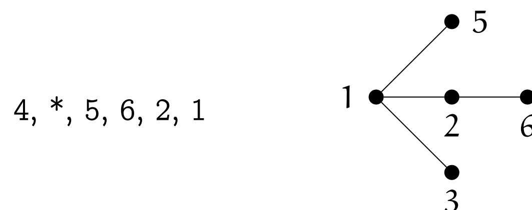
3. The first number in the sequence is 2, the smallest number not in the sequence is 6. Add the edge $\{2,6\}$ to the tree, remove 2 from the beginning of the sequence and add 6 to the end to obtain



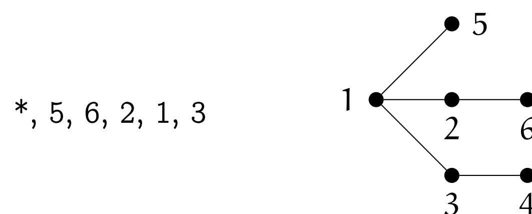

4. The first number in the sequence is 1, the smallest number not in the sequence is 2. Add the edge $\{1,2\}$ to the tree, remove 1 from the beginning of the sequence and add 2 to the end to obtain



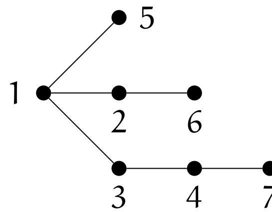

5. The first number in the sequence is 3, the smallest number not in the sequence is 1. Add the edge $\{3,1\}$ to the tree, remove 3 from the beginning of the sequence and add 1 to the end to obtain




6. The first number in the sequence is 4, the smallest number not sequence is 3. Add the edge $\{4,3\}$ to the tree, remove 4 from the beginning of the sequence and add 3 to the end to obtain




7. The sequence now begins with $*$, and the two smallest numbers not in the sequence are 4 and 7. Add the edge $\{4,7\}$ to the tree to obtain




## 2.5 Spanning Trees

Let us now return to our claim that trees are useful in exploring the connected components of a graph. A useful notion in this context is that of a spanning tree, a subgraph that is a tree and spans all vertices.

**Definition 2.23.** Let $G$ be a graph. Then a tree $T$ is a *spanning tree* of $G$ if it is a subgraph of $G$ and $V(T) = V(G)$.

Indeed, the graphs possessing a spanning tree are precisely those that are connected.

**Theorem 2.24.** A graph is connected if and only if it has a spanning tree.

We will not prove this result here, but will obtain it soon as a corollary of Theorem 4.4. It implies that we can test connectivity of a graph by attempting to grow a spanning tree within it. If a graph is not connected, we can grow a spanning tree for each connected component and thus determine the connected components. Once we have a spanning tree, we can also find paths and cycles. By Theorem 2.18, if a graph contains a $u$—$v$-path, there is a unique $u$—$v$-path in the spanning tree. By Theorem 2.17, any edge that is in the graph but not in the spanning tree yields a cycle.

# Chapter 3

# Complexity of Algorithms and Problems

The most important property of any algorithm is its correctness, its ability to solve the problem it claims to solve. Once we have established correctness of an algorithm, which in the case of Algorithm 2.19 we have done by proving that it determines for each tree a distinct Prüfer code, we may ask for its complexity. Here, what we mean by the complexity of an algorithm is the amount of time required to run it or the amount of space required to carry out individual steps and store intermediate results.

Our motivation in considering Algorithm 2.19 was a theoretical one, because it enables us to count trees by counting codes. We can do this without ever converting a particular tree into the corresponding code, and may thus not actually care about the complexity of the algorithm. For other, more practical, applications of Algorithm 2.19, like the contruction of a random tree, the complexity of the algorithm is important. All algorithms we will consider from now on will be motivated mainly by practical applications, and we will be interested in their complexity.

In this module we will focus on running time as a measure of complexity, because it is the most important such measure and has obvious implications for other measures like space: if we only have a certain amount of time available to execute an algorithm, this automatically limits the amount of intermediate results we can write down. An implication in the opposite direction turns out to hold as well, but it is much less obvious: the amount of available space limits the overall time of our computation, even if we can erase past steps and reuse the space.

We will measure running time in terms of the number of basic computational steps. The exact number of such steps will of course depend on the model of computation we use, and what the basic steps are in that model. It turns out that the right thing to do is to use a very simple theoretical model of computation, the Turing machine. Despite the simplicity of the Turing machine, it can be shown that even the most advanced computer ever built can only perform the equivalent

of a limited number of steps of a Turing machine in a single step. It is conjectured that the same is in fact true for any computer that could theoretically be built.

By measuring running time at a level of precision that ignores the differences in performance between the Turing machine and modern computers we will then benefit in two ways: we can measure running time relatively coarsly, and ignore small differences arising from the use of a particular model of computation; and the running times we obtain apply to all computers ever built rather than to a particular theoretical model.

## 3.1 Input and Running Time

Just like it takes longer to determine the Prüfer code of a tree with a larger number of vertices, the running time of most algorithms will grow with the size of the problem instance they are attempting to solve. It thus makes sense to express the running time of an algorithm as a function of the size of its input. As the Turing machine works with strings of bits, we will measure input size in terms of the number of bits required to represent the input. For problems concerning graphs the input can be represented by a incidence or adjacency matrix, which can be written down using a number of bits that is a linear function of the number of vertices and edges. For problems concerning networks, or other problems that involve numerical quantities, the size of the input also grows logarithmically in the magnitude of the numerical quantities, because this is the number of bits required to represent these quantities. It can be seen from this discussion that we could cheat, and make an algorithm appear faster relative to the size of its input, by not representing the input in the most efficient way. We will be careful not to cheat in this way.

We will focus in this module on algorithms that terminate after a finite amount of time for every input and that are deterministic in the sense that they will always produce the same output for a given input.

**Definition 3.1** (running time). The *running time* $T(n)$ of an algorithm is the maximum, over all problem instances of size $n$, of the number of basic operations used by the algorithm in solving such an instance.

To make this definition mathematically precise we would have to define precisely what we mean by a basic operation, and we could do so for example by using the operations available to a Turing machine or those available to a modern computer. We can, however, use the definition without making it precise. As we have mentioned, the Turing machine to model any existing computer with only a moderate increase in running time. Thus, if we limit ourselves to measuring running times only up to a factor equal to that increase, we can make statements

about the running time that are independent of the computational model and in particular apply to every computer built so far. In doing so, we can think of a basic operation relatively vaguely as a simple instruction in a programming language or a simple step carried out by hand on a piece of paper.

## 3.2 Asymptotic Upper Bounds

In order to formally describe the running time of an algorithm, while ignoring small differences that arise from a particular choice of computational model, we use the following notion of asymptotic upper bounds.

**Definition 3.2** (asymptotic upper bound). Consider two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$. Then $f(n)$ is $O(g(n))$, or *asymptotically bounded from above* by $g(n)$, if there exist constants $c > 0$ and $n_0 \geqslant 0$ such that for all $n \geqslant n_0$, $f(n) \leqslant c \cdot g(n)$.

Intuitively $f(n)$ is $O(g(n))$ if for sufficiently large $n$ it is bounded from above by a constant multiple of $g(n)$. If we imagine $f(n)$ to be the running time of an algorithm for input size $n$, which may be both complicated and difficult to determine exactly, this notation will allow us to give a different function $g$ that bounds the running time from above, as long as we ignore the behavior of the algorithm for small values of $n$ and factors that do not depend on $n$.

We will use Definition 3.2 to express the running time of an algorithm in simple terms, and will therefore always use a function $g$ that is simple and in particular does not involve a leading constant other than one. At the same time we will prefer functions $g$ that grow more slowly because they implies a slower growth of $f$. We should keep in mind, however, that a function $g$ that grows more slowly may make it more difficult to show that Definition 3.2 holds, or indeed it may not hold at all for such a function.

**Example 3.3.** Let $f(n) = 4n^3 + 6n^2 + 3n + 4$. For $n \geqslant 1$, $1 \leqslant n \leqslant n^2 \leqslant n^3$ and thus $f(n) \leqslant 4n^3 + 6n^3 + 3n^3 + 4n^3 = 17n^3$. If we set $g(n) = n^3$, $n_0 = 1$, and $c = 17$, then for all $n \geqslant n_0$, $f(n) \leqslant c \cdot g(n)$, so $f(n)$ is $O(n^3)$.

Note that Definition 3.2 only requires $g$ to be an upper bound of $f$, and it is also true for example that $f$ is $O(n^4)$. However, $O(n^3)$ is the best bound for $f$ we can give, in the sense that it is simple and up to constant factors has the same asymptotic growth as $f$.

Let us now consider how asymptotic notation can be used to describe the running time of an algorithm. We begin with an algorithm from linear algebra that we know quite well, which allows us to determine the rank of a matrix $A$.

**Example 3.4.** Recall that to determine the rank of a matrix $A \in \mathbb{R}^{n\times n}$ we can use Gaussian elimination to bring the matrix into row echelon form and then count the number of non-zero rows in the latter.

Gaussian elimination proceeds in a number of rounds $i = 1, 2, \dots, n-1$. In the $i$th round, the $i$th row of $A$ is multiplied by a constant, and then a multiple of the new $i$th row is added to each of the rows $j = i+1, \dots, n$. Multiplication of row $i$ by a constant requires at most $n$ multiplications of pairs of numbers. Addition of a multiple of row $i$ to row $j$ requires at most $n$ multiplications and $n$ additions, and in a given round we do this at most $n-1$ times. Once Gaussian elimination is complete, we can determine the rank of the original matrix by counting the number of non-zero rows of the matrix we have computed. This can be done by comparing at most $n^2$ values to zero. If we count multiplication and addition of pairs of numbers and comparison to zero as basic operations, then Gaussian elimination requires at most $(n-1)(n+2n(n-1))+n^2 = 2n^3-2n^2+n = O(n^3)$ basic operations. This analysis is actually a bit generous because row operations become easier in later rounds as entries below the diagonal become equal to zero, but a more careful analysis only changes the bound by a constant factor and we choose to ignore such constant factors.

Note that in the example we have expressed the running time of the algorithm in terms of a natural parameter of the problem, the size $n$ of matrix $A$, rather than the size of the problem input, i.e., the number of bits required to store $A$. We have done so because it is convenient, and it is acceptable as long as we are aware of what we do and explain it precisely. If in this particular example we wanted to measure running time relative the input size, we would for example have to explain for example how we would approximate real numbers within a finite number of bits. A potential benefit of such a more detailed analysis could be that it would allow us to justify our decision that simple arithmetic operations like addition and multiplication should count as basic operations.

Luckily, in the context of this module, we will not have to worry about the way in which numbers are stored and simple arithmetic operations are performed. We do have to keep in mind, however, that the running time of an algorithm may depend on the magnitude of the numbers involved, and we will encounter an example later where this is the case.

Let us use the algorithm that computes the Prüfer code of a tree as another example.

**Example 3.5.** Consider a tree $T$ with $V(T) = [n]$ that is represented as a sequence of edges. Algorithm 2.19 runs for $n-2$ rounds, because it removes

a vertex from $T$ in each round and stops when two vertices are left. The first step in each round requires us to find the smallest leaf, which can be achieved using $O(n)$ basic operations: we first create a vector of length $n$ and all entries equal to $0$; we then consider each of the $n-1$ edges in turn and increase the two entries of the vector corresponding to the endpoints of the edge by one each; when this has been done the entries of the vector correspond to the degrees of the vertices in $T$, and we can find the smallest leaf by going over the vector from the beginning to determine the index of the first entry that is equal to $1$. The two remaining steps in each round, of writing down the neighbor of the smallest leaf and deleting the leaf and its unique incident edge, can be completed in $O(n)$ basic operations by going over to the sequence of edges to find the edge in question, writing down its other endpoint, and deleting it from the sequence. The overall running time of the algorithm is thus $(n-2)(O(n)+O(n)) = O(n^2)$.

Note that we have again made a decision regarding basic operations, namely that writing down a number or comparing two numbers should qualify as a basic operation while finding the smallest leaf in a tree should not. This decision is reasonable and not difficult to justify, but we should be aware that we have made it.

A refined version of Algorithm 2.19 can in fact produce a Prüfer code in a running time of only $O(|V(T)|)$, which differs by more than just a constant factor from the bound given above. However, even the distinction between a linear and a quadratic running time will be beyond the scope of this module. Instead we will consider as efficient any algorithm whose running time is bounded by a polynomial in $n$.

**Definition 3.6** (polynomial-time algorithm). Consider an algorithm with a running time $T(n)$. Then the algorithm is a *polynomial-time algorithm* if there exists a constant $k$ such that $T(n)$ is $O(n^k)$.

The distinction between polynomial running times and running times with faster asymptotic growth, like exponential ones, may seem quite arbitrary. After all, we would be much happier for reasonable values of $n$ with an exponential running time of $1.00001^n$ than with a polynomial running time of $n^{100}$. The most compelling argument is perhaps that the distinction has turned out to work very well in practice: existence of some polynomial-time algorithm usually means that there is a polynomial-time algorithm where the degree of the polynomial is very small, whereas problems for which no polynomial-time algorithm is known tend to be genuinely difficult. To illustrate that there really is a qualitative difference between polynomial and non-polynomial running times, Table 3.1 lists example running times for various problem sizes.

| $n$ | $n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|
| 10 | $<1s$ | $<1s$ | $<1s$ | $<1s$ | 4s |
| 30 | $<1s$ | $<1s$ | $<1s$ | 18m | $10^{25}$y |
| 50 | $<1s$ | $<1s$ | $<1s$ | 36y | |
| $10^2$ | $<1s$ | $<1s$ | 1s | $10^{17}$y | |
| $10^3$ | $<1s$ | 1s | 18m | | |
| $10^4$ | $<1s$ | 2m | 12d | | |
| $10^5$ | $<1s$ | 3h | 32y | | |
| $10^6$ | 1s | 12d | 31,710y | | |

Table 3.1: Approximate example running times relative to problem size $n$, under the assumption that one million steps can be taken per second. Entries exceeding $10^{25}$ years have been left empty.

## 3.3 Complexity of Problems: P and NP

Now that we have a way to measure the speed of an algorithm, it makes sense to define the difficulty of a problem, such as that of determining the connected components of a graph, as the running time of the fastest algorithm that solves the problem. This leads to the class P of problems that can be solved in polynomial time.

**Definition 3.7** (complexity class P). The *complexity class P* is the set of all problems for which there exists a polynomial-time algorithm.

It is clear how we would show that a problem is in P: by giving an algorithm and proving that it solves the problem and requires only polynomially many steps. But what if for a given problem we cannot find a polynomial-time algorithm and suspect that no such algorithm exist? Is there a way to show that a problem cannot be solved in polynomial time? This seems difficult, because it would require us to argue that any conceivable algorithm for solving the problem is slow. And indeed, a way to do this has so far not been found. However, there does exist a way for showing that a problem cannot be solved efficiently that relies on the famous conjecture that $P \neq NP$. We just defined P as the set of problems that can be solved in polynomial time. NP, which stands for "nondeterministic polynomial" rather than "not polynomial," is the set of problems for which a given solution can be verified in polynomial time. The conjecture thus claims, intuitively, that it is more difficult to find a solution than to verify a given solution. This seems plausible, if we for example compare the difficulty of answering an exam question to that of understanding the model solution to a past exam. But so far nobody has been able to prove the conjecture, and it has been selected by the Clay Mathematics Institute as one of its Millennium Prize Problems, a

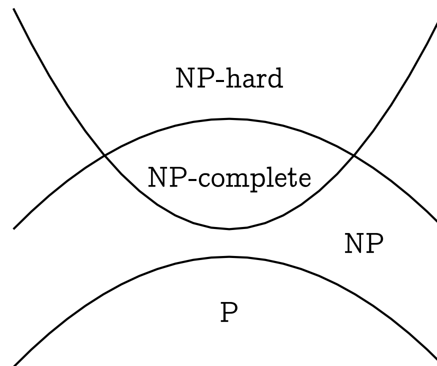



Figure 3.1: Relationship between P, NP, and the set of NP-hard problems. It is not known whether the intersection between P and the set of NP-complete problems is empty. If it is not, then P = NP.

solution to which would be awarded a prize of $1M.

While we do not currently have a way to identify problems that are not in P, we can identify problems that are what is called NP-hard, at least as hard as any problem in NP. We can then define NP-complete problems as problems that are both NP-hard and in NP. NP-complete problems are therefore the most difficult problems in NP. If indeed P $\neq$ NP, then the NP-hard problems are the ones that are not in P. NP-hardness thus provides evidence that a problem is in fact hard, and that we should not not waste our time trying to find a polynomial-time algorithm for it. The relationship between P, NP, and the sets of NP-hard and NP-complete problems is illustrated in Figure 3.1.

# Chapter 4

# Graph Traversal

Let us now return to algorithms that work with graphs.

## 4.1 Breadth-First and Depth-First Search

We begin with an algorithm that takes a graph $G$ and a vertex $v \in V(G)$, and completely explores the connected component containing $v$ by growing a maximal tree from $v$. In this context, $v$ is also called the root of the tree.

**Algorithm 4.1** (tree search). Consider a graph $G$ and a vertex $v \in V(G)$. *Tree search* in $G$ from $v$ starts from the tree with $V(T) = \{v\}$ and $E(T) = \emptyset$, and then repeats the following steps:

1. Let $u \in V(T)$ and $w \in V(G) \setminus V(T)$ such that $uw \in E(G)$. If no such element exists, then stop.
2. Add $w$ to $V(T)$ and $uw$ to $E(T)$.

It is worth noting at this point that the edge $uw$ in Step 1 may not be unique, so that the algorithm may have a choice which edge to select. In cases like this, we will be interested in and prove results about any implementation of the algorithm that is deterministic, i.e., that makes the choices in some arbitrary but consistent manner that only depends on the input.

Two variants of Algorithm 4.1 are particularly interesting.

**Algorithm 4.2** (breadth-first search, depth-first search). In Step 1 of Algorithm 4.1, let $U$ be the sequence of vertices in $V(T)$ in the order in which they were added to $V(T)$. *Breadth-first search* selects $u$ be the first element in $U$ such that there exists $w \in V(G) \setminus V(T)$ with $uw \in E(G)$. *Depth-first search* selects $u$ be the last element in $U$ such that there exists $w \in V(G) \setminus V(T)$ with $uw \in E(G)$.

Breadth-first search takes its name from the fact that it adds vertices to $T$ in

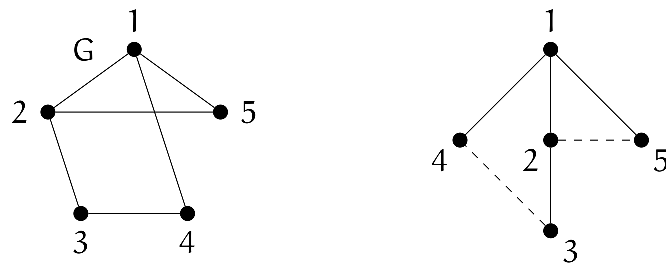
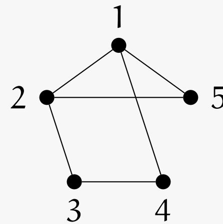



Figure 4.1: A graph G and one of its spanning trees. Vertices of the spanning tree have been arranged in layers according to their distance in G from vertex 1. Edges that are in G but not in the spanning tree are drawn as dashed lines.

increasing order of their distance from $v$, where the distance of $u$ from $v$ is the minimum length of a $v$—$u$-path. If we arrange the vertices of G in layers according to their distance from $v$, as shown in Figure 4.1, then breadth-first search adds all vertices in a given layer to T before any vertices in the following layer. Depth-first search, on the other hand, explores deeper layers first.

**Example 4.3.** Consider the following graph:




If we follow the convention that $w$ in Step 1 is selected to have the smallest possible label, breadth-first search and depth-first search from vertex 1 respectively produce the following tree:

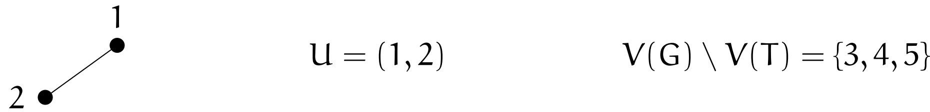
We consider the behavior of breadth-first search in detail and leave that of depth-first search as an exercise. We start from a tree containing only vertex 1.

1

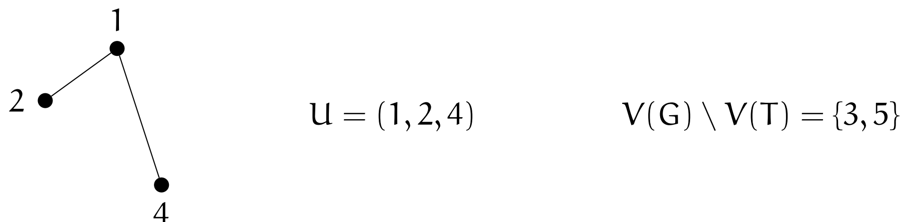
$$U = (1) \qquad V(G) \setminus V(T) = \{2, 3, 4, 5\}$$

Our goal is now to grow the tree by adding an edge from a vertex within the tree

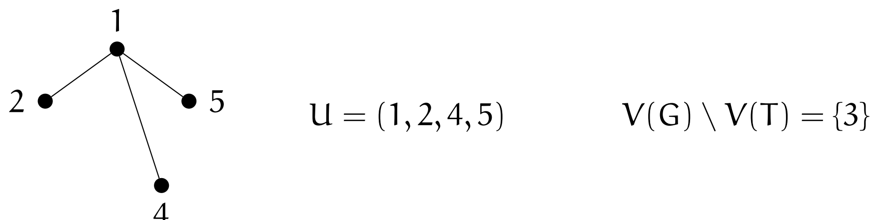
to a vertex outside the tree. Our only choice for the former is vertex 1, and for the latter we choose among the neighbors of vertex 1 the vertex with the smallest label, vertex 2. By adding vertex 2 and the edge $\{1, 2\}$ to the tree we obtain



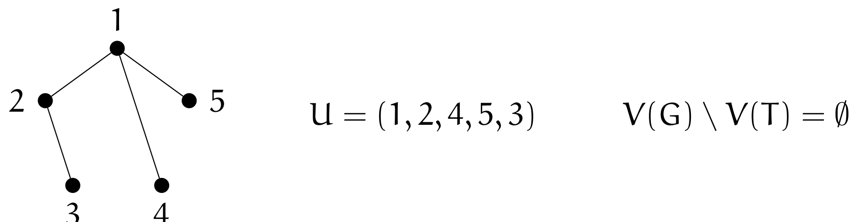

We continue to grow the tree by adding an edge from a vertex within the tree to a vertex outside it. Since there still exists such an edge incident to vertex 1, we choose vertex 1 as the vertex within the tree. As the vertex outside the tree we choose among the neighbors of vertex 1 the vertex with the smallest label, vertex 4. By adding vertex 4 and the edge $\{1, 4\}$ to the tree we obtain




We grow the tree further by choosing vertex 1 from within the tree and vertex 5 outside it. By adding vertex 5 and the edge $\{1, 5\}$ to the tree we obtain




While there remain edges between a vertex inside the tree and a vertex outside it, none of these edges is incident to vertex 1. We thus select vertex 2 as the vertex inside the tree and add vertex 3 and the edge $\{2, 3\}$ to the tree to obtain




The following result establishes that tree search algorithms, and breadth-first and depth-first in particular, produce the desired outcome.

**Theorem 4.4.** For any graph $G$ and any vertex $v \in V(G)$, Algorithm 4.1 stops after at most $|V(G)|$ iterations. When the algorithms stops, $T$ is a spanning

tree of the connected component of $G$ that contains $v$.

*Proof.* In a given iteration, the algorithm either stops or adds a vertex to the tree. It thus stops after at most $|V(G)|$ iterations. It is easy to show by induction that throughout the algorithm, $T$ is always a tree. This holds trivially in the beginning because we start from a single vertex. Assuming that it holds at the beginning of a particular iteration by the induction hypothesis, it then also holds at the end of each iteration because we add a new vertex and an edge between the new vertex and a vertex previously in the tree. This edge guarantees the existence of a path between the new vertex any vertex previously in the tree, and it cannot form a cycle with the edges already present. When the algorithm stops, it does so because no edge exists that has exactly one endpoint in $V(T)$. This means that $T$ is a maximal graph that contains vertex $v$, is connected, and does not contain a cycle, i.e., a spanning tree of the connected component of $G$ that contains $v$. □

Since the first step in each iteration can be performed with $O(|E(G)|)$ basic operations, and the second step with a constant number of basic operations, the running time of the algorithm is $O(|V(G)| \cdot |E(G)|)$. With some care it is in fact possible to reduce the running time to $O(|V(G)| + |E(G)|)$, because there is no need to consider any edge more than once.

**Example 4.5.** Assume that you want to connect computers in a network by adding links between pairs of computers. There may be some restrictions on the links that can be installed, but each link has the same installation cost. If we represent computers and possible links by a graph, we want to connect computers by a spanning tree, which by Theorem 2.16 guarantees with a minimal number of links that any computer can communicate with any other computer. It does not matter which spanning tree we choose, as all spanning trees have the same number of edges and thus the same cost. We can find a spanning tree, or establish that no spanning tree exists, by running a tree search from an arbitrary vertex in the graph.

## 4.2 Connected Components

Algorithm 4.1 finds a spanning tree of the connected component of a graph $G$ that contains the particular vertex $v$ the algorithm starts from. We can thus test connectivity of a graph $G$ by running Algorithm 4.1 from an arbitrary vertex and testing whether the tree $T$ produced satisfies $V(T) = V(G)$.

By Lemma 2.7 the connected components of a graph partition its set of vertices, so each vertex is contained in exactly one connected component. We can thus determine the connected components of a graph by running Algorithm 4.1 repeatedly, first from an arbitrary vertex, and then from a vertex not contained

in any of the connected components found so far until no such vertex exists. The set of vertices of each connected component is equal to the set of vertices of its spanning tree, and the connected component itself to the subgraph of $G$ induced by by its set of vertices.

## 4.3 Paths and Cycles

Consider a graph $G$, and let $T$ be the tree produced by running Algorithm 4.1 on $G$ from a vertex $v \in V(G)$. By Theorem 4.4, $T$ is a spanning tree of the connected component containing $v$. Moreover, by Theorem 2.18, $T$ contains a unique $v$—$u$-path for every $v \in V(T)$. In the case of breadth-first search, these $v$—$u$-paths in fact have minimum length among all $v$—$u$-paths in $G$.

**Theorem 4.6.** Let $G$ be a graph and $v \in V(G)$. Let $T$ be a tree obtained from breadth-first search of $G$ starting at $v$, and $u \in V(T)$. Then the unique $v$—$u$-path in $T$ has minimum length among all $v$—$u$-paths in $G$.

*Proof.* Consider the connected component of $G$ that contains $v$. Arrange its vertices into layers as in Figure 4.1, such that layer $i$ contains all vertices $u$ for which the shortest $v$—$u$-path in $G$ has length $i$. We can now argue by induction over the layers.

The claim holds trivially for layer 0, which contains only vertex $v$. Now assume that the statement holds for all vertices in layer $k$ and consider a vertex $u$ in layer $k+1$. By definition of the layers, $G$ contains an edge between $u$ and some vertex in layer $k$, and does not contain any edges between $u$ and any vertices in layers $l < k$. Note further that vertices in layer $k$ appear in the ordered sequence $U$ used by breadth-first search before any vertices in layers $l > k$. The algorithm thus adds to $T$ an edge between $u$ and a vertex $w$ in layer $k$. Together with a $v$—$w$-path of minimum length in $G$, which is contained in $T$ by the induction hypothesis, this edge forms a $v$—$u$-path of minimum length in $G$. □

**Example 4.7.** Again consider the situation of Example 4.5, but now assume that we want to make sure that each computer can communicate with a central server, with the additional requirement that this communication happens via a minimum number of other computers. By Theorem 4.6, we can run breadth-first search from the vertex representing the server and install the links in the resulting spanning tree to achieve this property.

Algorithm 4.1 can also be used to find cycles. A graph contains a cycle if and only if at least one of its connected components does. Moreover, by Theorem 2.17, a connected components contains a cycle if and only if it is not a tree. We can thus determine whether a graph contains a cycle by determining its connected

components, along with a spanning tree for each component, and test whether any component contains any edges in addition to those contained in its spanning tree.

## 4.4 Strongly Connected Components

With some slight modifications, Algorithm 4.1 can also be used to test strong connectivity and find the connected components of a digraph.

Consider a digraph $D$, and observe that two vertices $u, v \in V(D)$ are contained in the same strongly connected component if and only if there exists a directed path from $u$ to $v$ and a directed path from $v$ to $u$. If we modify Step 1 of Algorithm 4.1 to select $u$ and $w$ such that $uw \in A(G)$, the algorithm constructs a maximal tree in which all arcs are directed away from the root. If we start this modified algorithm from some vertex $v \in V(D)$ it thus yields a tree $T_1$ such that $u \in V(T_1)$ if and only if there is a directed $u$—$v$-path in $D$. If we modify Step 1 to instead select $u$ and $w$ such that $wu \in A(D)$, the algorithm constructs a maximal tree in which all arcs are directed towards the root. Started from the same vertex $v$ it thus constructs a tree $T_2$ such that $u \in V(T_2)$ if and only if there is a directed $v$—$u$-path in $D$. The set of vertices of the strongly connected component of $D$ containing $v$ then is the set $V(T_1) \cap V(T_2)$, and the strongly connected component itself the subgraph of $G$ induced by the set of vertices. To determine any other strongly connected components of $D$ we can repeat the procedure for a vertex not contained in a connected component found so far, until no such vertex exists.

**Example 4.8.** If we apply the procedure to the digraph $D$ in Figure 2.2 starting from vertex 1, using breadth-first search, and giving precedence to vertices with smaller labels, we obtain the following two trees:

The connected component $D_1$ containing vertex 1 is thus given by $V(D_1) = V(T_1) \cap V(T_2) = \{1, 2, 5\}$ and $A(D_1) = \{(1, 2), (2, 5), (5, 1)\}$. We can now apply the procedure starting from vertex $3 \notin V(D_1)$ to obtain the connected component $D_2$ with $V(D_2) = \{3\}$ and $A(D_2) = \emptyset$, and then starting from vertex 4 to obtain the connected component $D_3$ with $V(D_3) = \{4\}$ and $A(D_3) = \emptyset$.

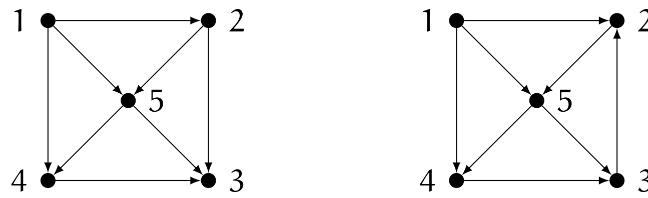



Figure 4.2: A directed acyclic graph and a directed graph that contains a directed cycle

## 4.5 Directed Cycles

We finally consider directed cycles in directed graphs.

**Definition 4.9** (directed acyclic graph). A digraph $D$ is a *directed acyclic graph* if it does not contain any directed cycles.

The digraph on the left of Figure 4.2 is a directed acyclic graph. The digraph on the right of Figure 4.2 contains the cycle $2, 5, 3, 2$.

A useful concept to understand directed cycles in digraphs is a so-called topological ordering.

**Definition 4.10** (topological ordering). Let $D$ be a digraph. A *topological ordering* of $D$ is a strict total order $\prec$ on $V(D)$ such that $u \prec v$ for every arc $uv \in A(D)$.

In other words, a topological ordering of a digraph is an ordering of its vertices from left to right such that all arcs also go from left to right. We will see that directed acyclic graphs are characterized by the existence of a topological ordering. One direction of this result is obvious: if there is a topological ordering and we draw the vertices of the graph from left to right according to this ordering, then all arcs go from left to right and there cannot be any cycles. We show the other direction by giving an algorithm that takes a digraph $D$ and determines a topological ordering of $D$ whenever such a topological ordering exists.

**Algorithm 4.11** (topological ordering). Let $D$ be a digraph. Start with the empty sequence, then repeat the following steps until no vertices remain in $D$:

1. Let $v \in V(D)$ such that $d^-_D(v) = 0$. If no such vertex exists, then stop. There is no topological ordering of the original digraph.
2. Write down $v$ as the next element in the sequence.
3. Remove $v$ from $D$, along with all arcs incident to $v$.

For a given digraph $D$, the algorithm either stops at some point in Step 1 or produces a sequence that contains each element of $V(D)$ exactly once. In the latter case the sequence produced actually describes a topological ordering of $D$: when $v \in V(D)$ is written down, there are no arcs in the remaining digraph whose head is $v$, and the remaining digraph in particular contains all arcs from the original digraph whose tail comes after $v$ in the sequence.

**Example 4.12.** When the algorithm is applied to the digraph on the left of Figure 4.2, which is acyclic, it produces the sequence $1, 2, 5, 4, 3$. It is easy to verify that this sequence describes a topological ordering. When the algorithm is applied to the digraph on the right of Figure 4.2, which contains a cycle, it writes down 1 and stops, concluding that the digraph does not have a topological ordering.

The algorithm produces a topological ordering for $D$ if and only if $D$ is acyclic. This is a consequence of the following lemma, and the fact that a digraph is acyclic if and only if all of its subdigraphs are acyclic.

**Lemma 4.13.** Let $D$ be a directed acyclic graph. Then there exists $v \in V(D)$ such that $d^-_D(v) = 0$.

*Proof.* Consider a directed path $P$ of of maximum length in $D$, and let $v$ be the first vertex on this path. Then no arcs exist from any vertex in $V(D) \setminus V(P)$ to $v$, because this would contradict the fact that $P$ has maximum length. There are also no arcs from any vertex in $V(P)$ to $v$, because this would imply the existence of a directed cycle. Thus $d^-_D(v) = 0$. □

We have established the following result.

**Theorem 4.14.** Let $D$ be a digraph. Then $D$ is acyclic if and only if it has a topological ordering.

Algorithm 4.11 can thus be used to test whether a given digraph is acyclic, and produces a topological ordering if it is. The running time of the algorithm is $O(|V(D)|^3)$: it removes an element of $V(D)$ in each iteration in which it does not stop; a vertex with indegree zero can be found, and then deleted along with any incident edges, using a constant number of basic operations like comparison or copying for each of the at most $|V(D)|^2$ entries of the adjacency matrix.

**Example 4.15.** Assume we are given a set of projects, along with dependencies among projects that require some projects to be completed before others can be started. This situation can be represented by a digraph in which each

vertex corresponds to a project and there is an arc $uv$ if project $u$ has to be completed before project $v$ can begin. Clearly the projects can only be completed if the digraph is acyclic, because any cycle would mean that none of the projects in the cycle can ever be started. In the case where the graph is acyclic, Algorithm 4.11 produces an ordering of the projects in which they can be executed.

# Chapter 5

# Minimum Spanning Trees and Shortest Paths in Networks

We have already looked at spanning trees and paths that were optimal in the sense that they contained a minimum number of edges. We will now consider optimal spanning trees and paths in networks, where we will seek to minimize the sum of the weights of the edges or arcs in the spanning tree or path.

**Definition 5.1** (minimum spanning tree, shortest path). Consider a network $(G, w)$, and let the weight or length of a subgraph $H$ of $G$ be the sum $\sum_{e \in E(H)} w(e)$ of the weights of its edges. A spanning tree $T$ of $G$ is a *minimum spanning tree* of $G$ if it has minimum weight among all spanning trees of $G$. A $u$–$v$-path of $G$ is a *shortest* $u$–$v$*-path* in $G$ if it has minimum length among all $u$–$v$-paths in $G$.

It will be convenient to discuss these problems only for simple graphs. This is without loss of generality, as spanning trees and paths may not contain any loops and will never contain any multiple edges. Each of the algorithms we will describe for simple graphs can thus be applied to general graphs by first removing all loops and removing all multiple edges with a given pair of endpoints except one with minimum weight. This preliminary step has running time $O(|E(G)|)$.

Figure 5.1 shows a network along with a minimum spanning tree and a spanning tree of shortest paths.

## 5.1 Minimum Spanning Trees

We will consider three different algorithms for finding a minimum spanning tree. All three algorithms follow what could be called a *greedy* strategy: they build a solution in small steps and in each step take a decision that myopically optimizes a certain criterion without looking into the future. We will begin by describing the algorithms and convincing ourselves that they all produce a spanning tree

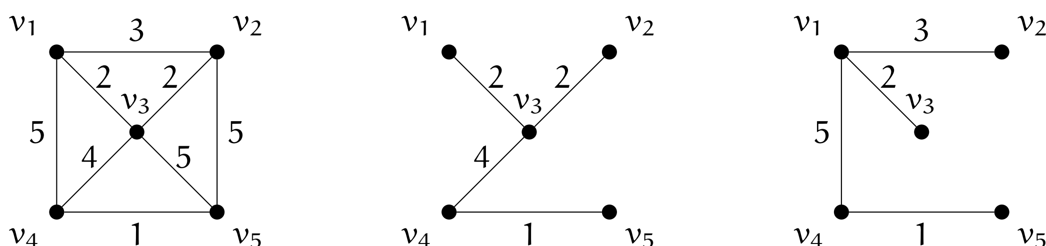



Figure 5.1: A network, a minimum spanning tree of the network, and a spanning tree of the network containing for each $i \in [5]$ a shortest $1-v_i$-path

and do so efficiently. Then we will argue, using two results concerning minimum spanning trees and the edges contained within them, that the algorithms actually produce a minimum spanning tree.

The first algorithm, named after Robert Prim but actually proposed earlier by Vojtěch Jarník, starts from an arbitrary vertex and grows a tree by repeatedly adding an edge of minimum weight between a vertex within the tree and a vertex outside it.

**Algorithm 5.2** (Prim). Consider a network $(G, w)$ and $s \in V(G)$. *Prim's algorithm* starts from the tree $T$ with $V(T) = \{s\}$ and $E(T) = \emptyset$, and then repeats the following:

1. Let $F = \{uv \in E(G) : u \in V(T), v \in V(G) \setminus V(T)\}$. If $F = \emptyset$, stop.
2. Let $uv \in F$ such that $w(u, v) = \min_{xy \in F} w(x, y)$.
3. Add $v$ to $V(T)$ and $uv$ to $E(T)$.

It is easy to see that the algorithm produces a spanning tree of the connected component containing the initial vertex $s$. Indeed, an edge is added in each iteration where the algorithm doesn't stop, an edge considered for inclusion never forms a cycle with the edges already present, and some edge is considered for inclusion until a spanning tree of the connected component has been found. The algorithm in fact stops after at most $|V(G)| - 1$ iterations, equal to the number of edges in a spanning tree, and edge $uv$ in Step 2 can be found in time $O(|E(G)|)$. The algorithm thus has running time $O(|V(G)| \cdot |E(G)|)$, and this can be reduced to $O(|V(G)|^2)$ by storing for each vertex outside the current tree the minimum weight of an edge connecting it to the tree. Using more sophisticated ideas it is possible to achieve a running time of $O(|E(G)| + |V(G)| \log |V(G)|)$, which is better in graphs that have relatively few edges.

**Example 5.3.** When Prim's algorithm is applied to the network on the left of Figure 5.1 starting from $v_1$, it successively adds the edges $v_1v_3$, $v_2v_3$, $v_3v_4$, and $v_4v_5$ along with their vertices. It thus obtains the spanning tree at the

center of Figure 5.1.

The second algorithm, discovered by Joseph Kruskal, also starts from a graph without any edges, and repeatedly adds edges with the smallest possible weight while ensuring that the graph remains acyclic.

**Algorithm 5.4 (Kruskal).** Consider a network $(G, w)$. Let $F$ be a sequence that contains the elements of $E(G)$ in non-decreasing order of weight. *Kruskal's algorithm* starts from the graph $T$ with $V(T) = V(G)$ and $E(T) = \emptyset$, and then repeats the following until $F$ is empty:

1. Let $uv$ be the first element of $F$. Remove $uv$ from $F$.
2. Unless this would creates a cycle, add $uv$ to $E(T)$.

It is again easy to see that the algorithm finds a spanning tree after a finite number of iterations, because it considers each edge exactly once and never creates a cycle. An important factor in the overall running time of the algorithm is the time it takes to sort the set of edges by weight. It is easy to see that this part of the algorithm can be completed in $O(|E(G)|^2)$ steps, and in fact it can be done in $O(|E(G)| \log |E(G)|)$ steps. The algorithm then runs for $|E(G)|$ iterations, and the difficult question in each iteration is whether the addition of an edge would create a cycle. We have seen how this can be done using tree search in time $O(|V(G)| \cdot |E(G)|)$, and it can in fact be done in time $O(|V(G)|)$ by remembering in a sophisticated way the connected component each vertex is part of. An edge then creates a cycle if it connects two vertices that are already in the same connected component.

**Example 5.5.** Again consider the network on the left of Figure 5.1 and observe that the following ordering of its edges is non-decreasing in weight:

$$v_4v_5, v_1v_3, v_2v_3, v_1v_2, v_3v_4, v_1v_4, v_2v_5, v_3v_5$$

Kruskal's algorithm thus begins by adding edges $v_4v_5$, $v_1v_3$, and $v_2v_3$. The edge $v_1v_2$ is considered next, but this edge forms a cycle with the edges $v_1v_3$ and $v_2v_3$ already present and is therefore not added. The algorithm then adds the edge $v_3v_4$ to again obtain the spanning tree at the center of Figure 5.1. As any additional edge would create a cycle, no further edges are added.

The third algorithm, which can be seen as an inverse version of Kruskal's, starts with all edges that are in the original graph and repeatedly deletes vertices with maximum weight while ensuring that the graph stays connected.

**Algorithm 5.6 (reverse-delete).** Consider a network $(G, w)$. Let $F$ be a se-

quence that contains the elements of $E(G)$ in non-decreasing order of weight. The *reverse-delete algorithm* starts from the graph $T$ with $V(T) = V(G)$ and $E(T) = E(G)$, and then repeats the following until $F$ is empty:

1. Let $uv$ be the last element of $F$. Remove $uv$ from $F$.
2. Unless this would yield a graph that is not connected, remove $uv$ from $E(T)$.

It is yet again easy to see that the algorithm finds a spanning tree after a finite number of iterations. Like Kruskal's algorithm the algorithm starts by sorting the set of edges and then runs for $|E(G)|$ iterations. The difficult question in each iteration, whether the removal of an edge would result in a graph that is not connected, can be be answered using tree search in time $O(|V(G)| \cdot |E(G)|)$.

**Example 5.7.** Consider the network on the left of Figure 5.1 one more time. The reverse-delete algorithm uses the same ordering of the edges as Kruskal's, but instead of adding edges to an empty network starting at the front of the ordering removes edges from the original network starting at the back. It thus removes edges $v_3v_5$, $v_2v_5$, and $v_1v_4$. The edge $v_3v_4$ is considered next, but is the only remaining path between vertices $v_3$ and $v_4$ and therefore not removed. The algorithm then removes the edge $v_1v_2$ and yet again obtains the spanning tree at the center of Figure 5.1. No further edges are removed, as any such removal would yield a graph that is not connected.

To show that Prim's and Kruskal's algorithms work correctly, we will use a more general result characterizing the edges contained in some or all minimum spanning trees.

**Theorem 5.8.** Consider a connected network $(G, w)$. Let $S \subseteq V(G)$, $F = \{uv \in E(G) : u \in S, v \in V(G) \setminus S\}$, and $uv \in F$. If $w(u, v) = \min_{st \in F} w(s, t)$, then there is a minimum spanning tree of $(G, w)$ containing $uv$. If $w(u, v) < \min_{st \in F \setminus \{uv\}} w(s, t)$, then every minimum spanning tree of $(G, w)$ contains $uv$.

*Proof.* Let $T$ be an arbitrary minimum spanning tree of $(G, w)$. If $uv \in E(T)$, there is nothing to show. If $uv \notin E(T)$ then, by Theorem 2.17, $E(T) \cup \{uv\}$ contains a cycle, and this cycle must contain another edge $xy \in F$. Let $T'$ be the graph with $V(T') = V(T)$ and $E(T') = (E(T) \setminus \{xy\}) \cup \{uv\}$. Observe that $T'$ is connected and acyclic, and thus is a spanning tree of $G$. Now consider the weight of $uv$. If $w(u, v) = \min_{st \in F \setminus \{uv\}} w(s, t)$, then $T'$ has the same weight as $T$ and is a minimum spanning tree. There thus exists a minimum spanning tree that contains $uv$. If $w(u, v) < \min_{st \in F \setminus \{uv\}} w(s, t)$, then $T'$ has a smaller weight than $T$, which contradicts the assumption that $T$ is a minimum spanning tree.

Thus $uv \in E(T)$ and, since $T$ is an arbitrary minimum spanning tree, $uv$ must be contained in every minimum spanning tree. □

**Theorem 5.9.** When applied to a connected network, Prim's algorithm produces a minimum spanning tree.

*Proof.* Any edge $e$ added by the algorithm has minimum weight in $F = \{uv \in E(G) : u \in V(T), v \in V(G) \setminus V(T)\}$, which is equal to the set $F$ in Theorem 5.8 if we choose $S = V(T)$.

If $e$ is the unique edge with minimum weight in $F$, then by Theorem 5.8 it is contained in every minimum spanning tree and can safely be added. To see that it is safe to add $e$ also if it is not the unique edge with minimum weight, imagine that we modify the network by adding some small value $\epsilon > 0$ to the weight of every edge of minimum weight in $F \setminus \{e\}$. As we only increase the weight of edges the algorithm has not yet selected, the algorithm would have made the same choices for the original and the modified network up to the point when it considers $e$. Since $e$ is now the unique edge with minimum weight in $F$, we can add it to the spanning tree and proceed with the algorithm to obtain a minimum spanning tree of the modified network. However, if $\epsilon$ is chosen to be small enough, then any spanning tree that was not a minimum spanning tree of the original network will also not be a minimum spanning tree of the modified network. The spanning tree found by the algorithm for the modified network will thus be a minimum spanning tree of the original network. □

Correctness of Kruskal's algorithm can be shown in analogous way to that of Prim's algorithm, and we leave this as an exercise.

**Theorem 5.10.** When applied to a connected network, Kruskal's algorithm produces a minimum spanning tree.

To prove that the reverse-delete algorithm works correctly, we use a result characterizing the edges not contained in all or any minimum spanning trees.

**Theorem 5.11.** Consider a connected network $(G, w)$. Let $F \subseteq E(G)$ be the set of edges contained in a cycle of $G$, and let $uv \in F$. If $w(u, v) = \max_{st \in F} w(s, t)$, then there is a minimum spanning tree of $(G, w)$ that does not contain $uv$. If $w(u, v) > \max_{st \in F \setminus \{uv\}} w(s, t)$, then no minimum spanning tree of $(G, w)$ contains $uv$.

*Proof.* Let $T$ be an arbitrary minimum spanning tree of $(G, w)$. If $uv \notin E(T)$, there is nothing to show. If $uv \in E(T)$, then there exists another edge $xy \in F$ such that $xy \notin E(T)$. Let $T'$ be the graph with $V(T') = V(T)$ and $E(T') = (E(T) \setminus \{uv\}) \cup \{xy\}$. Observe that $T'$ is connected and acyclic, and thus is a spanning tree of $G$.

Now consider the weight of $uv$. If $w(u,v) = \max_{st \in F \setminus \{uv\}} w(s,t)$, then $T'$ has the same weight as $T$ and is a minimum spanning tree. There thus exists a minimum spanning tree that does not contain $uv$. If $w(u,v) > \max_{st \in F \setminus \{uv\}} w(s,t)$, then $T'$ has a smaller weight than $T$, which contradicts the assumption that $T$ is a minimum spanning tree. Thus $uv \notin E(T)$ and, since $T$ is an arbitrary minimum spanning tree, $uv$ must not not be contained in any minimum spanning tree. □

Correctness of the reverse-delete algorithm can be shown in an analogous way to that of Prim's algorithm, by using Theorem 5.11 to argue that it is safe to remove each edge removed by the algorithm. We leave the proof as an exercise.

**Theorem 5.12.** When applied to a connected network, the reverse-delete algorithm produces a minimum spanning tree.

Theorems 5.8 and 5.11 could also be used more generally to reason about minimum spanning trees of a given network. We can show, for example, that the spanning tree at the center of Figure 5.1 is the unique minimum spanning tree of the network on the left by arguing that every edge contained in this spanning tree must be contained in every minimum spanning tree, or that none of the edges not contained in it are contained in any minimum spanning tree. However, this seems difficult to do because Theorems 5.8 and 5.11 do not tell us how to find a partition of the set of vertices or a cycle for which their conditions are satisfied.

The following result provides a necessary and sufficient condition for uniqueness that is easier to check. We leave the proof of this result as an exercise.

**Theorem 5.13.** Let $(G,w)$ be a network, $T$ a minimum spanning tree of $(G,w)$. Then $(G,w)$ has a unique minimum spanning tree if and only if the following condition is satisfied: for every edge $e \in E(G) \setminus E(T)$ with endpoints $u, v \in V(G)$ and every edge $d \in E(T)$ contained in the unique $u$–$v$-path in $T$, $w(e) > w(d)$.

## 5.2 Shortest Paths for Non-Negative Weights

Let us now turn to shortest paths. In the case where all weights are non-negative, shortest paths starting at a given vertex can be found by growing a spanning tree in a similar way as Prim's algorithm, but using a different greedy choice for the next edge to include. The algorithm was first proposed by Edsger Dijkstra.

**Algorithm 5.14 (Dijkstra).** Consider a network $(G,w)$ and $s \in V(G)$. *Dijkstra's algorithm* starts from the tree $T$ with $V(T) = \{s\}$ and $E(T) = \emptyset$, and

then repeats the following:

1. Let $F = \{uv \in E(G) : u \in V(T), v \in V(G) \setminus V(T)\}$. If $F = \emptyset$, stop.
2. For each $u \in V(T)$, let $\delta(u)$ be the length of the unique $s$—$u$-path in $T$.
3. Let $uv \in F$ such that $\delta(u) + w(u, v) = \min_{xy \in F}(\delta(x) + w(x, y))$.
4. Add $v$ to $V(T)$ and $uv$ to $E(T)$.

We will see that Dijkstra's algorithm indeed grows a tree containing shortest paths from $s$ to the other vertices. The value $\delta(u)$ defined in Step 2 is thus the distance of $u$ from $s$, i.e., the length of a shortest $s$—$u$ path in $(G, w)$. The algorithm then in each step greedily connects to the current tree a vertex $v$ outside the tree that is closest to $s$, where the distance from $s$ to $v$ is measured as the length of any shortest path inside the tree plus that of the single edge extending that path to $v$. As in the case of Prim's algorithm, it is easy to see that the running time of Dijkstra's algorithm is $O(|V(G)| \cdot |E(G)|)$. Indeed, the algorithm runs for at most $|V(G)| - 1$ iterations, and an edge $uv$ in Step 3 can be found in time $O(|E(G)|)$ assuming that we have stored $\delta(u)$ for all $u \in V(T)$. By storing for every vertex $v \in V(G)$ the length of a shortest $s$—$v$-path that apart from $v$ only includes vertices in the current tree $T$, the running time of Step 3 can be reduced to $O(|V(G)|)$ and that of the algorithm to $O(|V(G)|^2)$. A further improvement to $O(|E(G)| + |V(G)| \log |V(G)|)$ can be achieved using more sophisticated ideas.

**Example 5.15.** Consider the network in the left of Figure 5.1 and assume that Dijkstra's algorithm is applied to this network starting from $v_1$. At the beginning of the algorithm, $V(T) = \{v_1\}$ and $\delta(v_1) = 0$, and the edges considered for inclusion are $v_1v_2$, $v_1v_3$, and $v_1v_4$. Of these $v_1v_3$ is selected, as $\delta(v_1)+w(v_1, v_3) = \min\{\delta(v_1)+w(v_1, v_2), \delta(v_1)+w(v_1, v_3), \delta(v_1)+w(v_1, v_4)\}$. Now $V(T) = \{v_1, v_3\}$, $\delta(v_1) = 0$, and $\delta(v_3) = 2$, and the edges considered for inclusion are $v_1v_2$, $v_1v_4$, $v_3v_2$, $v_3v_4$, and $v_3v_5$. Of these $v_1v_2$ is selected, as $\delta(v_1) + w(v_1, v_2) = \min\{\delta(v_1) + w(v_1, v_2), \delta(v_1) + w(v_1, v_4), \delta(v_3) + w(v_3, v_2), \delta(v_3) + w(v_3, v_4), \delta(v_3) + w(v_3, v_5)\}$. The algorithm then proceeds to include edges $v_1v_4$ and $v_4v_5$ and obtains the spanning tree on the right of Figure 5.1.

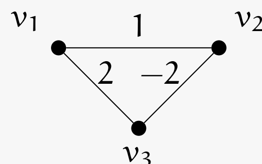
**Theorem 5.16.** When applied to a network $(G, w)$ with non-negative weights starting from $s \in V(G)$, Dijkstra's algorithm produces a tree containing shortest paths from $s$ to all vertices in the same connected component as $s$.

*Proof.* We show by induction on $|V(T)|$ that throughout the algorithm, $T$ contains a shortest $s$—$t$-path for every $t \in V(T)$. Initially $V(T) = \{s\}$, and the claim trivially holds. Now assume that the algorithm has constructed a tree $T$ containing a shortest $s$—$t$-path for every $t \in V(T)$, and selects the vertex $v$ and the edge $uv$ for

addition to the tree. Let $P$ be the $s$—$v$-path that follows the $s$—$u$-path contained in $T$ and then traverses the edge $uv$. For any path $Q$, let $l(Q)$ denote the length of $Q$. It suffices to show that $P$ is a shortest $s$—$v$-path. Assume for contradiction that this is not the case, i.e., that there exists an $s$—$v$-path $Q$ that is shorter than $P$. Since $v \notin V(T)$, $Q$ must contain an edge in $F$. Let $xy$ be the first edge of $Q$ contained in $F$, and let $R$ be the $s$—$x$-path obtained by following $Q$ to $x$. Since weights are non-negative, $l(Q) \geqslant l(R) + w(x, y)$. Since the algorithm chooses $uv \in F$ over $xy \in F$, $\delta(x) + w(x, y) \geqslant \delta(u) + w(u, v)$. Since $u, x \in V(T)$, and by the induction hypothesis, $l(R) \geqslant \delta(x)$ and $l(P) = \delta(u) + w(u, v)$. Thus $l(Q) \geqslant l(P)$, contradicting the assumption that $Q$ is shorter than $P$. □

When some of the weights in the network are negative, Dijkstra's algorithm may fail.

**Example 5.17.** Consider the following network:




When Dijkstra's algorithm is applied to this network starting from vertex $v_1$, it considers the edges $v_1v_2$ and $v_1v_3$ for inclusion and includes $v_1v_2$. This is incorrect, as the shortest $v_1$—$v_2$-path is not $v_1, v_2$ but $v_1, v_3, v_2$.

Shortest paths can be found efficiently even in the presence of negative weights as long as there are no negative cycles, but this is beyond the scope of this module. In the presence of negative cycles, finding shortest paths is NP-hard.

## 5.3 Shortest Directed Paths

In a directed network with non-negative weights, a version of Dijkstra's algorithm that in Step 1 considers arcs from $u$ to $v$ instead of edges between $u$ and $v$ can be used to find a shortest directed path from $s$ to any vertex to which such a path exists. A variant of Example 5.17 with arcs $v_1v_2$, $v_1v_3$, and $v_2v_3$ shows that the algorithm may again fail in the presence of negative weights. The reason for this failure is that Dijkstra's algorithm greedily commits to a path while ignoring that there might be a path with a greater number of arcs but shorter overall length. An alternative algorithm, due to Bellman and Ford, overcomes this problem by considering paths in increasing order of their number of arcs and looking specifically for shorter paths that have a greater number of arcs.

Consider a directed network $(D, w)$ and $s \in V(D)$. Let $n = |V(D)|$. For $v \in V(D)$ and $k = 0, 1, \ldots, n-1$, let $\delta_k(v)$ be the length of a shortest directed

$s$—$v$-path that uses at most $k$ arcs. In the absence of negative cycles, the length of a shortest $s$—$v$-path is equal to $\delta_{n-1}(v)$. The idea behind Bellman and Ford's algorithm is to express $\delta_k(v)$ recursively in terms of the values $\delta_{k-1}(u)$ for $u \in V(D)$ and to then computing the values inductively for $k = 0, \dots, n-1$. Clearly, an $s$—$v$-path with $k = 0$ arcs exists if an only if $v = s$. Defining the length of non-existent paths to be $\infty$, we thus have

$$\delta_0(v) = \begin{cases} 0 & \text{if } v = s, \\ \infty & \text{otherwise.} \end{cases} \tag{5.1}$$

When $k \geqslant 1$, a shortest $s$—$v$-path using $k$ arcs can have one of two forms: either it uses only $k-1$ arcs, or it uses $k-1$ arcs to get from $s$ to some vertex $u \in V(D)\setminus\{v\}$ and finally traverses the arc $uv$. Thus, for $k \geqslant 1$,

$$\delta_k(v) = \min\{\delta_{k-1}(v), \min_{u \in V(D)\setminus\{v\}} \delta_{k-1}(u) + w(u, v)\}. \tag{5.2}$$

Algorithm 5.18 below computes the lengths of shortest $s$—$v$-paths for all $v \in V(D)$ by initializing them according to (5.1) and then repeatedly updating them according to (5.2), subject to two additions.

First, the algorithm performs an additional $n$th round of updates to compute $\delta_n(v)$ for all $v \in V(D)$, and any further reduction in this final iteration indicates the existence of a directed cycle of negative length that invalidates the result of the algorithm. Second, the algorithm stores for each vertex $v \in V(D)$ a predecessor $p(v) \in V(D)$ that appears immediately before $v$ on a shortest $s$—$v$-path of length $\delta_k(v)$. This predecessor is updated whenever a shorter path is found, i.e., when the outer minimum in (5.2) is not attained by the first value. Denote by $U \subseteq V(D)$ the set of vertices $v$ such that $D$ contains a directed $s$—$v$-path. When the algorithm terminates, every vertex $v \in U$ will have a predecessor $p(v)$, and $p(v)v \in A(D)$. If the algorithm fails due to the existence of a directed cycle of negative length, the set of arcs $\{p(v)v \mid v \in U\}$ will contain a negative cycle. Otherwise these arcs will form a spanning tree of the induced subdigraph $D[U]$ of $D$ and will all be directed away from $v$. This is the equivalent for directed networks of the spanning tree of shortest paths of the connected component containing $s$ that is found by Dijkstra's algorithm.

**Algorithm 5.18** (Bellman-Ford). Consider a directed network $(D, w)$ and $s \in V(D)$. Let $n = |V(D)|$. The *Bellman-Ford algorithm* proceeds as follows:

1. For each $v \in V(D)$, set $\delta_0(v) = 0$ if $v = s$ and $\delta_0(v) = \infty$ otherwise.
2. Repeat the following for $k = 1, 2, \dots, n-1$:
   For each $v \in V(D)$ do the following:
   (a) Let $\delta_k(v) = \min\{\delta_{k-1}(v), \min_{u \in V(D)\setminus\{v\}} \delta_{k-1}(u) + w(u, v)\}$.
   (b) If $\delta_k(v) < \delta_{k-1}(v)$, let $p(v) = \arg\min_{u \in V(D)\setminus\{v\}} \delta_{k-1}(u) + w(u, v)$.
3. Repeat the previous step for $k = n$. If for any $v \in V(D)$, $\delta_n(v) <$

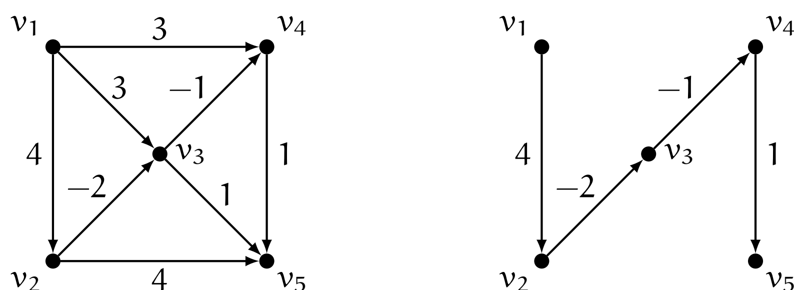



Figure 5.2: A directed network $(D, w)$ and a spanning tree of this network containing a shortest $v_1$—$v$-path of $(D, w)$ for every vertex $v \in V(D)$

$\delta_{n-1}(v)$, then stop. $(D, w)$ contains a directed cycle of negative length and the algorithm cannot be used to find shortest paths in $(D, w)$.

**Example 5.19.** Consider the directed network on the left of Figure 5.2, and assume that the Bellman-Ford algorithm is applied to the network starting from $v_1$.

When computing $\delta_k(v)$ for $k = 1, 2, \ldots, n$ and $v \in V(G)$ it is convenient to write down these values in a table with rows indexed by values of $k$ and columns indexed by values of $v$. As in addition to determining the length $\delta(v)$ of a shortest $v_1$—$v$-path we want to find the path itself, we also record the predecessor $p(v)$ of $v$ along a shortest path that uses at most $k$ vertices alongside the value $\delta_k(v)$. For the network in Figure 5.2 we obtain the following table.

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |
|---|---|---|---|---|---|
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 1 | 0 | $4(v_1)$ | $3(v_1)$ | $3(v_1)$ | $\infty$ |
| 2 | 0 | $4(v_1)$ | $2(v_2)$ | $2(v_3)$ | $4(v_3)$ |
| 3 | 0 | $4(v_1)$ | $2(v_2)$ | $1(v_3)$ | $3(v_3)$ |
| 4 | 0 | $4(v_1)$ | $2(v_2)$ | $1(v_3)$ | $2(v_4)$ |
| 5 | 0 | $4(v_1)$ | $2(v_2)$ | $1(v_3)$ | $2(v_4)$ |

The entry for $k = 1$ and $v = v_2$ has for example been obtained from entries in the row for $k = 0$ as $\delta_1(v_2) = \min\{\delta_0(v_2), \min_{u \in V(D)\setminus\{v_2\}} \delta_0(u) + w(u, v_2)\} = \min\{\infty, \delta_0(v_1) + w(v_1, v_2)\} = \min\{\infty, 0 + 4\} = 4$, where the second equality holds because $v_1$ is the only vertex that is the tail of an arc with head $v_2$. The vertex $v_1$ from which $v_2$ has been reached has been recorded in the table next to $\delta_1(v_2)$.

Since the entries in the row for $k = 5$ are unchanged compared to those in the row for $k = 4$, the algorithm has not encountered a cycle of negative

length and the values it has computed are therefore correct. The arcs $v_1v_2$, $v_2v_3$, $v_3v_4$, and $v_4v_5$ between each vertex and its predecessor on a shortest path form a spanning tree of the network $(D, w)$, and this spanning tree contains a shortest $v_1$–$v$-path for every $v \in V(D)$. The spanning tree, which in this particular example happens to be a path, is shown on the right of Figure 5.2.

We have already argued that the algorithm is correct. To see that it is efficient, observe that the algorithm performs $|V(G)|$ iterations. In each iteration $|V(G)|$ values are updated, and each such update considers a minimum over $|V(G)|$ values. All other operations can be performed in constant time. The running time of the algorithm is thus $O(|V(G)|^3)$. As it is unnecessary to consider a particular arc more than once in each iteration, it is possible to reduce the running time to $O(|V(G)| \cdot |A(G)|)$.

It is natural to ask whether an efficient algorithm exists that can find shortest paths even in the presence of negative cycles. This is unlikely, as the problem of finding shortest paths in this case is NP-hard.

## 5.4 Directed Cycles

If a directed network $(D, w)$ contains a directed cycle of negative length, and a directed path from $s \in V(D)$ to a vertex in that cycle, the Bellman-Ford algorithm cannot be used to find shortest directed paths in $(D, w)$ that start at $s$. The algorithm will, however, detect the presence of such a cycle. We can thus use the algorithm to detect a negative cycle in an arbitrary directed network $(D, w)$, by first adding to the digraph a new vertex $u \notin V(D)$ and arcs $\{uv : v \in V(D)\}$ with weight zero and then running the Bellman-Ford algorithm starting from $u$.

**Example 5.20.** Imagine that we are given a table of exchange rates among currencies and want to decide whether there exists an arbitrage opportunity, i.e., an opportunity to make a profit by a cyclic exchange of currencies. We can represent this situation by a digraph on the set of currencies where the arc from currency $u$ to currency $v$ has a weight equal to the negative of the logarithm of the exchange rate from $u$ to $v$. The reason we want to the use the negative of the logarithm as the weight is that an arbitrage opportunity corresponds to a directed cycle in the graph such that the product of the exchange rates along the cycle is greater than one, which is the case exactly if the negative of the sum of the logarithms of the exchange rates is negative.

## 5.5 Longest Paths in Directed Acyclic Networks

The Bellman-Ford algorithm can finally be used to find a longest directed path in a directed acyclic network. To this end we again add a new vertex and arcs of weight zero from the new vertex to any existing vertex, and then multiply all weights by $-1$. Since the network does not contain any directed cycles, starting the Bellman-Ford algorithm from the new vertex will produce a shortest path to any other vertex to which a path exists, and the shortest such path corresponds to a longest path in the original network. Just like the problem of finding a shortest path in the presence of a negative cycle, the problem of finding a longest directed path in a general directed network is NP-hard.

Longest directed paths in a directed acyclic network can in fact be found more easily by a greedy algorithm that like Dijkstra's algorithm grows a spanning tree $T$ starting from an initial vertex $s \in V(D)$, and uses the same notion of distance between $s$ and vertices $v \in V(D) \setminus V(T)$ as Dijkstra's algorithm. Unlike Dijkstra's algorithm, the new algorithm selects an arc for addition to $T$ to maximize rather than minimize distance, and adds vertices to $V(T)$ in the order of a topological ordering. When $v \in V(D) \setminus V(T)$ is added to $V(T)$, the latter property ensures that there cannot be any longer paths that visit vertices in $(V(D) \setminus V(T)) \setminus \{v\}$ before visiting $v$, as no arcs exist in $D$ from such vertices to $v$.

**Algorithm 5.21** (Morávek). Consider a directed acyclic network $(D, w)$, and $s \in V(D)$ such that there exists a directed $s$–$u$-path in $D$ for every $u \in V(D)$. Let $\prec$ be a topological ordering of $D$. *Morávek's algorithm* starts from the tree $T$ with $V(T) = \{s\}$ and $A(T) = \emptyset$ and then repeats the following:

1. Let $v \in V(D) \setminus V(T)$ such that $v \prec u$ for all $u \in V(D) \setminus V(T)$. If no such element exists, then stop.
2. Let $F = \{uv \in A(D) : u \in V(T)\}$.
3. For each $u \in V(T)$, let $\delta(u)$ be the length of the unique $s$–$u$-path in $T$.
4. Let $uv \in F$ such that $\delta(u) + w(u, v) = \max_{xy \in F} \delta(x) + w(x, y)$.
5. Add $v$ to $V(T)$ and $uv$ to $A(T)$.

We have already argued informally why the algorithm is correct and leave a formal proof of correctness as an exercise. The running time of the algorithm, including the construction of a topological orderging, is easily seen to be $O(|V(D)| \cdot |A(D)|)$, and this can be improved to $O(|A(D)|)$ by using that both the construction of a topological ordering and the rest of the algorithm can be completed by considering each arc only a constant number of times. It further is straighforward to adapt the algorithm to find shortest rather than longest directed paths in directed acyclic networks with the same running time.

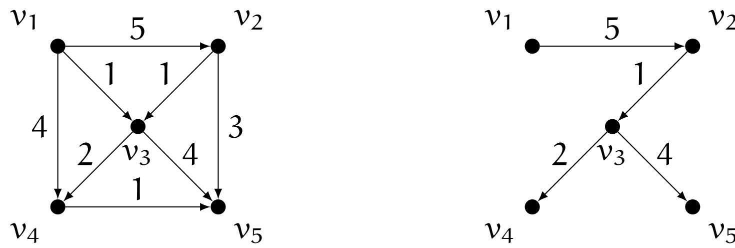



Figure 5.3: A directed acyclic network $(D, w)$ and a spanning tree of this network containing a longest $v_1$—$u$-path for every $u \in V(D)$

**Example 5.22.** Consider the directed acyclic network $(D, w)$ on the left of Figure 5.3. The unique topological ordering $\prec$ of $D$ is the one with $v_1 \prec v_2 \prec v_3 \prec v_4 \prec v_5$, and $D$ contains a $v_1$—$u$-path for every $u \in V(D)$. When Moravek's algorithm is applied to $(D, w)$ starting from $v_1$, it constructs the spanning tree on the right of Figure 5.3.

**Example 5.23.** Consider a directed acyclic network in which vertices and arcs represent milestones and activities in a project, and the weight of an arc corresponds to the duration of the activity represented by the arc. We may assume that there exists an initial milestone $s$ and a final milestone $t$ representing the start and end of the project, such that there exists a directed $s$—$v$-path and a directed $v$—$t$ path for every vertex $v$ in the network. A topological ordering of the network then represents a sequence in which the activities can be executed. A longest path, also called *critical path* in this context, represents a sequence of activities of maximum duration that have be to be executed sequentially and thus determine the overall duration of the project.

# Chapter 6

# Network Flows

Consider a directed network $(D, c)$, and imagine that certain quantities of a divisible resource such as water flow along the arcs of this network. The weight $c(e)$ of arc $e$, which we will assume to be non-negative, will now correspond to the *capacity* of that arc, i.e., the maximum amount of the resource that can flow along it. Given these capacities, and given two vertices $s, t \in V(D)$, we will ask how much of the resource we can send from $s$ to $t$. We will assume in the following that the network $(D, c)$ is simple, i.e., that it does not have any loops or multiple arcs. This is without loss of generality, as loops do not increase the amount of the resource we can send and multiple arcs can be replaced by a single arc with capacity equal to the sum of their capacities.

## 6.1 Maximum Flows

We need some notation. For $v \in V(D)$, let $A_D^-(v) \subseteq A(D)$ be the set of arcs in $D$ with head $v$ and $A_D^+(v) \subseteq A(D)$ the set of arcs with tail $v$. For $S \subseteq V(D)$, let $A_D^-(S) = \cup_{v \in S} A_D^-(v)$ be the set of arcs whose head is in $S$ and $A_D^+(S) = \cup_{v \in S} A_D^+(v)$ the set of arcs whose tail is in $S$.

**Definition 6.1** (flow, size, maximum flow). Let $(D, c)$ be a directed network, $s, t \in V(D)$. A function $f : A(D) \to \mathbb{R}$ is an $s$–$t$-*flow* of $(D, c)$ if

- $0 \leqslant f(e) \leqslant c(e)$ for all $e \in A(D)$, and
- $\sum_{e \in A_D^-(v)} f(e) = \sum_{e \in A_D^+(v)} f(e)$ for all $v \in V(D) \setminus \{s, t\}$.

The *size* of the $s$–$t$-flow $f$ is equal to $|f| = \sum_{e \in A_D^+(s)} f(e) - \sum_{e \in A_D^-(s)} f(e)$. An $s$–$t$-flow of $(D, c)$ is a *maximum* $s$–$t$-*flow* if it has maximum size among all $s$–$t$-flows of $(D, c)$.

The function $f$ specifies, for each arc $e$, the amount of the resource that is sent across $e$. The first property in the definition of a flow, which ensures that the flow on arc $e$ is between zero and the capacity of $e$, is also called a capacity constraint. The second property, which ensures that the amount of flow entering a vertex $v$

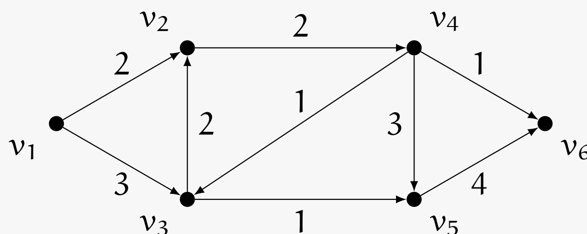
equals the amount of flow leaving that vertex, is also called flow conservation. Note that flow conservation is not imposed on $s$ and $t$. The flow conservation constrains imply, however, that any excess amount of the resource leaving $s$ has to be the same as the excess amount arriving at $t$, i.e., that $\sum_{e\in A_D^+(s)} f(e) - \sum_{e\in A_D^-(s)} f(e) = \sum_{e\in A_D^-(t)} f(e) - \sum_{e\in A_D^+(t)} f(e)$. Indeed,

$$\sum_{e\in A_D^+(s)} f(e) - \sum_{e\in A_D^-(s)} f(e) + \sum_{e\in A_D^+(t)} f(e) - \sum_{e\in A_D^-(t)} f(e) = \sum_{v\in V(D)} \Big( \sum_{e\in A_D^+(v)} f(e) - \sum_{e\in A_D^-(v)} f(e) \Big) = 0,$$

where the first equality holds because, by flow conservation, $\sum_{e\in A_D^-(v)} f(e) - \sum_{e\in A_D^+(v)} f(e) = 0$ for all $v \in V(D) \setminus \{s,t\}$, and the second equality because $f(e)$ appears exactly twice in the sum for each $e \in A(D)$, once with positive sign and once with negative sign. The excess amount leaving $s$ and arriving at $t$ is the size of the flow.

Since capacities are non-negative, a flow exists for every network, namely the flow that is equal to zero for every arc. That a maximum flow exists for every network is less obvious but also true. We will prove this later for the special case where all capacities are integers. For arbitrary non-negative capacities it can be shown by using that the set of flows is compact and the function that maps flows to their size is continuous.

**Example 6.2.** The following directed network represents the road network in the center of a small city, consisting of six roundabouts $v_1, \dots, v_6$ that are connected by a set of one-way streets. Cars can only enter the city center at $v_1$ and leave it at $v_6$, and for each road the maximum number of cars is given that could travel along this road per second.




Consider the function $f$ with

$$\begin{aligned} f(v_1v_2) &= 1, & f(v_1v_3) &= 1, & f(v_2v_4) &= 2, \\ f(v_3v_2) &= 1, & f(v_3v_5) &= 1, & f(v_4v_3) &= 1, \\ f(v_4v_5) &= 0, & f(v_4v_6) &= 1, & f(v_5v_6) &= 1. \end{aligned}$$

It is easily verified that $f$ is a $v_1$–$v_6$-flow of the network and that its size is 2.

However, $f$ is not a maximum $v_1$–$v_6$-flow, because the function $g$ with

$$\begin{aligned} g(v_1v_2) &= 2, & g(v_1v_3) &= 1, & g(v_2v_4) &= 2, \\ g(v_3v_2) &= 0, & g(v_3v_5) &= 1, & g(v_4v_3) &= 0, \\ g(v_4v_5) &= 2, & g(v_4v_6) &= 0, & g(v_5v_6) &= 3 \end{aligned}$$

is also a $v_1$–$v_6$-flow and has size 3.

## 6.2 Minimum Cuts

It was clear how we can show that a flow is *not* a maximum flow: we just give another flow with larger size. But how could we convince ourselves that a flow is a maximum flow? Consider the set $S = \{v_1, v_2, v_3\}$ and the set $A_D^+(S) \cap A_D^-(V(D) \setminus S) = \{v_2v_4, v_3v_5\}$ of arcs with tail in $S$ and head outside $S$. Intuitively, as $v_1 \in S$ and $v_6 \notin S$, any amount of flow that travels from $v_1$ to $v_6$ must at some point leave the set $S$, and it must do so by crossing either $v_2v_4$ or $v_3v_5$. This means that the size of any $v_1$–$v_6$-flow is bounded from above by $c(v_2v_4) + c(v_3v_5) = 3$. The $v_1$–$v_6$-flow $g$ has size 3 and must therefore be a maximum flow. The set $S$ is in this context called a $v_1$–$v_6$-cut because it separates $v_1$ from $v_6$, and the sum of the capacities of the arcs that cross the cut is referred to as the capacity of the cut.

**Definition 6.3** (cut, capacity, minimum cut)**.** Let $(D, c)$ be a directed network, $s, t \in V(D)$. A set $S \subseteq V(D)$ is an $s$–$t$-*cut* of $D$ if $s \in S$ and $t \notin S$. The *capacity* of an $s$–$t$-cut $S$ is equal to $C(S) = \sum_{e \in A_D^+(S) \cap A_D^-(V(D) \setminus S)} c(e)$. An $s$–$t$-cut of $(D, c)$ is a *minimum* $s$–$t$-*cut* if it has minimum capacity among all $s$–$t$-cuts of $(D, c)$.

We will now show that the size of $s$–$t$-flows is indeed bounded by the capacity of $s$–$t$-cuts.

**Lemma 6.4.** Let $(D, c)$ be a directed network, $s, t \in V(D)$. Let $f$ be an $s$–$t$-flow of $(D, c)$ and $S$ an $s$–$t$-cut of $(D, c)$. Then $|f| \leqslant C(S)$.

*Proof.* For $U, W \subseteq V(D)$, let $F(U, W)$ denote the overall amount of flow on arcs with tail in $U$ and head in $W$, i.e., $F(U, W) = \sum_{e \in A_D^+(U) \cap A_D^-(W)} f(e)$. We claim that

$$\begin{aligned} |f| &= \sum_{e \in A_D^+(s)} f(e) - \sum_{e \in A_D^-(s)} f(e) \\ &= \sum_{v \in S} \Big( \sum_{e \in A_D^+(v)} f(e) - \sum_{e \in A_D^-(v)} f(e) \Big) \\ &= F(S, V(D)) - F(V(D), S) \end{aligned}$$

$$= F(S,S) + F(S,V(D)\setminus S) - F(V(D)\setminus S, S) - F(S,S)$$
$$= F(S,V(D)\setminus S) - F(V(D)\setminus S, S) \leqslant F(S,V(D)\setminus S) \leqslant C(S).$$

Indeed, the second equality holds because $t \notin S$ and thus, by flow conservation, $\sum_{e\in A^+_D(v)} f(e) - \sum_{e\in A^-_D(s)} f(v) = 0$ for all $v \in S\setminus\{s\}$. The first inequality holds because $f(e) \geqslant 0$ for all $e \in A(D)$, the second inequality because $f(e) \leqslant c(e)$ for all $e \in A(D)$. □

It follows immediately that a flow and a cut with equal size and capacity must in fact be a maximum flow and a minimum cut.

**Corollary 6.5.** Let $(D, c)$ be a directed network, $s, t \in V(D)$. Let $f$ be an $s$—$t$-flow of $(D, c)$ and $S$ an $s$—$t$-cut of $(D, c)$ such that $|f| = C(S)$. Then $f$ is a maximum $s$—$t$-flow of $(D, c)$, and $S$ is a minimum $s$—$t$-cut of $(D, c)$.

**Example 6.6.** Again consider the network of Example 6.2. The $v_1$—$v_6$-flow $g$ has size $|g| = 3$. $S = \{v_1, v_2, v_3\}$ is a $v_1$—$v_6$-cut with capacity $C(S) = 3$. Thus, by Corollary 6.5, $|g|$ is a maximum $v_1$—$v_6$-flow and $S$ is a minimum $v_1$—$v_6$-cut.

## 6.3 Residual Capacities and Augmenting Paths

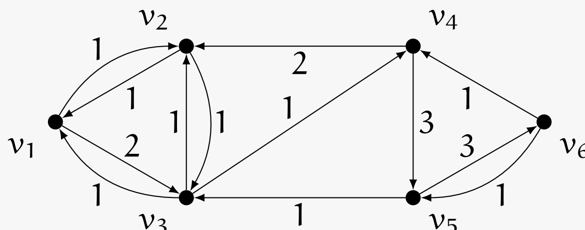
**Definition 6.7** (residual capacity, residual network)**.** Let $(D, c)$ be a directed network, $f$ an $s$—$t$-flow of $(D, c)$. Then the *residual capacity* $c_f(u, v)$ between $u \in V(D)$ and $v \in V(D)$ is

$$c_f(u,v) = \begin{cases} c(uv) - f(uv) + f(vu) & \text{if } uv \in A(D) \text{ and } vu \in A(D) \\ c(uv) - f(uv) & \text{if } uv \in A(D) \text{ and } vu \notin A(D) \\ f(vu) & \text{if } uv \notin A(D) \text{ and } vu \in A(D) \\ 0 & \text{otherwise.} \end{cases}$$

The *residual network* for $(D, c)$ and $f$ is the directed network $(R, c_f)$ where $V(R) = V(D)$ and $A(R) = \{uv : u, v \in V(D), c_f(u, v) > 0\}$.

It is important to note that the residual network may contain arcs not contained in the original network. If $uv \in A(D)$ and $f(uv) > 0$, then $c_f(v, u) > 0$ and the residual network contains the arc $vu$ even if the original network does not. This makes sense because the overall flow from $v$ to $u$ can be increased by decreasing the flow $f(uv)$ in the opposite direction.

**Example 6.8.** Again consider the network of Example 6.2 and the $v_1$–$v_6$-flow $f$, which we have seen is not a maximum $v_1$–$v_6$-flow. The residual network for the network and flow $f$ looks as follows.




Imagine that we find an $s$–$t$-path in the residual network. By definition the residual capacity will be positive for every consecutive pair of vertices along the path, and we can send some positive amount of flow along the path by increasing the flow for every consecutive pair of vertices by this amount. For every vertex on the path except $s$ and $t$, the overall amount of flow entering the vertex and the overall amount of flow leaving the vertex will increase by the same amount, so flow conservation will not be affected. An $s$–$t$-path in the residual network is also called an augmenting path, because it allows us to augment the current flow.

**Definition 6.9** (flow augmenting path, path residual capacity, forward arc, backward arc)**.** Let $(D, c)$ be a directed network, $s, t \in V(D)$. Let $f$ be an $s$–$t$-flow of $(D, c)$. Let $P = v_0 v_1 v_2 \dots v_m$ be a sequence of distinct vertices in $V(D)$. Then $P$ is an $f$-augmenting $s$–$t$-path if $v_0 = s$, $v_m = t$, and $c_f(v_{i-1}, v_i) > 0$ for all $i \in [m]$. The *residual capacity* of $P$ is $c_f(P) = \min_{i \in [m]} c_f(v_{i-1}, v_i)$. An arc $e \in A(D)$ is a *forward arc* on $P$ if it has tail $v_{i-1}$ and head $v_i$ for some $i \in [m]$, and a *backward arc* on $P$ if if it has tail $v_i$ and head $v_{i-1}$ for some $i \in [m]$.

**Lemma 6.10.** Let $(D, c)$ be a directed network, $s, t \in V(D)$, and $f$ be an $s$–$t$-flow of $(D, c)$. Let $P$ be an $f$-augmenting $s$–$t$-path, and let $F \subseteq A(D)$ and $B \subseteq A(D)$ the sets of forward and backward arcs on $P$. Let $g : A(D) \to \mathbb{R}$ such that

$$g(uv) = \begin{cases} \max\{f(uv) - c_f(P), 0\} & \text{if } uv \in B, \\ f(uv) + \max\{c_f(P) - f(vu), 0\} & \text{if } uv \in F \text{ and } vu \in B, \\ f(uv) + c_f(P) & \text{if } uv \in F \text{ and } vu \notin B, \\ f(uv) & \text{if } uv \notin F \cup B. \end{cases}$$

Then $g$ is an $s$–$t$-flow of $(D, c)$ and $|g| = |f| + c_f(P)$.

It is straightforward if a bit tedious to verify that $g$ satisfies the capacity and flow conservation constraints and the excess amount leaving $s$ is greater by $c_f(P)$ compared to $f$. We leave this as an exercise.

**Example 6.11.** Again consider the network of Example 6.2 and the $v_1$—$v_6$-flow $f$. We can see from the residual network in Example 6.8 that $P = v_1, v_3, v_4, v_5, v_6$ is an $f$-augmenting $v_1$—$v_6$-path, and $c_f(P) = \min\{c_f(v_1, v_3), c_f(v_3, v_4), c_f(v_4, v_5), c_f(v_5, v_6)\} = \min\{2, 1, 3, 3\} = 1$. By augmenting $f$ as in Lemma 6.10 we obtain the flow $h$ with

$$\begin{aligned}
h(v_1v_2) &= f(v_1v_2) = 1, & h(v_1v_3) &= f(v_1v_3) + 1 = 2,\\
h(v_3v_2) &= f(v_3v_2) = 1, & h(v_2v_4) &= f(v_2v_4) = 2,\\
h(v_4v_3) &= f(v_4v_3) - 1 = 0, & h(v_3v_5) &= f(v_3v_5) = 1,\\
h(v_4v_6) &= f(v_4v_6) = 1, & h(v_4v_5) &= f(v_4v_5) + 1 = 1,\\
h(v_5v_6) &= f(v_5v_6) + 1 = 2.
\end{aligned}$$

Flow $h$ has size $|h| = |f| + 1 = 3$, and the $v_1$—$v_6$-cut $\{v_1, v_2, v_3\}$ with capacity 3 shows that it is a maximum $v_1$—$v_6$-flow. Note that $h$ is different from the maximum $v_1$—$v_6$-flow $g$ given in Example 6.2, which illustrates that maximum flows need not be unique.

It now seems natural to ask whether we can find a maximum flow by repeatedly finding an augmenting path and thus a flow of greater size. It turns out that this will work, because we can always find an augmenting path unless we are prevented from doing so by a minimum cut.

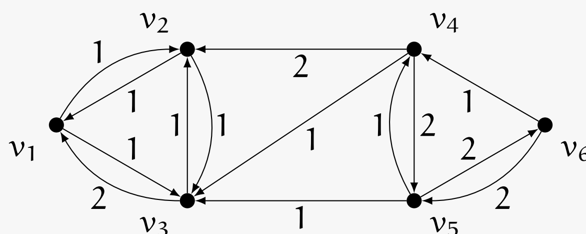
**Lemma 6.12.** Let $(D, c)$ be a directed network, $s, t \in V(D)$, and $f$ be an $s$—$t$-flow of $(D, c)$. Then one of the following is true: (i) there exists an $f$-augmenting $s$—$t$-path; (ii) there exists an $s$—$t$-cut $S$ of $(D, c)$ with $C(S) = |f|$.

*Proof.* Let $S$ be the set of all vertices $v$ such that there exists an $f$-augmenting $s$—$v$-path in $(D, c)$. If $t \in S$, there exists an $f$-augmenting $s$—$t$-path in $(D, c)$.

Now consider the case where $t \notin S$, i.e., where $S$ is an $s$—$t$-cut. As in the proof of Lemma 6.4, let $F(U, W)$ denote the overall amount of flow on arcs with tail in $U \subseteq V(D)$ and head in $W \subseteq V(D)$. Since $S$ is an $s$—$t$-cut, we know from the proof of Lemma 6.4 that $|f| = F(S, V(D) \setminus S) - F(V(D) \setminus S, S) \leqslant F(S, V(D) \setminus S) \leqslant C(S)$. We will see that both inequalities in fact hold with equality. By definition of $S$, the flow across any arc with tail in $S$ and head in $V(D) \setminus S$ must equal the capacity of that arc. That is, $f(e) = c(e)$ for any $e \in A_D^+(S) \cap A_D^-(V(D) \setminus S)$, and thus $F(S, V(D) \setminus S) = C(S)$. Similarly, the flow across any arc with tail in $V(D) \setminus S$ and head in $S$ must be equal to zero. That is, $f(e) = 0$ for any $e \in A_D^+(V(D) \setminus S) \cap A_D^-(S)$, and thus $F(V(D) \setminus S, S) = 0$. Thus $C(S) = |f|$, as

claimed. □

**Example 6.13.** Consider the residual network for the network of Example 6.2 and the $v_1$–$v_6$-flow $h$ of Example 6.11, which looks as follows.




The set of vertices reachable from $v_1$ along a $h$-augmenting path is $S = \{v_1, v_2, v_3\}$. For the two arcs with tail in $S$ and head outside $S$, $v_2v_4$ and $v_3v_5$, flow is equal to capacity. For the unique arc with tail outside $S$ and head in $S$, $v_4v_3$, flow is equal to 0. This implies that $C(S) = |h|$, so $S$ is a minimum $v_1$–$v_6$-cut and $h$ a maximum $v_1$–$v_6$-flow.

Lemma 6.4 and Lemma 6.12 together imply the following result.

**Theorem 6.14** (max-flow min-cut)**.** Let $(D, c)$ be a directed network and $s, t \in V(D)$. Let $f$ be a maximum $s$–$t$-flow of $(D, c)$, and $S$ a minimum $s$–$t$-cut of $(D, c)$. Then $|f| = C(S)$.

## 6.4 The Ford-Fulkerson Algorithm

Our strategy for finding a maximum flow is now clear. We start from some flow, like the one that is zero for every arc, and repeatedly augment the flow until this is no longer possible. When this process terminates, we are guaranteed to have found a maximum flow, and we also obtain a minimum cut to prove this.

**Algorithm 6.15** (Ford-Fulkerson)**.** Let $(D, c)$ be a directed network. The *Ford-Fulkerson algorithm* starts from $f : A(D) \to \mathbb{R}$ with $f(e) = 0$ for all $e \in A(D)$ and then repeats the following steps:

1. Let $P$ be an $f$-augmenting $s$–$t$-path of $(D, c)$. If no such path exists, then stop.
2. Augment $f$ by sending $c_f(P)$ units of flow along $P$, as in Lemma 6.10.

It is easy to see that Steps 1 and 2 can be performed efficiently. Indeed, the residual network for a given flow can be computed in time $O(|V(D)| \cdot |A(D)|)$. Given the residual network, an augmenting path can be found in time $O(|V(D)| \cdot |A(D)|)$ using tree search. Given an augmenting path, the flow can finally be

updated in time $O(|A(D)|)$. Bounding the number of iterations seems more difficult, and in fact it is not even clear that this number is finite. In the special case where all capacities are integers, i.e., where $c(e) \in \mathbb{N}$ for all $e \in A(D)$, it is easy to show by induction over the number of iterations that throughout the algorithm all residual capacities and all flow amounts remain integers. Indeed, all residual capacities are differences between capacities and flows, and flow is always increased by a minimum residual capacity. The size of the flow thus increases by at least one in each iteration, and reaches the size of a maximum flow, which is finite, after a finite number of iterations.

Perhaps more surprisingly, this argument also shows the existence of an integral maximum flow, i.e., of a maximum flow where the flow on each arc is an integer.

**Theorem 6.16.** Let $(D, c)$ be a directed network such that $c(e) \in \mathbb{N}$ for all $e \in A(D)$. Let $s, t \in V(D)$. Then there exists a maximum $s$–$t$-flow $f$ of $(D, c)$ such that $f(e) \in \mathbb{N}$ for all $e \in A(D)$.

The argument does not show that Algorithm 6.15 is an efficient algorithm. Indeed, if the augmenting paths are not chosen carefully, the algorithm may take $|f|$ iterations, where $f$ is a maximum flow. As the size of the input to the algorithm is logarithmic in the capacities, $|f|$ may be exponential in the size of the input. When capacities are real numbers, it is in fact possible to construct an example where the algorithm uses augmenting paths with smaller and smaller residual capacity and thus never terminates. Dinitz and later Edmunds and Karp have shown, however, that Algorithm 6.15 has running time $O(|V(D)| \cdot |A(D)|^2)$ even for real-valued capacities if in each iteration $P$ is chosen to be a shortest $f$-augmenting $s$–$t$-path, i.e., one that has a minimum number of vertices among all $f$-augmenting $s$–$t$-paths.

# Chapter 7

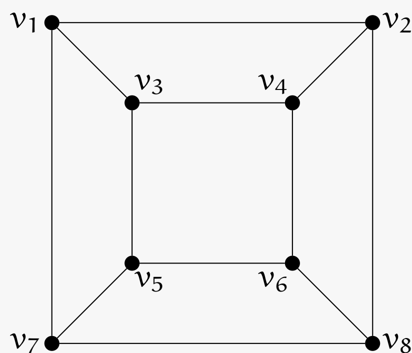
# Matchings

Consider a graph $G$, and imagine that we pair up vertices of $G$ in such a way that only vertices incident to the same edge are paired up and no vertex is paired up with more than one other vertex. The resulting subset of the edges of $G$ is called a matching, and we can ask for a matching in which a certain subset of the vertices are paired up, or one that pairs up as many vertices as possible.

**Definition 7.1** (matching, maximum matching, saturation, perfect matching)**.** Let $G$ be a graph. A set $M \subseteq E(G)$ is a *matching* of $G$ if every vertex $v \in V(G)$ is an endpoint of at most one edge in $M$. A matching $M$ of $G$ is a *maximum matching* of $G$ if it has maximum cardinality among all matchings of $G$. A matching $M$ of $G$ *saturates* $X \subseteq V(G)$ if every $v \in X$ is an endpoint of an edge in $M$. A matching $M$ of $G$ is a *perfect matching* of $G$ if it saturates $V(G)$.

Clearly $|M| \leqslant |V(G)|/2$ for any matching $M$ of $G$. If $M$ is a perfect matching of $G$ then $|M| = |V(G)|/2$, and $M$ must also be a maximum matching.

**Example 7.2.** Consider the following graph.




$M_1 = \{v_1v_2, v_3v_5, v_6v_8\}$ is a matching of $G$. $M_1$ saturates $\{v_1, v_2, v_3, v_5, v_6, v_8\}$,

Figure 7.1: A bipartite graph and a graph that is not bipartite

but it is not a maximum matching. $M_2 = \{v_1v_2, v_3v_4, v_5v_6, v_7v_8\}$ is a perfect matching of $G$, and thus also a maximum matching.

Since a matching cannot contains any loops or multiple edges, we can again restrict our attention to simple graphs.

## 7.1 Bipartite Graphs

We will first study matchings in a restricted but important class of graphs. In graphs from this class, every vertex is of one of two types and every edge has one endpoint of each type.

**Definition 7.3** (bipartite graph, parts). Let $G$ be a graph. Then $G$ is *bipartite* if there exist $L, R \subseteq V(G)$ such that $L \cup R = V(G)$, $L \cap R = \emptyset$, and every edge in $E(G)$ has one endpoint in $L$ and one endpoint in $R$. The sets $L$ and $R$ are then called *parts* of $G$.

The graph on the left of Figure 7.1 is bipartite with parts $L = \{v_1, v_4\}$ and $R = \{v_2, v_3\}$. Indeed, for any bipartite graph, we can prove bipartiteness by giving two parts, i.e., two sets $L$ and $R$ that satisfy the condition of Definition 7.3. The graph on the right of Figure 7.1 is *not* bipartite, but this seems more difficult to show. The following result provides a characterization that is easy to check.

**Theorem 7.4.** Let $G$ be a graph. Then $G$ is bipartite if and only if $G$ does not contain any cycles of odd length.

*Proof.* For the direction from left to right, assume that $G$ is bipartite with parts $L$ and $R$ and consider any cycle contained in $G$. As all edges of $G$ have one endpoint in $L$ and one endpoint in $R$, the cycle must alternate between $L$ and $R$ and must therefore have even length.

For the direction from right to left assume that $G$ does not contain any cycles of odd length. It suffices to show that every connected component of $G$ is bipartite. Let $H$ be a connected component of $G$, $T$ a spanning tree of $H$, and $s \in V(H)$.

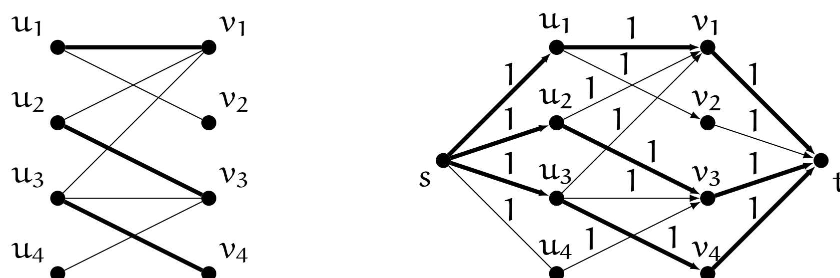



Figure 7.2: A bipartite graph and the corresponding directed network. A matching of the bipartite graph and the corresponding flow of the directed network are shown in bold. Flow is equal to 1 for bold arcs and equal to 0 for all other arcs.

Let $L = \{v \in V(H) :$ the unique $s$—$v$-path in $H$ has even length$\}$ and $R\{v \in V(H) :$ the unique $s$—$v$-path in $H$ has odd length$\}$. Clearly $L \cup R = V(H)$ and $L \cap R = \emptyset$. Assume for contradiction that $G$ contains an edge $uv$ such that $u, v \in L$ or $u, v \in R$. Let $P_1$ be the unique $s$—$u$-path in $T$, and $P_2$ the unique $s$—$v$-path in $T$. Since $P_1$ and $P_2$ are paths in $T$, they contain a common $s$—$x$-path for some $x \in V(T)$, as well as an $x$—$u$-path $Q_1$ and an $x$—$v$-path $Q_2$ that do not share any vertices apart from $x$. Since the length of $P_1$ and $P_2$ are both even or both odd, the same is true for the lengths of $Q_1$ and $Q_2$. Together with the edge $uv$, $Q_1$ and $Q_2$ form a cycle, and the length of this cycle is odd. This contradicts the assumption that $G$ does not contain any cycles of odd length. □

Breadth-first search provides an efficient way to test whether a given graph $G$ is bipartite. When applied to $G$ starting from vertex $s \in V(G)$, breadth-first search constructs a spanning tree of the connected component containing $s$ and arranges the vertices in this connected component in layers according to their distance in $G$ from $s$. By the definition of these layers, the endpoints of every edge in the connected component are either in the same layer or in consecutive layers. The connected component thus contains a cycle of odd length if and only if the endpoints of some edge in the connected component are in the same layer.

## 7.2 Maximimum Matchings in Bipartite Graphs

The limited structure of bipartite graphs will allow us to find maximum matchings by finding maximum flows in a closely related directed network.

**Definition 7.5.** Let $G$ be a bipartite graph with parts $L$ and $R$. Then $(D_G, c_G)$ is the directed network with $V(D_G) = V(G) \cup \{s, t\}$, $A(D_G) = \{su : u \in L\} \cup \{uv \in E(G) : u \in L, v \in R\} \cup \{vt : v \in R\}$, and $c_G(e) = 1$ for all $e \in A(D_G)$.

An example for a bipartite graph $G$ and the corresponding directed network

$(D_G, c_G)$ is shown in Figure 7.2.

We will now establish a relationship between matchings of $G$ and $s$–$t$-flows of $(D_G, c_G)$. We first show that any matching $M$ of $G$ naturally gives rise to a flow of size $|M|$ of $(D_G, c_G)$.

**Lemma 7.6.** Let $G$ be a bipartite graph, and let $M$ be a matching of $G$. Let $f : A(D_G) \to \mathbb{R}$ such that $f(e) = 1$ if $e \in M$ or if one of the endpoints of $e$ is equal to $s$ or $t$ and the other endpoint is saturated by $M$, and $f(e) = 0$ otherwise. Then $f$ is an $s$–$t$-flow in $(D_G, c_G)$, and $|f| = |M|$.

It is straightforward to verify that $f$ satisfies the capacity and flow conservation constraints and has size $|M|$. We leave this as an exercise.

On the other hand any integral flow $f$ of $(D_G, c_G)$, i.e., any flow where $f(e) \in \{0, 1\}$ for every arc $e \in A(D_G)$, can be turned back into a matching of cardinality $|f|$.

**Lemma 7.7.** Let $f$ be an $s$–$t$-flow of $(D_G, c_G)$ such that $f(e) \in \{0, 1\}$ for all $e \in A(D_G)$. Let $M = \{uv \in E(G) : u \in L, v \in R, f(uv) = 1\}$. Then $M$ is a matching of $G$, and $|M| = |f|$.

*Proof.* For every $u \in L$,

$$\begin{aligned} |\{uv \in M : v \in R\}| &= |\{uv \in E(G) : v \in R, f(uv) = 1\}| \\ &\leqslant \sum_{e \in A^+_{D_G}(u)} f(e) = \sum_{e \in A^-_{D_G}(u)} f(e) = f(su) \leqslant 1, \end{aligned}$$

where the first inequality holds by the capacity constraints for arcs $e \in A^+_{D_G}(u)$, the second equality by the flow conservation constraint for vertex $u$, and the second inequality by the capacity constraint for arc $su$. Analogously, for every $v \in R$,

$$\begin{aligned} |\{uv \in M : u \in L\}| &= |\{uv \in E(G) : u \in L, f(uv) = 1\}| \\ &\leqslant \sum_{e \in A^-_{D_G}(v)} f(e) = \sum_{e \in A^+_{D_G}(v)} f(e) = f(vt) \leqslant 1. \end{aligned}$$

where the first inequality holds by the capacity constraints for arcs $e \in A^-_{D_G}(v)$, the second equality by the flow conservation constraint for vertex $v$, and the second inequality by the capacity constraint for arc $vt$. Thus every vertex $v \in V(G)$ is an endpoint of at most one edge in $M$, so $M$ is a matching.

Moreover,

$$\begin{aligned}|M| &= |\{uv \in E(G) : u \in L, v \in R, f(uv) = 1\}| \\ &= \sum_{u \in L} \sum_{e \in A^+_{D_G}(u)} f(e) = \sum_{u \in L} \sum_{e \in A^-_{D_G}(u)} f(e) \\ &= \sum_{u \in L} f(su) = \sum_{e \in A^+_{D_G}(s)} f(e) - \sum_{e \in A^-_{D_G}(s)} f(e) = |f|,\end{aligned}$$

where the second equality holds because $f(e) \in \{0, 1\}$ for all $e \in A(D_G)$ and the third equality by the flow conservation constraint for vertex $u$. □

Imagine now that in order to find a maximum matching of a bipartite graph $G$ we apply the Ford-Fulkerson algorithm to $(D_G, c_G)$. By Lemma 7.6 the algorithm finds a flow $f$ whose size equals at least the cardinality of a maximum matching of $G$. However, since all capacities of $(D_G, c_G)$ are integers, the algorithm finds such a flow $f$ that is integral and by Lemma 7.7 can be converted into a matching $M$ of $G$ of size $|f|$. Since the cardinality of $M$ is at least that of a maximum matching, it must itself be a maximum matching. The Ford-Fulkerson algorithm can be executed in polynomial time, and so can the construction of $(D_G, c_G)$ from $G$ and the construction of $M$ from $f$. There thus exists an efficient algorithm for finding a maximum matching of a bipartite graph.

## 7.3 Augmenting Paths

In order to find maximum matchings more directly, it will be useful to investigate what augmenting paths look like when the Ford-Fulkerson algorithm is used to find a maximum matching.

To this end, again consider the example shown in Figure 7.2 involving the matching $M = \{u_1v_1, u_2v_3, u_3v_4\}$ of the bipartite graph on the left and a corresponding $s$—$t$-flow $f$ in the network on the right.

The only $f$-augmenting path in the network on the right is $su_4v_3u_2v_1u_1v_2t$. If we trace this path in the graph on the left, with the exception of $s$ and $t$, we see that it alternates between edges in $M$ and edges not in $M$, corresponding respectively to forward and backward arcs in the network, and both starts and ends with edges not in $M$. The Ford-Fulkerson algorithm augments flow $f$ by sending a flow of 1 along the augmenting path, thus increasing the flow on forward arcs from 0 to 1 and decreasing the flow on backward arcs from 1 to 0. In the graph on the left this corresponds to removing from $M$ those edges that are also on the augmenting path, and replacing them by the edges that are on the augmenting path but not in $M$. We thus obtain a new matching equal to the symmetric difference $M \triangle \{u_4v_3, u_2v_3, u_2v_1, u_1v_1, u_1v_2\} = \{u_1v_2, u_2v_1, u_3v_4, u_4v_3\}$ between matching $M$ and the set of edges on the augmenting path. Since the number of

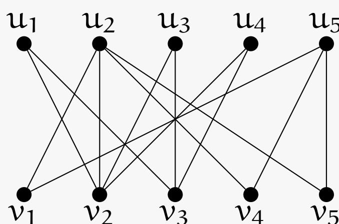
edges on the augmenting path that are not in $M$ is greater by 1 than those that are, the new matching has size $|M| + 1$.

This motivates the following definition of augmenting paths for matchings, as a path that alternates between edges that are and are not in the matching, and both starts and ends with vertices not saturated by the matching.

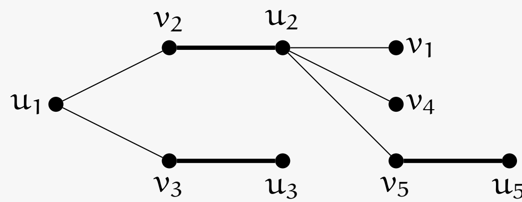
**Definition 7.8** (matching alternating path, matching augmenting path)**.** Let $G$ be a graph, $M$ a matching of $G$. A path in $G$ is an $M$-*alternating path* if it alternates between edges in $M$ and edges in $E(G) \setminus M$, and an $M$-*augmenting path* if it is an $M$-alternating path and starts and ends with vertices not saturated by $M$.

When it is applied to the network $(D_G, c_G)$ for a bipartite graph $G$ with parts $L$ and $R$, the Ford-Fulkerson algorithm finds in each iteration a set of maximal trees of $M$-alternating paths, where $M$ is the current matching. The root of each of these trees is a vertex $u \in L$ not saturated by $M$. If one of the trees contains another vertex $v \in V(G)$ that is not saturated by $M$, then $v \in R$ and the unique $u$–$v$-path in the tree is an $M$-augmenting path. A particular maximal tree $T$ of $M$-alternating paths can be found more directly by a variant of breadth-first search in which the set of edges incident to $v \in T$ that are considered for addition to $T$ is limited to those in $M$ if the unique edge incident to $v$ in $T$ is in $E(G) \setminus M$, and limited to those in $E(G) \setminus M$ if the unique edge incident to $v$ in $T$ is in $M$.

**Example 7.9.** Consider the following bipartite graph with parts $L = \{u_1, u_2, u_3, u_4, u_5\}$ and $R = \{v_1, v_2, v_3, v_4, v_5\}$, and assume that we have found the matching $M_1 = \{u_2v_2, u_3v_3, u_5v_5\}$.

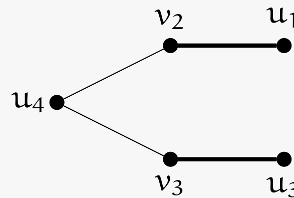



Let $S_1 = \{u_2, u_3, u_5, v_2, v_3, v_5\}$ be the set of vertices saturated by $M_1$. For each $x \in L \setminus S_1 = \{u_1, u_4\}$, we can use a variant of breadth-first search to construct a maximal $M_1$-alternating tree with root $x$. The tree with root $u_1$ for example looks as follows.




This tree contains the vertex $v_1 \in (V(G) \setminus S_1) \setminus \{u_1\} = \{u_4, v_1, v_4\}$, and thus contains the $M_1$-augmenting $u_1$—$v_1$-path $P_1 = u_1v_2u_2v_1$. This implies that $M_2 = M_1 \triangle E(P_1) = \{u_1v_2, u_2v_1, u_3v_3, u_5v_5\}$ is a matching of G with $|M_2| = |M_1| + 1$.

Let $S_2 = \{u_1, u_2, u_3, u_5, v_1, v_2, v_3, v_5\}$ be the set of vertices saturated by $M_2$. For each $x \in L \setminus S_2 = \{u_4\}$, we can again construct a maximal $M_2$-alternating tree with root $x$. There is only one such tree, the one with root $u_4$, which looks as follows.




This tree does not contain any vertices in $(V(G) \setminus S_2) \setminus \{u_4\} = \{v_4\}$, so $M_2$ is a maximum matching.

It follows from the discussion above that $M$ is a maximum matching of a bipartite graph $G$ if and only $G$ does not contain an $M$-augmenting path. This turns out to hold even for graphs that are not bipartite.

**Theorem 7.10.** Let $G$ be a graph, $M$ a matching of $G$. Then $M$ is a maximum matching of $G$ if and only if there is no $M$-augmenting path in $G$.

*Proof.* For the direction from left to right, assume that $M$ is a maximum matching of $G$, and for contradiction that $P$ is an $M$-augmenting path. Let $M' = M \triangle E(P)$. Then $M'$ is a matching of $G$ and $|M'| = |M + 1|$, which contradicts the assumption that $M$ is a maximum matching of $G$.

For the direction from right to left, assume that $M$ is not a maximum matching of $G$, and let $M'$ be a matching of $G$ with $|M'| > |M|$. Let $S = M \triangle M'$, and let $H$ be the graph with $V(H) = V(G)$ and $E(H) = S$. Each vertex $v \in V(G)$ is incident to at most two edges of $S$, so $d_H(v) \leqslant 2$ for all $v \in V(H)$. Each connected component of $H$ is thus either a path or a cycle. Moreover, $d_H(v) = 2$ if and only if $v$ is incident to an edge of $M$ and an edge of $M'$, so the paths and cycles alternate between edges in $M$ and edges in $M'$. It follows in particular that each cycle in

$H$ has even length. Since $|M'| > |M|$, some connected component of $H$ must be a path which starts and ends with an edge of $M'$. This path is an $M$-augmenting path in $G$. □

There exists an efficient algorithm, due to Edmonds, that constructs a maximum matching of a graph $G$ by repeatedly finding an augmenting path. We have seen that in the special case where $G$ is bipartite, augmenting paths can be found by a greedy algorithm. In the case where $G$ is not bipartite, the search for an $M$-augmenting path may be complicated by the presence of so-called blossoms, cycles of length $2k + 1$ in which exactly $k$ edges belong to $M$. The key component of Edmonds' algorithm is a procedure to find and remove blossoms, but the details of this procedure are beyond the scope of this module.

## 7.4 Saturating Matchings in Bipartite Graphs

Consider a bipartite graph $G$ with parts $L$ and $R$ such that $|L| \leqslant |R|$. Since every edge in a matching of $G$ has exactly one endpoint in $L$, any matching of $G$ has size at most $|L|$. On the other hand, any matching of $G$ of size $|L|$ saturates $L$. The following result characterizes when $G$ possesses such a matching.

**Theorem 7.11** (Hall). Let $G$ be a bipartite graph with parts $L$ and $R$ such that $|L| \leqslant |R|$. Then $G$ has a matching that saturates $L$ if and only if $|N_G(X)| \geqslant |X|$ for every $X \subseteq L$, where $N_G(X) = \cup_{v \in X} N_G(v)$.

*Proof.* For the direction from left to right, let $M$ be a matching that saturates $L$. Let $X \subseteq L$. Since every vertex in $L$ is the endpoint of at most one edge in $M$, and every element of $L$ is an endpoint of some edge in $M$, we can define a function $g : X \to N_G(X)$ that maps $v \in X$ to the other endpoint of the edge in $M$ incident to $v$. Since every vertex in $R$ is the endpoint of at most one edge in $M$, $g$ is injective and thus $|N_G(X)| \geqslant |X|$.

For the direction from right to left, assume that $G$ does not have a matching that saturates $L$. Let $M$ be a maximum matching of $G$, and let $u \in L$ be unsaturated by $M$. Let $W$ be the set of all vertices $v \in V(G)$ such that there exists an $M$-alternating $u$–$v$-path in $G$. Let $X = W \cap L$ and $Y = W \cap R$, and note that $u \in X$. To complete the proof it suffices to show that $|N_G(X)| < |X|$. $M$ is a maximum matching of $G$, so by Theorem 7.10 there is no $M$-augmenting path in $G$. $M$ must therefore saturate $W \setminus \{u\}$. Specifically, for every $x \in X \setminus \{u\}$ there exists exactly one $y \in Y$ such that $xy \in M$, and for every $y \in Y$ there exists exactly one $x \in X \setminus \{u\}$ such that $xy \in M$. This means that $M$ defines a bijection between $X \setminus \{u\}$ and $Y$, so $|X \setminus \{u\}| = |Y|$ and thus $|X| = |Y| + 1$. Now consider $y \in N_G(X)$ and let $x \in X$ such that $xy \in E(G)$. If $xy \in M$, then $y$ precedes $x$ on any $M$-alternating $u$–$x$-path. If $xy \notin M$, then any $M$-alternating $u$–$x$-path

$P$ that does not contain $y$ can be turned into an $M$-alternating $u$–$y$-path by appending edge $xy$ and vertex $y$ to $P$. In both cases there exists an $M$-alternating $u$–$y$-path, so $y \in Y$ and thus $N_G(X) \subseteq Y$. Therefore $|N_G(X)| \leqslant |Y| = |X| - 1 < |X|$, as required. □

It is worth pointing out that this proof is constructive, in the sense that a set $X \subseteq L$ with $|N_G(X)| < |X|$ can be found efficiently when it exists. With $X$ at hand, it is then obvious that $G$ cannot have a matching that saturates $L$.

**Example 7.12.** Again consider the bipartite graph of Example 7.9 and the matching $M_2$ of this graph. In an attempt to find a matching of cardinality greater than that of $M_2$ we constructed a maximal $M_2$-alternating tree with root $u_2$, and the vertex set of this tree is the set $W = \{u_4, v_2, v_3, u_1, u_3\}$ of vertices reachable from $u_2$ along an $M_2$-alternating path. Taking $X = W \cap L = \{u_1, u_3, u_4\}$ to be the set of vertices in $W$ that are in the same part of the graph as $u_2$, we see that $N_G(X) = \{v_2, v_3\}$ and thus $|N_G(X)| < |X|$. There can thus be no matching saturating $X$, no matching saturating $L$, and no matching with cardinality greater than $|L| - 1 = 4$. Matching $M_2$ has cardinality 4 and is thus a maximum matching.

**Example 7.13.** Consider a situation where each member of a set $R$ of workers is qualified to complete a subset of a set $L$ of jobs, and we wonder if it is possible to assign jobs to workers such that (i) each job is assigned to a worker, (ii) no worker is assigned more than one job, and (iii) workers are only assigned jobs they are qualified to complete. We can model this situation by a bipartite graph with parts $L$ and $R$ where each edge corresponds to a job and a worker qualified to complete this job. An assignment of jobs to workers as desired then corresponds to a matching in this graph that saturates $L$. If no such matching exists, Theorem 7.11 yields a subset $X \subset L$ of the jobs such that the number of workers qualified to complete jobs in $X$ is smaller than $|X|$.

# Chapter 8

# Euler Trails and Tours

Let us now return to the problem of the Seven Bridges of Königsberg, in the context of which Leonhard Euler laid the foundations of graph theory. The city of Königsberg, now Kaliningrad, is situated on the Pregolya river, and during Euler's time the two sides of the river and two large islands were connected by seven bridges. A graph representing this situation is shown in Figure 8.1. The problem of the seven bridges asks whether it is possible to walk around the city, cross each bridge exactly once, and return to the starting point. It can be stated formally as a question concerning the existence of a tour that visits every edge of the graph. We may also drop the requirement that we need to return to the starting point and ask instead for a trail that visits every edge.

**Definition 8.1** (Euler trail, Euler tour). An *Euler trail* in a graph is a trail that contains every edge of the graph. An *Euler tour* in a graph is an Euler trail that is closed.

Recall that a trail is walk in which all edges are distinct. An Euler trail or tour thus visits every edge exactly once. The graph in Figure 8.2 for example contains the Euler trail $v_4, v_2, v_1, v_3, v_5, v_2, v_3, v_4, v_5$.

An obvious necessary condition for the existence of an Euler tour is that all edges are contained in the same connected component and that all vertices have an even degree. This condition, which was already given by Euler, turns out to also be sufficient.

**Theorem 8.2.** Let $G$ be a connected graph. Then $G$ contains an Euler tour if and only if $d_G(v)$ is even for all $v \in V(G)$.

*Proof.* For the direction from left to right, assume that $G$ contains an Euler tour $R$. Let $v \in V(G)$, and observe that the number of edges that immediately precede $v$ in $R$ is the same as the number of edges that immediately follow $v$ in $R$. Since $R$ contains every edge of $G$ exactly once, $d_G(v)$ is equal to the number of

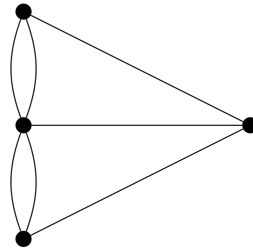


Figure 8.1: A graph representing the seven bridges of Königsberg

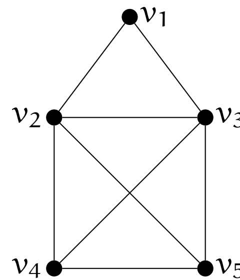



Figure 8.2: A graph containing an Euler trail

edges that immediately precede or follow $v$ in $R$, and this number is even.

For the direction from right to left, assume that $d_G(v)$ is even for all $v \in V(G)$. We will show by induction on $|E(G)|$ that $G$ contains an Euler tour. First assume that $|E(G)| = 0$. Because $G$ is connected, $|V(G)| \leqslant 1$. If $|V(G)| = 0$ the empty sequence is an Euler tour of $G$, if $|V(G)| = 1$ the sequence containing the unique element of $V(G)$ is an Euler tour of $G$. Now assume that $|E(G)| > 0$, and that all connected graphs with fewer edges than $G$ in which all vertices have an even degree contain an Euler tour. Since $|E(G)| > 0$, and since $d_G(v)$ is even for all $v \in V(G)$, $G$ is not a tree and thus contains a cycle. Let $R$ be a tour of maximum length in $G$, and observe that $|E(R)| > 0$. Assume for contradiction that $R$ is not an Euler tour of $G$. Let $H$ be the graph with $V(H) = V(G)$ and $E(H) = E(G) \setminus E(R)$, and observe that $E(H) \neq \emptyset$. Let $J$ be a connected component of $H$ such that $E(J) \neq \emptyset$. Then $d_J(v) = d_G(v) - d_R(v)$ for all $v \in V(J)$, which is even because both $d_G(v)$ and $d_R(v)$ are even. Thus $J$ is a connected graph in which all vertices have an even degree, and $|E(J)| < |E(G)|$, so by the induction hypothesis $J$ contains an Euler tour $Q$. Moreover, since $G$ is connected, there exists a vertex $u \in V(J) \cap V(R)$. There thus exists a tour in $G$ that starts at $u$, follows $R$, and then follows $Q$. This tour is longer than $R$, which contradicts the assumption that $R$ has maximum length. $\square$

The condition for the existence of an Euler trail is only slightly more complicated.

**Corollary 8.3.** Let $G$ be a connected graph. Then $G$ contains an Euler trail if and only if $|\{v \in V(G) : d_G(v) \text{ is odd}\}| \leqslant 2$.

*Proof.* For the direction from left to right assume that $G$ contains an Euler trail $R$. Let $s$ be the first vertex of $R$, $t$ the last vertex of $R$. Let $v \in V(G) \setminus \{s, t\}$, and observe that the number of edges that immediately precede $v$ in $R$ is the same as the number of edges that immediately follow $v$ in $R$. Since $R$ contains every edge of $G$ exactly once, $d_G(v)$ is equal to the number of edges that immediately precede or follow $v$ in $R$, and this number is even. The only vertices whose degrees may be odd are $s$ and $t$, and thus $|\{v \in V(G) : d_G(v) \text{ is odd}\}| \leqslant 2$.

For the direction from right to left assume that $|\{v \in V(G) : d_G(v) \text{ is odd}\}| \leqslant 2$. By Corollary 1.13, the number of vertices of $G$ with odd degree must be either zero or two. If there are no vertices with odd degree, then by Theorem 8.2 $G$ contains an Euler tour, which is also an Euler trail. Now assume that there are two vertices $u, v \in V(G)$ with odd degree. Let $H$ be the graph with $V(H) = V(G)$ and $E(H) = E(G) \cup \{e\}$, where $e$ is a new edge with endpoints $u$ and $v$. Then $H$ is connected and $d_H(v)$ is even for all $v \in V(H)$, so by Theorem 8.2 it contains an Euler tour $R$. If we write $R$ such that it ends with edge $e$ and vertex $v$, and then remove these last two elements, we obtain an Euler trail of $G$. □

In the graph of Figure 8.1 all four vertices have an odd degree, so the graph does not contain an Euler trail. This shows that it was impossible to cross each of the seven bridges of Königsberg exactly once, even without the additional requirement of returning to the starting point.

The proofs of Theorem 8.2 and Corollary 8.3 are constructive, and can be turned into an algorithm that finds an Euler trail if it exists.

**Algorithm 8.4 (Euler trail).** Let $G$ be a connected graph, $s, t \in V(G)$. Assume that $d_G(v)$ is even for all $v \in V(G) \setminus \{s, t\}$. Let $R$ be a maximal trail in $G$ that starts at $s$, and repeat the following steps:

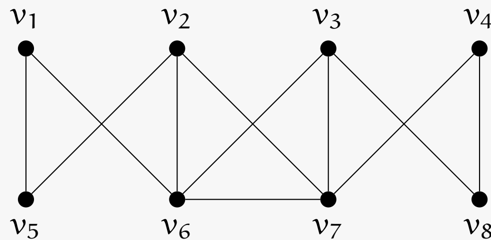
1. Let $x \in V(R)$ with $d_G(x) > d_R(x)$. If no such vertex exists, then stop. $R$ is an Euler trail of $G$.
2. Let $H$ be the graph with $V(H) = V(G)$ and $E(H) = E(G) \setminus E(R)$.
3. Let $Q$ be a maximal trail in $H$ that starts at $x$.
4. Replace one of the occurences of $x$ in $R$ by $Q$.

The initial maximal trail $R$ in $G$ can be found by starting from vertex $s$ and repeatedly following an unvisited edge incident to the current vertex until such an edge no longer exists. If $d_G(s)$ and $d_G(t)$ are odd the trail must end at $t$, if $d_G(s)$ and $d_G(t)$ are even it must end at $s$ and must therefore be a tour. It then remains a trail or a tour until the algorithm terminates. The maximal trail $Q$ in Step 3 can be found by starting from vertex $x$ and repeatedly following an

unvisited edge incident to the current vertex until such an edge no longer exists. The trail must end at $x$ and must therefore be a tour.

To achieve a running time of $O(|E|)$, it suffices to show that a vertex $x \in V(R)$ with $d_G(x) > d_R(x)$ and an edge in $E(G) \setminus E(R)$ incident to a given vertex can be found in constant time. This can be done by maintaining a list of edges in $E(G) \setminus E(R)$ incident to each vertex, as well as a list of vertices for which there is at least one such edge.

**Example 8.5.** Consider the following graph.




Vertices $v_2$ and $v_3$ are the only vertices with odd degree, so the graph contains an Euler trail from $v_2$ to $v_3$. If Algorithm 8.4 is started from vertex $v_2$, it may for example find the maximal trail $R = v_2, v_6, v_3, v_7, v_4, v_8, v_3$. It may then start from the vertex $v_6$, which is both contained in $R$ and an endpoint of an edge *not* contained in $R$, to find the maximal trail $Q = v_6, v_7, v_2, v_5, v_1, v_6$. Replacing $v_6$ in $R$ by $Q$ yields the trail $v_2, v_6, v_7, v_2, v_5, v_1, v_6, v_3, v_7, v_4, v_8, v_3$, which is an Euler trail.