Algorithmic Graph Theory

David Joyner, Minh Van Nguyen, Nathann Cohen

Version 0.7
2013 March 24

Copyright © 2010–2012 David Joyner ⟨wdjoyner@gmail.com⟩
Copyright © 2009–2012 Minh Van Nguyen ⟨mvngu.name@gmail.com⟩
Copyright © 2010 Nathann Cohen ⟨nathann.cohen@gmail.com⟩

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

The latest version of the book is available from its website at

http://code.google.com/p/graphbook/

Edition
Version 0.7
2013 March 24

Contents

Acknowledgments

- Fidel Barrera-Cruz: reported typos in Chapter 3. See changeset 101. Suggested making a note about disregarding the direction of edges in undirected graphs. See changeset 277.

- Daniel Black: reported a typo in Chapter 1. See changeset 61.

- Kevin Brintnall: reported typos in the definition of $\text{iadj}(v) \cap \text{oadj}(v)$; see changesets 240 and 242. Solution to Example 1.14(2); see changeset 246.

- John Costella: helped to clarify the idea that the adjacency matrix of a bipartite graph can be permuted to obtain a block diagonal matrix. See page 22 and revisions 1865 and 1869.

- Aaron Dutle: reported a typo in Figure 1.18. See changeset 125.

- Péter L. Erdős (http://www.renyi.hu/~elp) for informing us of the reference [**?**] on the Havel-Hakimi theorem for directed graphs.

- Noel Markham: reported a typo in Algorithm 2.5. See changeset 131 and Issue 2.

- Caroline Melles: clarify definitions of various graph types (weighted graphs, multigraphs, and weighted multigraphs); clarify definitions of degree, isolated vertices, and pendant and using the butterfly graph with 5 vertices (see Figure 1.10) to illustrate these definitions; clarify definitions of trails, closed paths, and cycles; see changeset 448. Some rearrangements of materials in Chapter 1 to make the reading flow better and a few additions of missing definitions; see changeset 584. Clarifications about unweighted and weighted degree of a vertex in a multigraph; notational convention about a graph being simple unless otherwise stated; an example on graph minor; see changeset 617. Reported a missing edge in Figure 1.5(b); see changeset 1945.

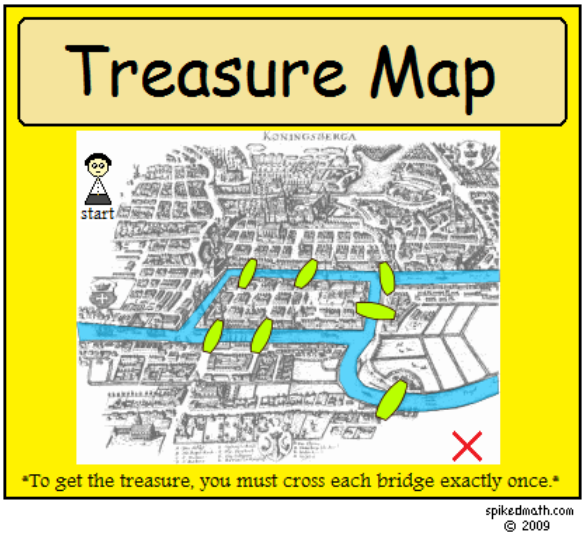
- Pravin Paratey: simplify the sentence formation in the definition of digraphs; see changeset 714 and Issue 7.

- Henrique Rennó: pointed out the ambiguity in the definition of weighted multigraphs; see changeset 1936. Reported typos; see changeset 1938.

- The world map in Figure 2.16 was adapted from an SVG image file from Wikipedia. The original SVG file was accessed on 2010-10-01 at http://en.wikipedia.org/wiki/File:Worldmap_location_NED_50m.svg.

Chapter 1

Introduction to graph theory



— Spiked Math, http://spikedmath.com/120.html

Our journey into graph theory starts with a puzzle that was solved over 250 years ago by Leonhard Euler (1707–1783). The Pregel River flowed through the town of Königsberg, which is present day Kaliningrad in Russia. Two islands protruded from the river. On either side of the mainland, two bridges joined one side of the mainland with one island and a third bridge joined the same side of the mainland with the other island. A bridge connected the two islands. In total, seven bridges connected the two islands with both sides of the mainland. A popular exercise among the citizens of Königsberg was determining if it was possible to cross each bridge exactly once during a single walk. For historical perspectives on this puzzle and Euler's solution, see Gribkovskaia et al. [**?**] and Hopkins and Wilson [**?**].

To visualize this puzzle in a slightly different way, consider Figure 1.1. Imagine that points a and c are either sides of the mainland, with points b and d being the two islands. Place the tip of your pencil on any of the points a, b, c, d. Can you trace all the lines in the figure exactly once, without lifting your pencil? Known as the seven bridges of Königsberg puzzle, Euler solved this problem in 1735 and with his solution he laid the foundation of what is now known as graph theory.

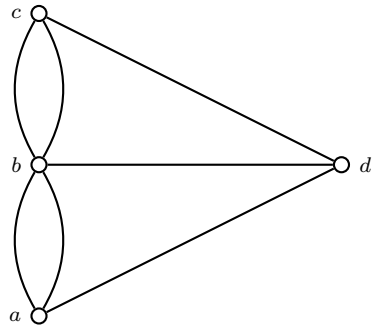
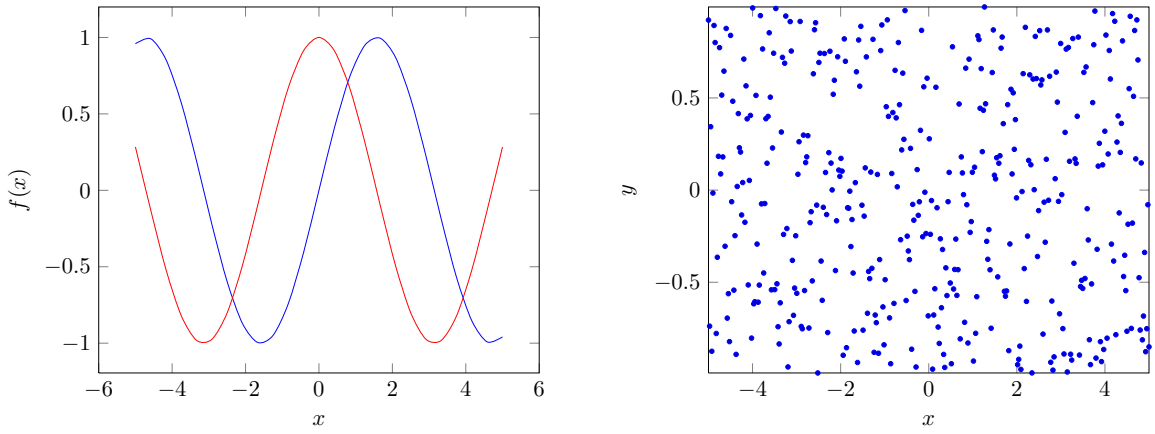


Figure 1.1: The seven bridges of Königsberg puzzle.

1.1 Graphs and digraphs

> When I use a word, it means just what I choose it to mean—neither more nor less.
> — Humpty Dumpty in Lewis Carroll's *Through the Looking Glass*

The word "graph" is commonly understood to mean a visual representation of a dataset, such as a bar chart, a histogram, a scatterplot, or a plot of a function. Examples of such graphs are shown in Figure 1.2.



(a) Plots of functions.

(b) A scatterplot.

Figure 1.2: Visual representations of datasets as plots.

This book is not about graphs in the sense of plots of functions or datasets. Rather, our focus is on *combinatorial graphs* or *graphs* for short. A graph in the combinatorial sense is a collection of discrete interconnected elements, an example of which is shown in Figure 1.1. How can we elaborate on this brief description of combinatorial graphs? To paraphrase what Felix Klein said about curves,[1] it is easy to define a graph until we realize the countless number of exceptions. There are directed graphs, weighted graphs, multigraphs, simple graphs, and so on. Where do we begin?

Notation If S is a set, let $S^{(n)}$ denote the set of unordered n-tuples (with possible repetition). We shall sometimes refer to an unordered n-tuple as an *n-set*.

[1] "Everyone knows what a curve is, until he has studied enough mathematics to become confused through the countless number of possible exceptions."

We start by calling a "graph" what some call an "unweighted, undirected graph without multiple edges."

Definition 1.1. A *graph* $G = (V, E)$ is an ordered pair of finite sets. Elements of V are called *vertices* or *nodes*, and elements of $E \subseteq V^{(2)}$ are called *edges* or *arcs*. We refer to V as the *vertex set* of G, with E being the *edge set*. The cardinality of V is called the *order* of G, and $|E|$ is called the *size* of G. We usually disregard any direction of the edges and consider (u, v) and (v, u) as one and the same edge in G. In that case, G is referred to as an *undirected graph*.

One can label a graph by attaching labels to its vertices. If $(v_1, v_2) \in E$ is an edge of a graph $G = (V, E)$, we say that v_1 and v_2 are *adjacent* vertices. For ease of notation, we write the edge (v_1, v_2) as v_1v_2. The edge v_1v_2 is also said to be *incident* with the vertices v_1 and v_2.

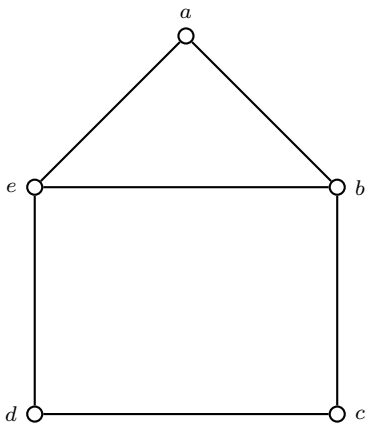


Figure 1.3: A house graph.

Example 1.2. *Consider the graph in Figure 1.3.*

1. *List the vertex and edge sets of the graph.*
2. *For each vertex, list all vertices that are adjacent to it.*
3. *Which vertex or vertices have the largest number of adjacent vertices? Similarly, which vertex or vertices have the smallest number of adjacent vertices?*
4. *If all edges of the graph are removed, is the resulting figure still a graph? Why or why not?*
5. *If all vertices of the graph are removed, is the resulting figure still a graph? Why or why not?*

Solution. (1) Let $G = (V, E)$ denote the graph in Figure 1.3. Then the vertex set of G is $V = \{a, b, c, d, e\}$. The edge set of G is given by

$$E = \{ab,\ ae,\ ba,\ bc,\ be,\ cb,\ cd,\ dc,\ de,\ ed,\ eb,\ ea\}. \tag{1.1}$$

We can also use Sage to construct the graph G and list its vertex and edge sets:

```
sage: G = Graph({"a":["b","e"], "b":["a","c","e"], "c":["b","d"],
...    "d":["c","e"], "e":["a","b","d"]})
sage: G
Graph on 5 vertices
sage: G.vertices()
['a', 'b', 'c', 'd', 'e']
sage: G.edges(labels=False)
[('a', 'b'), ('a', 'e'), ('b', 'e'), ('c', 'b'), ('c', 'd'), ('e', 'd')]
```

The graph G is undirected, meaning that we do not impose direction on any edges. Without any direction on the edges, the edge ab is the same as the edge ba. That is why `G.edges()` returns six edges instead of the 12 edges listed in (1.1).

(2) Let $\text{adj}(v)$ be the set of all vertices that are adjacent to v. Then we have

$$\begin{aligned}\text{adj}(a) &= \{b, e\}\\ \text{adj}(b) &= \{a, c, e\}\\ \text{adj}(c) &= \{b, d\}\\ \text{adj}(d) &= \{c, e\}\\ \text{adj}(e) &= \{a, b, d\}.\end{aligned}$$

The vertices adjacent to v are also referred to as its *neighbors*. We can use the function `G.neighbors()` to list all the neighbors of each vertex.

```
sage: G.neighbors("a")
['b', 'e']
sage: G.neighbors("b")
['a', 'c', 'e']
sage: G.neighbors("c")
['b', 'd']
sage: G.neighbors("d")
['c', 'e']
sage: G.neighbors("e")
['a', 'b', 'd']
```

(3) Taking the cardinalities of the above five sets, we get $|\text{adj}(a)| = |\text{adj}(c)| = |\text{adj}(d)| = 2$ and $|\text{adj}(b)| = |\text{adj}(e)| = 3$. Thus a, c and d have the smallest number of adjacent vertices, while b and e have the largest number of adjacent vertices.

(4) If all the edges in G are removed, the result is still a graph, although one without any edges. By definition, the edge set of any graph is a subset of $V^{(2)}$. Removing all edges of G leaves us with the empty set $\emptyset$, which is a subset of every set.

(5) Say we remove all of the vertices from the graph in Figure 1.3 and in the process all edges are removed as well. The result is that both of the vertex and edge sets are empty. This is a special graph known as the *empty* or *null* graph. ■

Example 1.3. *Consider the illustration in Figure 1.4. Does Figure 1.4 represent a graph? Why or why not?*

Solution. If $V = \{a, b, c\}$ and $E = \{aa, bc\}$, it is clear that $E \subseteq V^{(2)}$. Then (V, E) is a graph. The edge aa is called a *self-loop* of the graph. In general, any edge of the form vv is a self-loop. ■

In Figure 1.3, the edges ae and ea represent one and the same edge. If we do not consider the direction of the edges in the graph of Figure 1.3, then the graph has six edges. However, if the direction of each edge is taken into account, then there are 12 edges as listed in (1.1). The following definition captures the situation where the direction of the edges are taken into account.

A *directed edge* is an edge such that one vertex incident with it is designated as the *head* vertex and the other incident vertex is designated as the *tail* vertex. In this

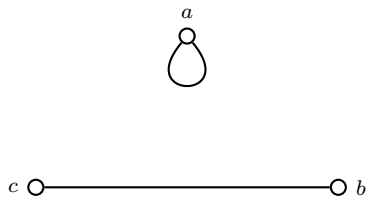


Figure 1.4: A figure with a self-loop.

situation, we may assume that the set of edges is a subset of the ordered pairs $V \times V$. A directed edge uv is said to be directed from its tail u to its head v. A *directed graph* or *digraph* G is a graph each of whose edges is directed. The *indegree* $\mathrm{id}(v)$ of a vertex $v \in V(G)$ counts the number of edges such that v is the head of those edges. The *outdegree* $\mathrm{od}(v)$ of a vertex $v \in V(G)$ is the number of edges such that v is the tail of those edges. The *degree* $\deg(v)$ of a vertex v of a digraph is the sum of the indegree and the outdegree of v.

Let G be a graph without self-loops and multiple edges. It is important to distinguish a graph G as being directed or undirected. If G is undirected and $uv \in E(G)$, then uv and vu represent the same edge. In case G is a digraph, then uv and vu are different directed edges. For a digraph $G = (V, E)$ and a vertex $v \in V$, all the neighbors of v in G are contained in $\mathrm{adj}(v)$, i.e. the set of all neighbors of v. Just as we distinguish between indegree and outdegree for a vertex in a digraph, we also distinguish between in-neighbors and out-neighbors. The set of *in-neighbors* $\mathrm{iadj}(v) \subseteq \mathrm{adj}(v)$ of $v \in V$ consists of all those vertices that contribute to the indegree of v. Similarly, the set of *out-neighbors* $\mathrm{oadj}(v) \subseteq \mathrm{adj}(v)$ of $v \in V$ are those vertices that contribute to the outdegree of v. Then

$$\mathrm{iadj}(v) \cap \mathrm{oadj}(v) = \{u \mid uv \in E \text{ and } vu \in E\}$$

and $\mathrm{adj}(v) = \mathrm{iadj}(v) \cup \mathrm{oadj}(v)$.

1.1.1 Multigraphs

This subsection presents a larger class of graphs. For simplicity of presentation, in this book we shall assume *usually* that a graph is not a multigraph. In other words, when you read a property of graphs later in the book, it will be assumed (unless stated explicitly otherwise) that the graph is not a multigraph. However, as multigraphs and weighted graphs *are* very important in many applications, we will try to keep them in the back of our mind. When appropriate, we will add as a remark how an interesting property of "ordinary" graphs extends to the multigraph or weighted graph case.

An important class of graphs consist of those graphs having multiple edges between pairs of vertices. A *multigraph* is a graph in which there are multiple edges between a pair of vertices. A *multi-undirected graph* is a multigraph that is undirected. Similarly, a *multidigraph* is a directed multigraph.

Example 1.4. Sage can compute with and plot multigraphs, or multidigraphs, having loops.

```
sage: G = Graph({0:{0:'e0',1:'e1',2:'e2',3:'e3'}, 2:{5:'e4'}})
sage: G.show(vertex_labels=True, edge_labels=True, graph_border=True)
sage: H = DiGraph({0:{0:"e0"}}, Loops=True)
sage: H.add_edges([(0,1,'e1'), (0,2,'e2'), (0,2,'e3'), (1,2,'e4'), (1,0,'e5')])
sage: H.show(vertex_labels=True, edge_labels=True, graph_border=True)
```

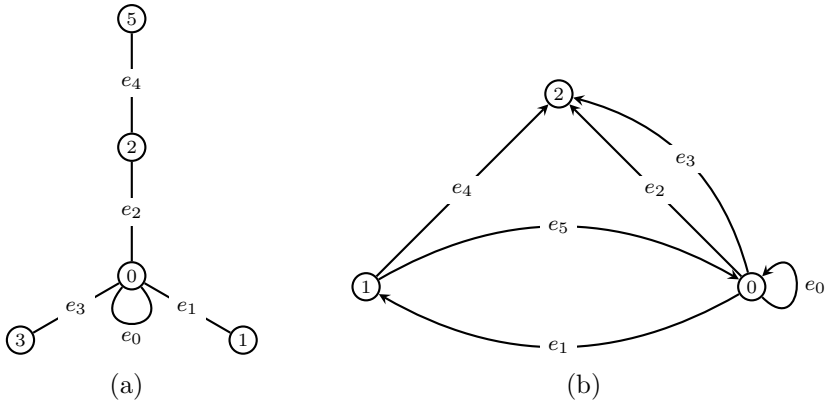


Figure 1.5: A graph G and digraph H with a loop and multi-edges.

These graphs are plotted in Figure 1.5.

As we indicated above, a graph may have "weighted" edges.

Definition 1.5. A *weighted graph* is a graph $G = (V, E)$ where each set V and E is a pair consisting of a vertex and a real number called the *weight*.

The illustration in Figure 1.1 is actually a multigraph, a graph with multiple edges, called the Königsberg graph.

Definition 1.6. For a weighted multigraph G, we are given:

- A finite set V whose elements are pairs (v, w_v), where v is called a *vertex* and $w_v \in \mathbf{R}$ is the *vertex weight*. (Sometimes, the pair (v, w_v) is called the vertex.)
- A finite set E whose elements are *weighted edges*. We do not necessarily assume that $E \subseteq V^{(2)}$, where $V^{(2)}$ is the set of unordered pairs of vertices.[2] Each weighted edge can be represented as a 3-tuple of the form (w_e, u, v), where (u, v) is the edge in question and $w_e \in \mathbf{R}$ is the *edge weight*.
- An incidence function
$$i : E \to V^{(2)}. \tag{1.2}$$

Such a multigraph is denoted $G = (V, E, i)$. An *orientation* on G is a function

$$h : E \to V \tag{1.3}$$

where $h(e) \in i(e)$ for all $e \in E$. The element $v = h(e)$ is called the *head* of $i(e)$. If G has no self-loops, then $i(e)$ is a set having exactly two elements denoted $i(e) = \{h(e), t(e)\}$. The element $v = t(e)$ is called the *tail* of $i(e)$. For self-loops, we set $t(e) = h(e)$. A multigraph with an orientation can therefore be described as the 4-tuple (V, E, i, h). In other words, $G = (V, E, i, h)$ is a multidigraph. Figure 1.6 illustrates a weighted multigraph.

[2] However, we always assume that $E \subseteq \mathbf{R} \times V^{(2)}$, where the $\mathbf{R}$-component is called the *weight* of the edge.

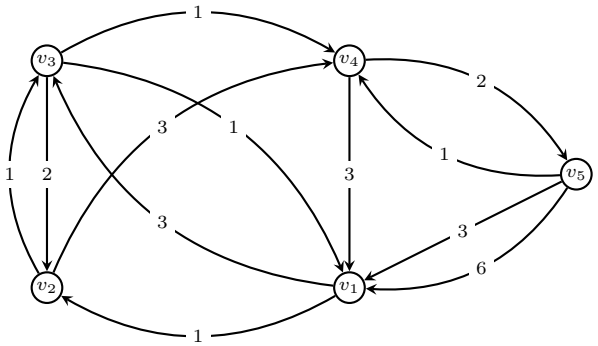


Figure 1.6: An example of a weighted multigraph.

The vertex degree of a weighted multigraph must be defined. There is a weighted degree and an unweighted degree. Let G be a graph as in Definition 1.6. The *unweighted indegree* of a vertex $v \in V$ counts the edges going into v:

$$\deg_+(v) = \sum_{\substack{e \in E \\ h(e)=v}} 1.$$

The *unweighted outdegree* of a vertex $v \in V$ counts the edges going out of v:

$$\deg_-(v) = \sum_{\substack{e \in E \\ v \in i(e)=\{v,v'\} \\ h(e)=v'}} 1.$$

The *unweighted degree* $\deg(v)$ of a vertex v of a weighted multigraph is the sum of the unweighted indegree and the unweighted outdegree of v:

$$\deg(v) = \deg_+(v) + \deg_-(v). \tag{1.4}$$

Loops are counted twice.

Similarly, there is the set of *in-neighbors*

$$\operatorname{iadj}(v) = \{w \in V \mid \text{for some } e \in E,\ i(e) = \{v, w\},\ h(e) = v\}$$

and the set of *out-neighbors*

$$\operatorname{oadj}(v) = \{w \in V \mid \text{for some } e \in E,\ i(e) = \{v, w\},\ h(e) = w\}.$$

Define the *adjacency* of v to be the union of these:

$$\operatorname{adj}(v) = \operatorname{iadj}(v) \cup \operatorname{oadj}(v). \tag{1.5}$$

It is clear that $\deg_+(v) = |\operatorname{iadj}(v)|$ and $\deg_-(v) = |\operatorname{oadj}(v)|$.

The *weighted indegree* of a vertex $v \in V$ counts the weights of edges going into v:

$$\operatorname{wdeg}_+(v) = \sum_{\substack{e \in E \\ h(e)=v}} w_v.$$

The *weighted outdegree* of a vertex $v \in V$ counts the weights of edges going out of v:

$$\operatorname{wdeg}_-(v) = \sum_{\substack{e \in E \\ v \in i(e) = \{v, v'\} \\ h(e) = v'}} w_v.$$

The *weighted degree* of a vertex of a weighted multigraph is the sum of the weighted indegree and the weighted outdegree of that vertex,

$$\operatorname{wdeg}(v) = \operatorname{wdeg}_+(v) + \operatorname{wdeg}_-(v).$$

In other words, it is the sum of the weights of the edges incident to that vertex, regarding the graph as an undirected weighted graph. Unweighted degrees are a special case of weighted degrees. For unweighted degrees, we merely set each edge weight to unity.

Definition 1.7. *Let $G = (V, E, h)$ be an unweighted multidigraph. The* line graph *of G, denoted $\mathcal{L}(G)$, is the multidigraph whose vertices are the edges of G and whose edges are (e, e') where $h(e) = t(e')$ (for $e, e' \in E$). A similar definition holds if G is undirected.*

For example, the line graph of the cyclic graph is itself.

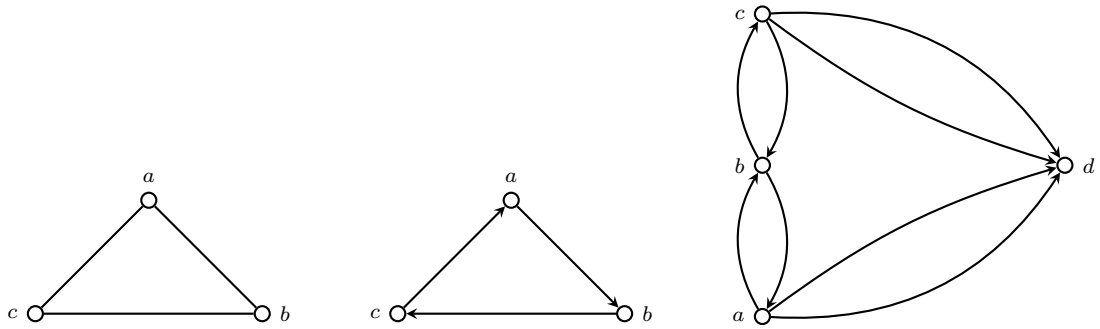
1.1.2 Simple graphs

> Our life is frittered away by detail. ... Simplify, simplify. Instead of three meals a day, if it be necessary eat but one; instead of a hundred dishes, five; and reduce other things in proportion.
> — Henry David Thoreau, *Walden*, 1854, Chapter 2: Where I Lived, and What I Lived For

A *simple graph* is a graph with no self-loops and no multiple edges. Figure 1.7 illustrates a simple graph and its digraph version, together with a multidigraph version of the Königsberg graph. The edges of a digraph can be visually represented as directed arrows, similar to the digraph in Figure 1.7(b) and the multidigraph in Figure 1.7(c). The digraph in Figure 1.7(b) has the vertex set $\{a, b, c\}$ and the edge set $\{ab, bc, ca\}$. There is an arrow from vertex a to vertex b, hence ab is in the edge set. However, there is no arrow from b to a, so ba is not in the edge set of the graph in Figure 1.7(b). The family $\mathrm{Sh}(n)$ of Shannon multigraphs is illustrated in Figure 1.8 for integers $2 \leq n \leq 7$. These graphs are named after Claude E. Shannon (1916–2001) and are sometimes used when studying edge colorings. Each Shannon multigraph consists of three vertices, giving rise to a total of three distinct unordered pairs. Two of these pairs are connected by $\lfloor n/2 \rfloor$ edges and the third pair of vertices is connected by $\lfloor (n+1)/2 \rfloor$ edges.

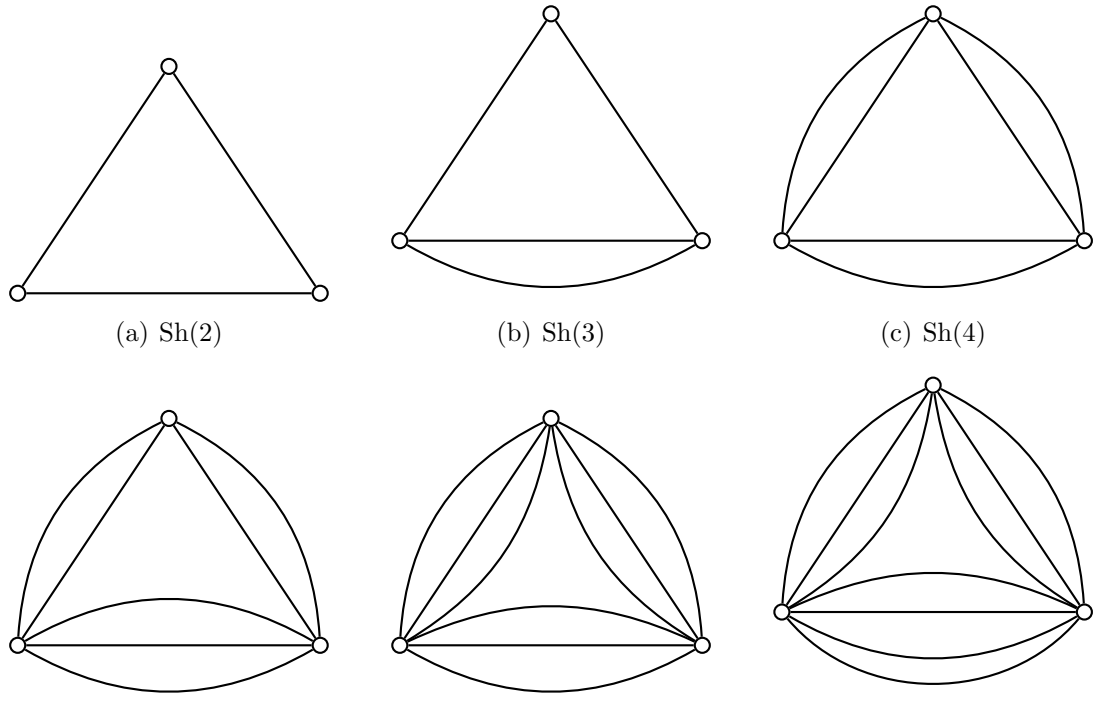
Notational convention *Unless stated otherwise, all graphs are simple graphs in the remainder of this book.*

Definition 1.8. *For any vertex v in a graph $G = (V, E)$, the cardinality of* $\operatorname{adj}(v)$ *(as in 1.5) is called the* degree *of v and written as* $\deg(v) = |\operatorname{adj}(v)|$. *The degree of v counts the number of vertices in G that are adjacent to v. If* $\deg(v) = 0$, *then v is not incident to any edge and we say that v is an* isolated *vertex. If G has no loops and* $\deg(v) = 1$, *then v is called a* pendant.



(a) Simple graph. (b) Digraph. (c) Multidigraph.

Figure 1.7: A simple graph, its digraph version, and a multidigraph.



(a) Sh(2) (b) Sh(3) (c) Sh(4)

(d) Sh(5) (e) Sh(6) (f) Sh(7)

Figure 1.8: The family of Shannon multigraphs Sh(n) for $n = 2, \ldots, 7$.

Some examples would put the above definition in concrete terms. Consider again the graph in Figure 1.4. Note that no vertices are isolated. Even though vertex a is not incident to any vertex other than a itself, note that $\deg(a) = 2$ and so by definition a is not isolated. Furthermore, each of b and c is a pendant. For the house graph in Figure 1.3, we have $\deg(b) = 3$. For the graph in Figure 1.7(b), we have $\deg(b) = 2$. If $V \neq \emptyset$ and $E = \emptyset$, then G is a graph consisting entirely of isolated vertices. From Example 1.2 we know that the vertices a, c, d in Figure 1.3 have the smallest degree in the graph of that figure, while b, e have the largest degree.

The minimum degree among all vertices in G is denoted $\delta(G)$, whereas the maximum degree is written as $\Delta(G)$. Thus, if G denotes the graph in Figure 1.3 then we have $\delta(G) = 2$ and $\Delta(G) = 3$. In the following Sage session, we construct the digraph in Figure 1.7(b) and compute its maximum and minimum number of degrees.

```
sage: G = DiGraph({"a":"b", "b":"c", "c":"a"})
sage: G
Digraph on 3 vertices
sage: G.degree("a")
2
sage: G.degree("b")
2
sage: G.degree("c")
2
```

So for the graph G in Figure 1.7, we have $\delta(G) = \Delta(G) = 2$.

The graph G in Figure 1.7 has the special property that its minimum degree is the same as its maximum degree, i.e. $\delta(G) = \Delta(G)$. Graphs with this property are referred to as *regular*. An *r-regular* graph is a regular graph each of whose vertices has degree r. For instance, G is a 2-regular graph. The following result, due to Euler, counts the total number of degrees in any graph.

Theorem 1.9. ***Euler 1736.*** *If $G = (V, E)$ is a graph, then $\sum_{v \in V} \deg(v) = 2|E|$.*

Proof. Each edge $e = v_1v_2 \in E$ is incident with two vertices, so e is counted twice towards the total sum of degrees. The first time, we count e towards the degree of vertex v_1 and the second time we count e towards the degree of v_2. ■

Theorem 1.9 is sometimes called the "handshaking lemma," due to its interpretation as in the following story. Suppose you go into a room. Suppose there are n people in the room (including yourself) and some people shake hands with others and some do not. Create the graph with n vertices, where each vertex is associated with a different person. Draw an edge between two people if they shook hands. The degree of a vertex is the number of times that person has shaken hands (we assume that there are no multiple edges, i.e. that no two people shake hands twice). The theorem above simply says that the total number of handshakes is even. This is "obvious" when you look at it this way since each handshake is counted twice (A shaking B's hand is counted, and B shaking A's hand is counted as well, since the sum in the theorem is over all vertices). To interpret Theorem 1.9 in a slightly different way within the context of the same room of people, there is an even number of people who shook hands with an odd number of other people. This consequence of Theorem 1.9 is recorded in the following corollary.

Corollary 1.10. *A graph $G = (V, E)$ contains an even number of vertices with odd degrees.*

Proof. Partition V into two disjoint subsets: V_e is the subset of V that contains only vertices with even degrees; and V_o is the subset of V with only vertices of odd degrees.

That is, $V = V_e \cup V_o$ and $V_e \cap V_o = \emptyset$. From Theorem 1.9, we have

$$\sum_{v \in V} \deg(v) = \sum_{v \in V_e} \deg(v) + \sum_{v \in V_o} \deg(v) = 2|E|$$

which can be re-arranged as

$$\sum_{v \in V_o} \deg(v) = \sum_{v \in V} \deg(v) - \sum_{v \in V_e} \deg(v).$$

As $\sum_{v \in V} \deg(v)$ and $\sum_{v \in V_e} \deg(v)$ are both even, their difference is also even. ■

As $E \subseteq V^{(2)}$, then E can be the empty set, in which case the total degree of $G = (V, E)$ is zero. Where $E \neq \emptyset$, then the total degree of G is greater than zero. By Theorem 1.9, the total degree of G is nonnegative and even. This result is an immediate consequence of Theorem 1.9 and is captured in the following corollary.

Corollary 1.11. *If G is a graph, then the sum of its vertex degrees is nonnegative and even.*

If $G = (V, E)$ is an r-regular graph with n vertices and m edges, it is clear by definition of r-regular graphs that the total degree of G is rn. By Theorem 1.9 we have $2m = rn$ and therefore $m = rn/2$. This result is captured in the following corollary.

Corollary 1.12. *If $G = (V, E)$ is an r-regular graph having n vertices and m edges, then $m = rn/2$.*

1.2 Subgraphs and other graph types

We now consider several common types of graphs. Along the way, we also present basic properties of graphs that could be used to distinguish different types of graphs.

Let G be a multigraph as in Definition 1.6, with vertex set $V(G)$ and edge set $E(G)$. Consider a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Furthermore, if $e \in E(H)$ and $i(e) = \{u, v\}$, then $u, v \in V(H)$. Under these conditions, H is called a *subgraph* of G.

1.2.1 Walks, trails, and paths

> I like long walks, especially when they are taken by people who annoy me.
> — Noel Coward

If u and v are two vertices in a graph G, a *u-v walk* is an alternating sequence of vertices and edges starting with u and ending at v. Consecutive vertices and edges are incident. Formally, a *walk* W of length $n \geq 0$ can be defined as

$$W : v_0, e_1, v_1, e_2, v_2, \ldots, v_{n-1}, e_n, v_n$$

where each edge $e_i = v_{i-1}v_i$ and the length n refers to the number of (not necessarily distinct) edges in the walk. The vertex v_0 is the starting vertex of the walk and v_n is the end vertex, so we refer to W as a v_0-v_n walk. The *trivial walk* is the walk of length $n = 0$ in which the start and end vertices are one and the same vertex. If the graph has

no multiple edges then, for brevity, we omit the edges in a walk and usually write the walk as the following sequence of vertices:

$$W : v_0, v_1, v_2, \ldots, v_{n-1}, v_n.$$

For the graph in Figure 1.9, an example of a walk is an a-e walk: a, b, c, b, e. In other words, we start at vertex a and travel to vertex b. From b, we go to c and then back to b again. Then we end our journey at e. Notice that consecutive vertices in a walk are adjacent to each other. One can think of vertices as destinations and edges as footpaths, say. We are allowed to have repeated vertices and edges in a walk. The number of edges in a walk is called its *length*. For instance, the walk a, b, c, b, e has length 4.

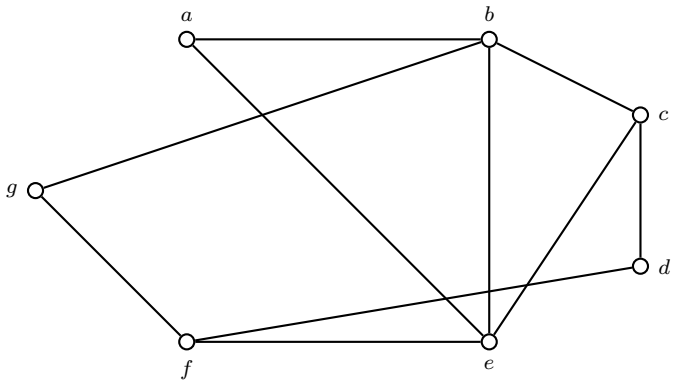


Figure 1.9: Walking along a graph.

A *trail* is a walk with no repeating edges. For example, the a-b walk a, b, c, d, f, g, b in Figure 1.9 is a trail. It does not contain any repeated edges, but it contains one repeated vertex, i.e. b. Nothing in the definition of a trail restricts a trail from having repeated vertices. A walk with no repeating vertices, except possibly the first and last, is called a *path*. Without any repeating vertices, a path cannot have repeating edges, hence a path is also a trail.

Proposition 1.13. *Let $G = (V, E)$ be a simple (di)graph of order $n = |V|$. Any path in G has length at most $n - 1$.*

Proof. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the vertex set of G. Without loss of generality, we can assume that each pair of vertices in the digraph G is connected by an edge, giving a total of n^2 possible edges for $E = V \times V$. We can remove self-loops from E, which now leaves us with an edge set E_1 that consists of $n^2 - n$ edges. Start our path from any vertex, say, v_1. To construct a path of length 1, choose an edge $v_1v_{j_1} \in E_1$ such that $v_{j_1} \notin \{v_1\}$. Remove from E_1 all v_1v_k such that $v_{j_1} \neq v_k$. This results in a reduced edge set E_2 of $n^2 - n - (n-2)$ elements and we now have the path $P_1 : v_1, v_{j_1}$ of length 1. Repeat the same process for $v_{j_1}v_{j_2} \in E_2$ to obtain a reduced edge set E_3 of $n^2 - n - 2(n-2)$ elements and a path $P_2 : v_1, v_{j_1}, v_{j_2}$ of length 2. In general, let $P_r : v_1, v_{j_1}, v_{j_2}, \ldots, v_{j_r}$ be a path of length $r < n$ and let E_{r+1} be our reduced edge set of $n^2 - n - r(n-2)$ elements. Repeat the above process until we have constructed a path $P_{n-1} : v_1, v_{j_1}, v_{j_2}, \ldots, v_{j_{n-1}}$ of length $n - 1$ with reduced edge set E_n of $n^2 - n - (n-1)(n-2)$ elements. Adding another vertex to P_{n-1} means going back to a vertex that was previously visited, because P_{n-1} already contains all vertices of V. ∎

A walk of length $n \geq 3$ whose start and end vertices are the same is called a *closed walk*. A trail of length $n \geq 3$ whose start and end vertices are the same is called a *closed trail*. A path of length $n \geq 3$ whose start and end vertices are the same is called a *closed path* or a *cycle* (with apologies for slightly abusing terminology).[3] For example, the walk a, b, c, e, a in Figure 1.9 is a closed path. A path whose length is odd is called *odd*, otherwise it is referred to as *even*. Thus the walk a, b, e, a in Figure 1.9 is a cycle. It is easy to see that if you remove any edge from a cycle, then the resulting walk contains no closed walks. An *Euler subgraph* of a graph G is either a cycle or an edge-disjoint union of cycles in G. An example of a closed walk which is not a cycle is given in Figure 1.10.

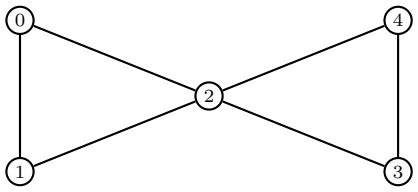


Figure 1.10: Butterfly graph with 5 vertices.

The length of the shortest cycle in a graph is called the *girth* of the graph. By convention, an acyclic graph is said to have infinite girth.

Example 1.14. *Consider the graph in Figure 1.9.*

1. *Find two distinct walks that are not trails and determine their lengths.*
2. *Find two distinct trails that are not paths and determine their lengths.*
3. *Find two distinct paths and determine their lengths.*
4. *Find a closed trail that is not a cycle.*
5. *Find a closed walk C which has an edge e such that $C - e$ contains a cycle.*

Solution. (1) Here are two distinct walks that are not trails: $w_1 : g, b, e, a, b, e$ and $w_2 : f, d, c, e, f, d$. The length of walk w_1 is 5 and the length of walk w_2 is also 5.

(2) Here are two distinct trails that are not paths: $t_1 : a, b, c, e, b$ and $t_2 : b, e, f, d, c, e$. The length of trail t_1 is 4 and the length of trail t_2 is 5.

(3) Here are two distinct paths: $p_1 : a, b, c, d, f, e$ and $p_2 : g, b, a, e, f, d$. The length of path p_1 is 5 and the length of path p_2 is also 5.

(4) Here is a closed trail that is not a cycle: d, c, e, b, a, e, f, d.

(5) Left as an exercise. ■

Theorem 1.15. *Every u-v walk in a graph contains a u-v path.*

Proof. A walk of length $n = 0$ is the trivial path. So assume that W is a u-v walk of length $n > 0$ in a graph G:

$$W : u = v_0, v_1, \ldots, v_n = v.$$

It is possible that a vertex in W is assigned two different labels. If W has no repeated vertices, then W is already a path. Otherwise W has at least one repeated vertex. Let

[3] A cycle in a graph is sometimes also called a "circuit". Since that terminology unfortunately conflicts with the closely related notion of a circuit of a matroid, we do not use it here.

$0 \leq i, j \leq n$ be two distinct integers with $i < j$ such that $v_i = v_j$. Deleting the vertices $v_i, v_{i+1}, \ldots, v_{j-1}$ from W results in a u-v walk W_1 whose length is less than n. If W_1 is a path, then we are done. Otherwise we repeat the above process to obtain a u-v walk shorter than W_1. As W is a finite sequence, we only need to apply the above process a finite number of times to arrive at a u-v path. ■

A graph is said to be *connected* if for every pair of distinct vertices u, v there is a u-v path joining them. A graph that is not connected is referred to as *disconnected*. The empty graph is disconnected and so is any nonempty graph with an isolated vertex. However, the graph in Figure 1.7 is connected. A *geodesic path* or *shortest path* between two distinct vertices u, v of a graph is a u-v path of minimum length. A nonempty graph may have several shortest paths between some distinct pair of vertices. For the graph in Figure 1.9, both a, b, c and a, e, c are geodesic paths between a and c. Let H be a connected subgraph of a graph G such that H is not a proper subgraph of any connected subgraph of G. Then H is said to be a *component* of G. We also say that H is a maximal connected subgraph of G. Any connected graph is its own component. The number of connected components of a graph G will be denoted $\omega(G)$.

The following is an immediate consequence of Corollary 1.10.

Proposition 1.16. *Suppose that exactly two vertices of a graph have odd degree. Then those two vertices are connected by a path.*

Proof. Let G be a graph all of whose vertices are of even degree, except for u and v. Let C be a component of G containing u. By Corollary 1.10, C also contains v, the only remaining vertex of odd degree. As u and v belong to the same component, they are connected by a path. ■

Example 1.17. *Determine whether or not the graph in Figure 1.9 is connected. Find a shortest path from g to d.*

Solution. In the following Sage session, we first construct the graph in Figure 1.9 and use the method `is_connected()` to determine whether or not the graph is connected. Finally, we use the method `shortest_path()` to find a geodesic path between g and d.

```
sage: g = Graph({"a":["b","e"], "b":["a","g","e","c"], \
...    "c":["b","e","d"], "d":["c","f"], "e":["f","a","b","c"], \
...    "f":["g","d","e"], "g":["b","f"]})
sage: g.is_connected()
True
sage: g.shortest_path("g", "d")
['g', 'f', 'd']
```

This shows that g, f, d is a shortest path from g to d. In fact, any other g-d path has length greater than 2, so we can say that g, f, d is the shortest path between g and d. ■

Remark 1.18. We will explain Dijkstra's algorithm in Chapter 2. Dijkstra's algorithm gives one of the best algorithms for finding shortest paths between two vertices in a connected graph. What is very remarkable is that, at the present state of knowledge, finding the shortest path from a vertex v to a *particular* (but arbitrarily given) vertex w appears to be as hard as finding the shortest path from a vertex v to *all* other vertices in the graph!

Trees are a special type of graphs that are used in modelling structures that have some form of hierarchy. For example, the hierarchy within an organization can be drawn as a tree structure, similar to the family tree in Figure 1.11. Formally, a *tree* is an

undirected graph that is connected and has no cycles. If one vertex of a tree is specially designated as the *root vertex*, then the tree is called a *rooted tree*. Chapter 3 covers trees in more details.

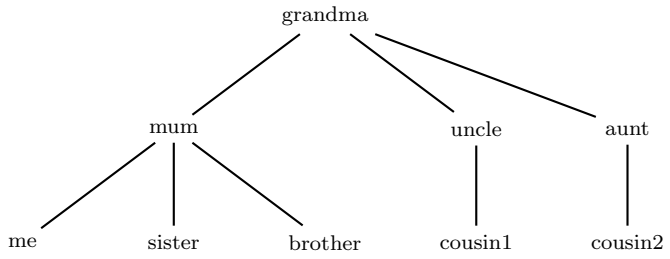


Figure 1.11: A family tree.

1.2.2 Subgraphs, complete and bipartite graphs

Let G be a graph with vertex set $V(G)$ and edge set $E(G)$. Suppose we have a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Furthermore, suppose the incidence function i of G, when restricted to $E(H)$, has image in $V(H)^{(2)}$. Then H is a *subgraph* of G. In this situation, G is referred to as a *supergraph* of H.

Starting from G, one can obtain its subgraph H by deleting edges and/or vertices from G. Note that when a vertex v is removed from G, then all edges incident with v are also removed. If $V(H) = V(G)$, then H is called a *spanning subgraph* of G. In Figure 1.12, let G be the left-hand side graph and let H be the right-hand side graph. Then it is clear that H is a spanning subgraph of G. To obtain a spanning subgraph from a given graph, we delete edges from the given graph.

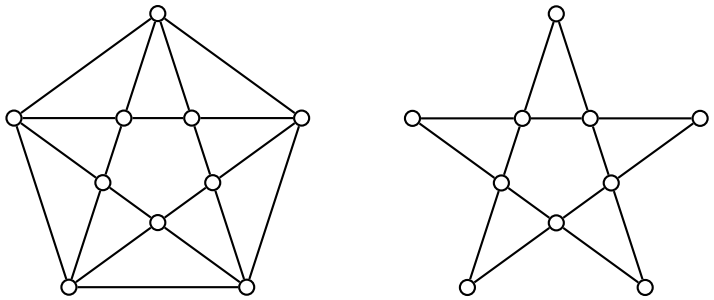


Figure 1.12: A graph and one of its subgraphs.

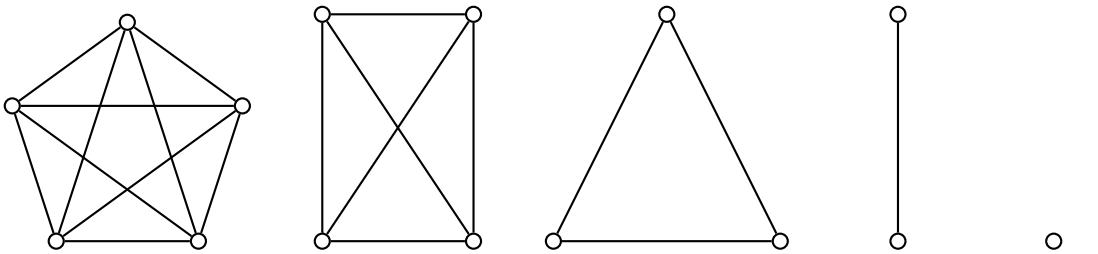
We now consider several standard classes of graphs. The *complete graph* K_n on n vertices is a graph such that any two distinct vertices are adjacent. As $|V(K_n)| = n$, then $|E(K_n)|$ is equivalent to the total number of 2-combinations from a set of n objects:

$$|E(K_n)| = \binom{n}{2} = \frac{n(n-1)}{2}. \tag{1.6}$$

Thus for any simple graph G with n vertices, its total number of edges $|E(G)|$ is bounded above by

$$|E(G)| \leq \frac{n(n-1)}{2}. \tag{1.7}$$

Figure 1.13 shows complete graphs each of whose total number of vertices is bounded by $1 \leq n \leq 5$. The complete graph K_1 has one vertex with no edges. It is also called the *trivial graph*.



(a) K_5 (b) K_4 (c) K_3 (d) K_2 (e) K_1

Figure 1.13: Complete graphs K_n for $1 \leq n \leq 5$.

The following result is an application of inequality (1.7).

Theorem 1.19. *Let G be a simple graph with n vertices and k components. Then G has at most $\frac{1}{2}(n-k)(n-k+1)$ edges.*

Proof. If n_i is the number of vertices in component i, then $n_i > 0$ and it can be shown (see the proof of Lemma 2.1 in [**?**, pp.21–22]) that

$$\sum n_i^2 \leq \left(\sum n_i\right)^2 - (k-1)\left(2\sum n_i - k\right). \tag{1.8}$$

(This result holds true for any nonempty but finite set of positive integers.) Note that $\sum n_i = n$ and by (1.7) each component i has at most $\frac{1}{2}n_i(n_i-1)$ edges. Apply (1.8) to get

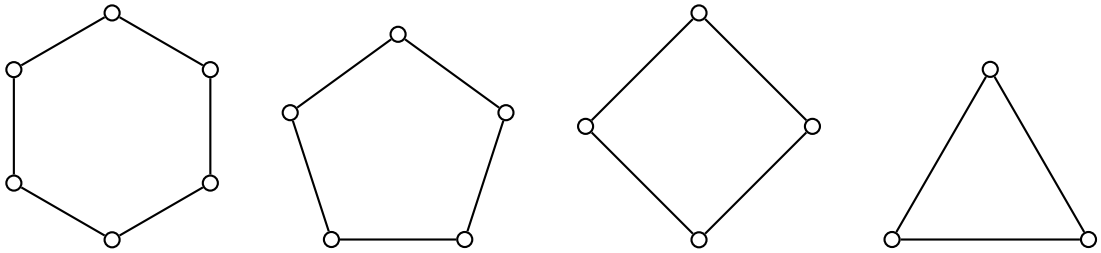
$$\begin{aligned}\sum \frac{n_i(n_i-1)}{2} &= \frac{1}{2}\sum n_i^2 - \frac{1}{2}\sum n_i \\ &\leq \frac{1}{2}(n^2 - 2nk + k^2 + 2n - k) - \frac{1}{2}n \\ &= \frac{(n-k)(n-k+1)}{2}\end{aligned}$$

as required. ■

The *cycle graph* on $n \geq 3$ vertices, denoted C_n, is the connected 2-regular graph on n vertices. Each vertex in C_n has degree exactly 2 and C_n is connected. Figure 1.14 shows cycles graphs C_n where $3 \leq n \leq 6$. The *path graph* on $n \geq 1$ vertices is denoted P_n. For $n = 1, 2$ we have $P_1 = K_1$ and $P_2 = K_2$. Where $n \geq 3$, then P_n is a spanning subgraph of C_n obtained by deleting one edge.

A *bipartite graph* G is a graph with at least two vertices such that $V(G)$ can be split into two disjoint subsets V_1 and V_2, both nonempty. Every edge $uv \in E(G)$ is such that $u \in V_1$ and $v \in V_2$, or $v \in V_1$ and $u \in V_2$. See Kalman [**?**] for an application of bipartite graphs to the problem of allocating satellites to radio stations.

Example 1.20. The Franklin graph, shown in Figure 1.15, is named after Philip Franklin. It is a 3-regular graph with 12 vertices and 18 edges. It is bipartite, Hamiltonian and has radius 3, diameter 3 and girth 4. It is also a 3-vertex-connected and 3-edge-connected perfect graph. ■



(a) C_6 (b) C_5 (c) C_4 (d) C_3

Figure 1.14: Cycle graphs C_n for $3 \leq n \leq 6$.

```
sage: G = graphs.LCFGraph(12, [5,-5], 6)
sage: G.show(dpi=300)
sage: G.is_bipartite()
True
sage: G.chromatic_number()
2
sage: G.girth()
4
sage: G.is_hamiltonian()
True
sage: G.is_vertex_transitive()
True
sage: G.is_planar()
False
sage: G.is_regular()
True
sage: G.coloring()
[[1, 3, 5, 7, 9, 11], [0, 2, 4, 6, 8, 10]]
```

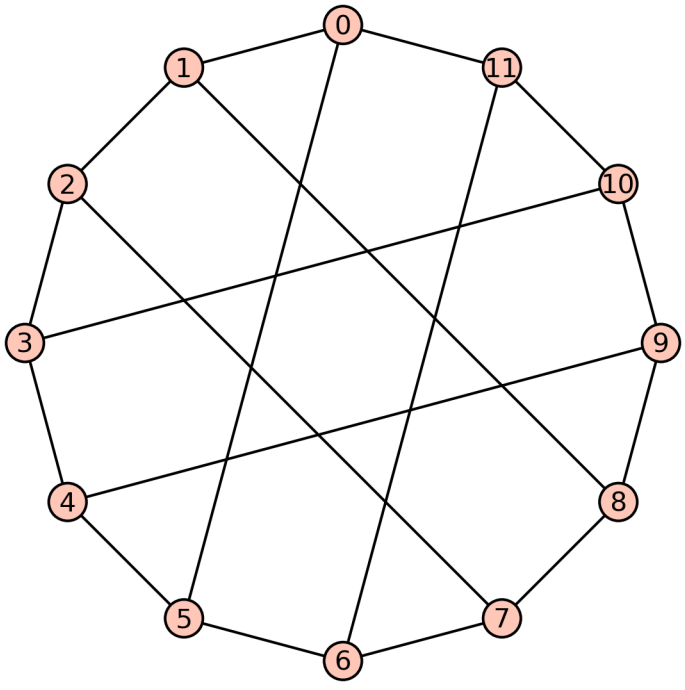


Figure 1.15: Franklin graph example.

Example 1.21. The Foster graph, shown in Figure 1.16, is a 3-regular graph with 90 vertices and 135 edges. This is a bipartite, Hamiltonian graph that has radius 8, diameter

8 and girth 10. It is also a 3-vertex-connected and 3-edge-connected graph. ■

```
sage: G = graphs.LCFGraph(90, [17,-9,37,-37,9,-17], 15)
sage: G.plot(vertex_labels=False, vertex_size=0, graph_border=True).show(dpi=300)
sage: G.is_vertex_transitive()
True
sage: G.is_hamiltonian()
True
sage: G.girth()
10
sage: G.is_bipartite()
True
sage: len(G.vertices())
90
sage: len(G.edges())
135
```

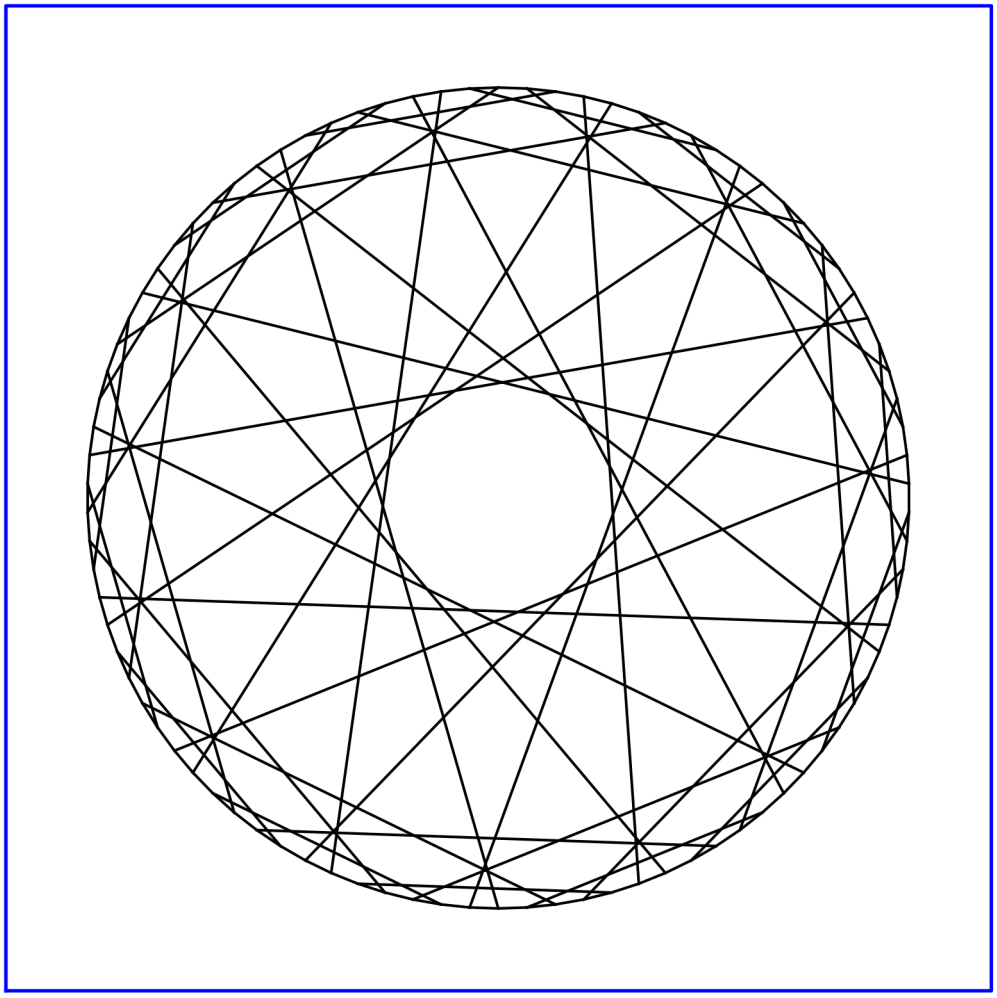


Figure 1.16: Foster graph example.

Theorem 1.22. *A graph is bipartite if and only if it has no odd cycles.*

Proof. Necessity ($\Longrightarrow$): Assume G to be bipartite. Traversing each edge involves going from one side of the bipartition to the other. For a walk to be closed, it must have

even length in order to return to the side of the bipartition from which the walk started. Thus, any cycle in G must have even length.

Sufficiency ($\Longleftarrow$): Assume $G = (V, E)$ has order $n \geq 2$ and no odd cycles. If G is connected, choose any vertex $u \in V$ and define a partition of V thus:

$$X = \{x \in V \mid d(u, x) \text{ is even}\},$$
$$Y = \{y \in V \mid d(u, y) \text{ is odd}\}$$

where $d(u, v)$ denotes the distance (or length of the shortest path) from u to v. If (X, Y) is a bipartition of G, then we are done. Otherwise, (X, Y) is not a bipartition of G. Then one of X and Y has two vertices v, w joined by an edge e. Let P_1 be a shortest u-v path and P_2 be a shortest u-w path. By definition of X and Y, both P_1 and P_2 have even lengths or both have odd lengths. From u, let x be the last vertex common to both P_1 and P_2. The subpath u-x of P_1 and u-x of P_2 have equal length. That is, the subpath x-v of P_1 and x-w of P_2 both have even or odd lengths. Construct a cycle C from the paths x-v and x-w, and the edge e joining v and w. Since x-v and x-w both have even or odd lengths, the cycle C has odd length, contradicting our hypothesis that G has no odd cycles. Hence, (X, Y) is a bipartition of G.

Finally, if G is disconnected, each of its components has no odd cycles. Repeat the above argument for each component to conclude that G is bipartite. ■

Example 1.23. The Gray graph, shown in Figure 1.17, is an undirected bipartite graph with 54 vertices and 81 edges. It is a 3-regular graph discovered by Marion C. Gray in 1932. The Gray graph has chromatic number 2, chromatic index 3, radius 6, and diameter 6. It is also a 3-vertex-connected and 3-edge-connected non-planar graph. The Gray graph is an example of a graph which is edge-transitive but not vertex-transitive.■

```
sage: G = graphs.LCFGraph(54, [-25,7,-7,13,-13,25], 9)
sage: G.plot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: G.is_bipartite()
True
sage: G.is_vertex_transitive()
False
sage: G.is_hamiltonian()
True
sage: G.diameter()
6
```

The *complete bipartite* graph $K_{m,n}$ is the bipartite graph whose vertex set is partitioned into two nonempty disjoint sets V_1 and V_2 with $|V_1| = m$ and $|V_2| = n$. Any vertex in V_1 is adjacent to each vertex in V_2, and any two distinct vertices in V_i are not adjacent to each other. If $m = n$, then $K_{n,n}$ is n-regular. Where $m = 1$ then $K_{1,n}$ is called the *star graph*. Figure 1.18 shows a bipartite graph together with the complete bipartite graphs $K_{4,3}$ and $K_{3,3}$, and the star graph $K_{1,4}$.

As an example of $K_{3,3}$, suppose that there are 3 boys and 3 girls dancing in a room. The boys and girls naturally partition the set of all people in the room. Construct a graph having 6 vertices, each vertex corresponding to a person in the room, and draw an edge form one vertex to another if the two people dance together. If each girl dances three times, once with with each of the three boys, then the resulting graph is $K_{3,3}$.

1.3 Representing graphs as matrices

Neo: What is the Matrix?
Morpheus: Unfortunately, no one can be told what the Matrix is. You have to see it for

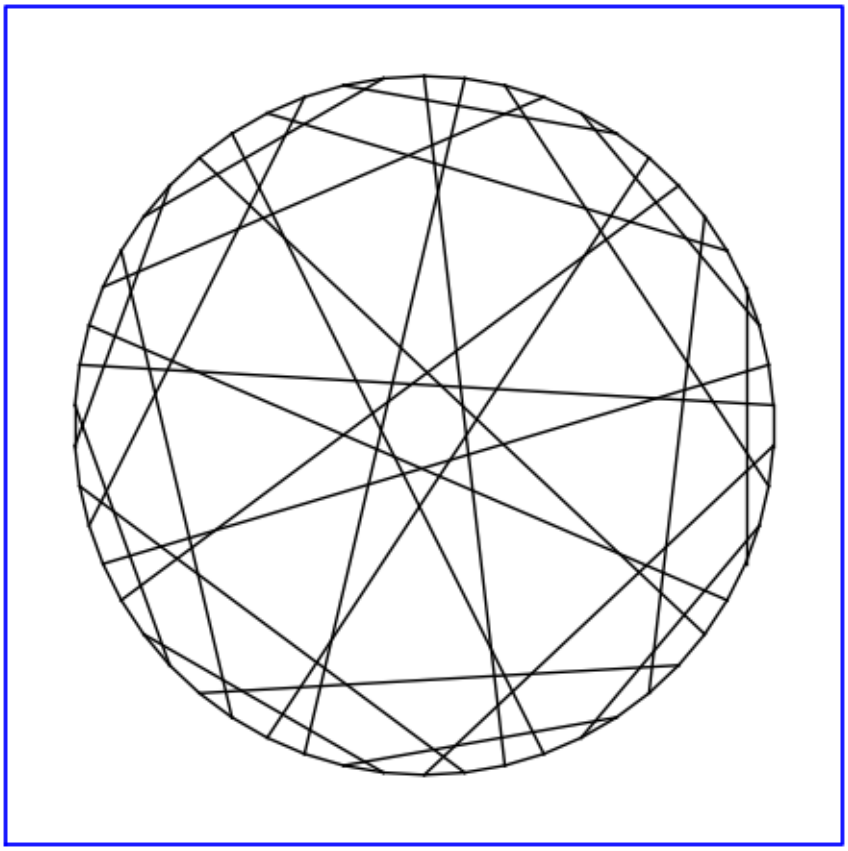
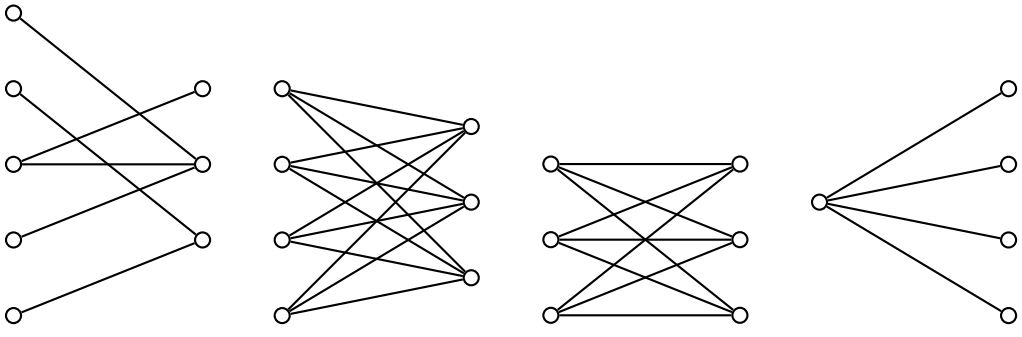


Figure 1.17: Gray graph example.



(a) bipartite (b) $K_{4,3}$ (c) $K_{3,3}$ (d) $K_{1,4}$

Figure 1.18: Bipartite, complete bipartite, and star graphs.

yourself.
— From the movie *The Matrix*, 1999

An $m \times n$ matrix A can be represented as

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

The positive integers m and n are the row and column dimensions of A, respectively. The entry in row i column j is denoted a_{ij}. Where the dimensions of A are clear from context, A is also written as $A = [a_{ij}]$.

Representing a graph as a matrix is very inefficient in some cases and not so in other cases. Imagine you walk into a large room full of people and you consider the "handshaking graph" discussed in connection with Theorem 1.9. If not many people shake hands in the room, it is a waste of time recording all the handshakes and also all the "non-handshakes." This is basically what the adjacency matrix does. In this kind of "sparse graph" situation, it would be much easier to simply record the handshakes as a Python dictionary.[4] This section requires some concepts and techniques from linear algebra, especially matrix theory. See introductory texts on linear algebra and matrix theory [**?**] for coverage of such concepts and techniques.

1.3.1 Adjacency matrix

Let G be an undirected graph with vertices $V = \{v_1, \ldots, v_n\}$ and edge set E. The *adjacency matrix* of G is the $n \times n$ matrix $A = [a_{ij}]$ defined by

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

The adjacency matrix of G is also written as $A(G)$. As G is an undirected graph, then A is a symmetric matrix. That is, A is a square matrix such that $a_{ij} = a_{ji}$.

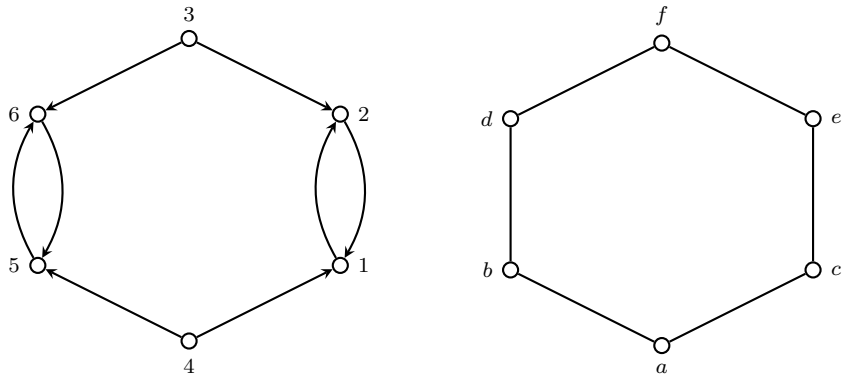
Now let G be a directed graph with vertices $V = \{v_1, \ldots, v_n\}$ and edge set E. The $(0, -1, 1)$-*adjacency matrix* of G is the $n \times n$ matrix $A = [a_{ij}]$ defined by

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

Example 1.24. *Compute the adjacency matrices of the graphs in Figure 1.19.*

Solution. Define the graphs in Figure 1.19 using `DiGraph` and `Graph`. Then call the method `adjacency_matrix()`.

[4] A Python dictionary is basically an indexed set. See the reference manual at http://www.python.org for further details.



(a) (b)

Figure 1.19: What are the adjacency matrices of these graphs?

```
sage: G1 = DiGraph({1:[2], 2:[1], 3:[2,6], 4:[1,5], 5:[6], 6:[5]})
sage: G2 = Graph({"a":["b","c"], "b":["a","d"], "c":["a","e"], \
...    "d":["b","f"], "e":["c","f"], "f":["d","e"]})
sage: m1 = G1.adjacency_matrix(); m1
[0 1 0 0 0 0]
[1 0 0 0 0 0]
[0 1 0 0 0 1]
[1 0 0 0 1 0]
[0 0 0 0 0 1]
[0 0 0 0 1 0]
sage: m2 = G2.adjacency_matrix(); m2
[0 1 1 0 0 0]
[1 0 0 1 0 0]
[1 0 0 0 1 0]
[0 1 0 0 0 1]
[0 0 1 0 0 1]
[0 0 0 1 1 0]
sage: m1.is_symmetric()
False
sage: m2.is_symmetric()
True
```

In general, the adjacency matrix of a digraph is not symmetric, while that of an undirected graph is symmetric. ■

More generally, if G is an undirected multigraph with edge $e_{ij} = v_i v_j$ having multiplicity w_{ij}, or a weighted graph with edge $e_{ij} = v_i v_j$ having weight w_{ij}, then we can define the (weighted) *adjacency matrix* $A = [a_{ij}]$ by

$$a_{ij} = \begin{cases} w_{ij}, & \text{if } v_i v_j \in E, \\ 0, & \text{otherwise.} \end{cases}$$

For example, Sage allows you to easily compute a weighted adjacency matrix.

```
sage: G = Graph(sparse=True, weighted=True)
sage: G.add_edges([(0,1,1), (1,2,2), (0,2,3), (0,3,4)])
sage: M = G.weighted_adjacency_matrix(); M
[0 1 3 4]
[1 0 2 0]
[3 2 0 0]
[4 0 0 0]
```

Bipartite case

Suppose $G = (V, E)$ is an undirected bipartite graph with $n = |V|$ vertices. Any adjacency matrix A of G is symmetric and we assume that it is indexed from zero up to

$n-1$, inclusive. Then there exists a permutation π of the index set $\{0, 1, \ldots, n-1\}$ such that the matrix $A' = [a_{\pi(i)\pi(j)}]$ is also an adjacency matrix for G and has the form

$$A' = \begin{bmatrix} \mathbf{0} & B \\ B^T & \mathbf{0} \end{bmatrix} \tag{1.9}$$

where $\mathbf{0}$ is a zero matrix. The matrix B is called a *reduced adjacency matrix* or a *bi-adjacency matrix* (the literature also uses the terms "transfer matrix" or the ambiguous term "adjacency matrix"). In fact, it is known [**?**, p.16] that any undirected graph is bipartite if and only if there is a permutation π on $\{0, 1, \ldots, n-1\}$ such that $A'(G) = [a_{\pi(i)\pi(j)}]$ can be written as in (1.9).

Tanner graphs

If H is an $m \times n$ $(0,1)$-matrix, then the *Tanner graph* of H is the bipartite graph $G = (V, E)$ whose set of vertices $V = V_1 \cup V_2$ is partitioned into two sets: V_1 corresponding to the m rows of H and V_2 corresponding to the n columns of H. For any i, j with $1 \leq i \leq m$ and $1 \leq j \leq n$, there is an edge $ij \in E$ if and only if the (i,j)-th entry of H is 1. This matrix H is sometimes called the reduced adjacency matrix or the *check matrix* of the Tanner graph. Tanner graphs are used in the theory of error-correcting codes. For example, Sage allows you to easily compute such a bipartite graph from its matrix.

```
sage: H = Matrix([(1,1,1,0,0), (0,0,1,0,1), (1,0,0,1,1)])
sage: B = BipartiteGraph(H)
sage: B.reduced_adjacency_matrix()
[1 1 1 0 0]
[0 0 1 0 1]
[1 0 0 1 1]
sage: B.plot(graph_border=True)
```

The corresponding graph is similar to that in Figure 1.20.

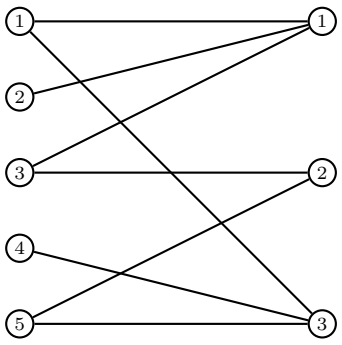


Figure 1.20: A Tanner graph.

Theorem 1.25. *Let A be the adjacency matrix of a graph G with vertex set $V = \{v_1, v_2, \ldots, v_p\}$. For each positive integer n, the ij-th entry of A^n counts the number of v_i-v_j walks of length n in G.*

Proof. We shall prove by induction on n. For the base case $n = 1$, the ij-th entry of A^1 counts the number of walks of length 1 from v_i to v_j. This is obvious because A^1 is merely the adjacency matrix A.

Suppose for induction that for some positive integer $k \geq 1$, the ij-th entry of A^k counts the number of walks of length k from v_i to v_j. We need to show that the ij-th

entry of A^{k+1} counts the number of v_i-v_j walks of length $k+1$. Let $A = [a_{ij}]$, $A^k = [b_{ij}]$, and $A^{k+1} = [c_{ij}]$. Since $A^{k+1} = AA^k$, then

$$c_{ij} = \sum_{r=1}^{p} a_{ir} b_{rj}$$

for $i, j = 1, 2, \ldots, p$. Note that a_{ir} is the number of edges from v_i to v_r, and b_{rj} is the number of v_r-v_j walks of length k. Any edge from v_i to v_r can be joined with any v_r-v_j walk to create a walk $v_i, v_r, \ldots, v_j$ of length $k+1$. Then for each $r = 1, 2, \ldots, p$, the value $a_{ir}b_{rj}$ counts the number of v_i-v_j walks of length $k+1$ with v_r being the second vertex in the walk. Thus c_{ij} counts the total number of v_i-v_j walks of length $k+1$. ■

1.3.2 Incidence matrix

The relationship between edges and vertices provides a very strong constraint on the data structure, much like the relationship between points and blocks in a combinatorial design or points and lines in a finite plane geometry. This incidence structure gives rise to another way to describe a graph using a matrix.

Let G be a digraph with edge set $E = \{e_1, \ldots, e_m\}$ and vertex set $V = \{v_1, \ldots, v_n\}$. The *incidence matrix* of G is the $n \times m$ matrix $B = [b_{ij}]$ defined by

$$b_{ij} = \begin{cases} -1, & \text{if } v_i \text{ is the tail of } e_j, \\ 1, & \text{if } v_i \text{ is the head of } e_j, \\ 2, & \text{if } e_j \text{ is a self-loop at } v_i, \\ 0, & \text{otherwise.} \end{cases} \tag{1.10}$$

Each column of B corresponds to an edge and each row corresponds to a vertex. The definition of incidence matrix of a digraph as contained in expression (1.10) is applicable to digraphs with self-loops as well as multidigraphs.

For the undirected case, let G be an undirected graph with edge set $E = \{e_1, \ldots, e_m\}$ and vertex set $V = \{v_1, \ldots, v_n\}$. The *unoriented incidence matrix* of G is the $n \times m$ matrix $B = [b_{ij}]$ defined by

$$b_{ij} = \begin{cases} 1, & \text{if } v_i \text{ is incident to } e_j, \\ 2, & \text{if } e_j \text{ is a self-loop at } v_i, \\ 0, & \text{otherwise.} \end{cases}$$

An *orientation* of an undirected graph G is an assignment of direction to each edge of G. In other words, each edge has a distinguished vertex called a *head*. In this case, the letter $D = D(G)$ is sometimes used instead of B for the incidence matrix of a digraph or an oriented graph. The *oriented incidence matrix* D of G is defined similarly to the case where G is a digraph: it is the incidence matrix of any orientation of G. For each column of D, we have 1 as an entry in the row corresponding to one vertex of the edge under consideration and -1 as an entry in the row corresponding to the other vertex. Similarly, $d_{ij} = 2$ if e_j is a self-loop at v_i.

Sage allows you to compute the incidence matrix of a graph:

```
sage: G = Graph({1: [2, 4], 2: [1, 3], 3: [2, 6], 4: [1, 5], 5: [4, 6], 6: [3, 5]})
sage: G.incidence_matrix()
 [-1 -1  0  0  0  0]
 [ 0  1 -1  0  0  0]
 [ 0  0  1 -1  0  0]
 [ 1  0  0  0 -1  0]
 [ 0  0  0  0  1 -1]
 [ 0  0  0  1  0  1]
```

The *integral cycle space* of a graph is equal to the kernel of an oriented incidence matrix, viewed as a matrix over $\mathbb{Q}$. The *binary cycle space* is the kernel of its oriented or unoriented incidence matrix, viewed as a matrix over $GF(2)$.

Theorem 1.26. *The incidence matrix of an undirected graph G is related to the adjacency matrix of its line graph $L(G)$ by the following theorem:*

$$A(L(G)) = D(G)^T D(G) - 2I_n \ ,$$

where $A(L(G))$ is the adjacency matrix of the line graph of G.

Proof. Let D_i denote the ith column of D.

Consider the dot product of D_i and D_j, $i \neq j$. The terms contributing to this expression are associated to the vertices which are incident to the ith edge and also to the jth edge. In other words, there is such a vertex (and only one such vertex) if and only if this dot product is equal to 1 if and only if ith edge is incident to the jth edge in G if and only if the vertex in $L(G)$ associated to the ith edge in G is adjacent to the vertex in $L(G)$ associated to the jth edge in G. But this is exactly the condition that the corresponding entry of $A(L(G))$ is equal to 1.

Consider the dot product of D_i with itself. The terms contributing to this expression are associated to the vertices which are incident to the ith edge. There are 2 such vertices so this dot product is equal to 2. Subtracting, the 2 in $2I_n$, gives 0, as expect for the diagonal entries of $A(L(G))$. ■

For a directed graph, the result in the above theorem does not hold in general (except in characteristic 2), as the following example shows.

Example 1.27. Consider the graph shown in Figure 1.21, whose line graph is shown in Figure 1.22. ■

```
sage: G1 = DiGraph({0:[1,2,4], 1:[2,3,4], 2:[3,4]})
sage: G1.show()
sage: D1 = G1.incidence_matrix(); D1
[-1 -1 -1  0  0  0  0  0]
[ 0  0  1 -1 -1 -1  0  0]
[ 0  1  0  0  0  1 -1 -1]
[ 0  0  0  0  1  0  0  1]
[ 1  0  0  1  0  0  1  0]
sage: A1 = G1.adjacency_matrix()
sage: A1
[0 1 1 0 1]
[0 0 1 1 1]
[0 0 0 1 1]
[0 0 0 0 0]
[0 0 0 0 0]

sage: G = Graph({0:[1,2,4], 1:[2,3,4], 2:[3,4]})
sage: D = G.incidence_matrix(); D
[-1 -1 -1  0  0  0  0  0]
[ 0  0  1 -1 -1 -1  0  0]
[ 0  1  0  0  0  1 -1 -1]
[ 0  0  0  0  1  0  0  1]
[ 1  0  0  1  0  0  1  0]
```

```
sage: D.transpose()*D
[ 2  1  1  1  0  0  1  0]
[ 1  2  1  0  0  1 -1 -1]
[ 1  1  2 -1 -1 -1  0  0]
[ 1  0 -1  2  1  1  1  0]
[ 0  0 -1  1  2  1  0  1]
[ 0  1 -1  1  1  2 -1 -1]
[ 1 -1  0  1  0 -1  2  1]
[ 0 -1  0  0  1 -1  1  2]
sage: D*D.transpose()
[ 3 -1 -1  0 -1]
[-1  4 -1 -1 -1]
[-1 -1  4 -1 -1]
[ 0 -1 -1  2  0]
[-1 -1 -1  0  3]
sage: (-1)*G.adjacency_matrix()
[ 0 -1 -1  0 -1]
[-1  0 -1 -1 -1]
[-1 -1  0 -1 -1]
[ 0 -1 -1  0  0]
[-1 -1 -1  0  0]
sage: V = G.vertices()
sage: [G.degree(v) for v in V]
[3, 4, 4, 2, 3]
sage: MS8 = MatrixSpace(QQ, 8,8)
sage: I8 = MS8(1)
sage: D.transpose()*D - 2*I8
[ 0  1  1  1  0  0  1  0]
[ 1  0  1  0  0  1 -1 -1]
[ 1  1  0 -1 -1 -1  0  0]
[ 1  0 -1  0  1  1  1  0]
[ 0  0 -1  1  0  1  0  1]
[ 0  1 -1  1  1  0 -1 -1]
[ 1 -1  0  1  0 -1  0  1]
[ 0 -1  0  0  1 -1  1  0]
sage: LG = G.line_graph()
sage: ALG = LG.adjacency_matrix()
sage: ALG
[0 1 1 1 1 1 0 0]
[1 0 1 1 0 0 1 1]
[1 1 0 0 0 1 0 1]
[1 1 0 0 1 1 1 1]
[1 0 0 1 0 1 1 0]
[1 0 1 1 1 0 0 1]
[0 1 0 1 1 0 0 1]
[0 1 1 1 0 1 1 0]

sage: G3 = Graph({0:[1,2,3,6], 1:[2,5,6,7], 2:[3,4,5],3:[4],4:[5,7],5:[7]})  ## line graph
sage: G3.adjacency_matrix()
[0 1 1 1 0 0 1 0]
[1 0 1 0 0 1 1 1]
[1 1 0 1 1 1 0 0]
[1 0 1 0 1 0 0 0]
[0 0 1 1 0 1 0 1]
[0 1 1 0 1 0 0 1]
[1 1 0 0 0 0 0 0]
[0 1 0 0 1 1 0 0]
```

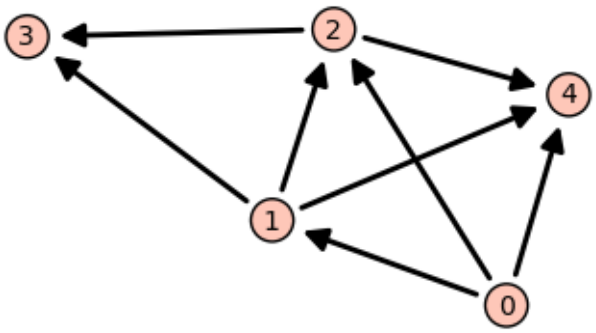


Figure 1.21: A digraph example having 5 vertices and 8 edges.

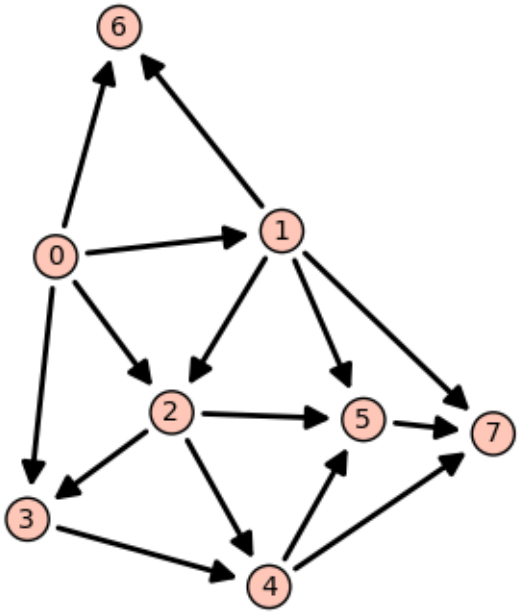


Figure 1.22: The line graph of a digraph example having 5 vertices and 8 edges.

Theorem 1.28. *The rank (over $\mathbb{Q}$) of the incidence matrix of a directed connected simple graph having n vertices is $n-1$.*

Since G is a simple graph, it has fewer edges than vertices.

Proof. Consider the column of D coresponding to $e \in E$. The number of entries equal to $+1$ is one (corresponding to the vertex at the "head" of e) and the number of entries equal to -1 is also one (corresponding to the vertex at the "tail" of e). All other entries are equal to 0. Therefore, the sum of all the rows in D is the zero vector. This implies

$$\operatorname{rank}_{\mathbb{Q}}(D) \leq n-1.$$

To show that equality is attained, we exhibit $n-1$ linearly independent columns of D. Let T be a spanning tree for G. This tree has $n-1$ edges and, if you reindex the vertices of G suitably, the columns of D associated to the edges in T are of the form $w_k = (0, \ldots, 0, 1, -1, 0, \ldots, 0) \in \mathbb{Q}^n$, where the kth entry is a 1 and the $(k+1)$st entry is -1 $(1 \leq k \leq n-1)$. These are clearly linearly independent. ■

1.3.3 Laplacian matrix

The *degree matrix* of a graph $G = (V, E)$ is an $n \times n$ diagonal matrix D whose i-th diagonal entry is the degree of the i-th vertex in V. The *Laplacian matrix* $\mathcal{L}$ of G is the difference between the degree matrix and the adjacency matrix:

$$\mathcal{L} = D - A.$$

In other words, for an undirected unweighted simple graph, $\mathcal{L} = [\ell_{ij}]$ is given by

$$\ell_{ij} = \begin{cases} -1, & \text{if } i \neq j \text{ and } v_i v_j \in E, \\ d_i, & \text{if } i = j, \\ 0, & \text{otherwise,} \end{cases}$$

where $d_i = \deg(v_i)$ is the degree of vertex v_i.

Sage allows you to compute the Laplacian matrix of a graph:

```
sage: G = Graph({1:[2,4], 2:[1,4], 3:[2,6], 4:[1,3], 5:[4,2], 6:[3,1]})
sage: G.laplacian_matrix()
[ 3 -1  0 -1  0 -1]
[-1  4 -1 -1 -1  0]
[ 0 -1  3 -1  0 -1]
[-1 -1 -1  4 -1  0]
[ 0 -1  0 -1  2  0]
[-1  0 -1  0  0  2]
```

There are many remarkable properties of the Laplacian matrix. It shall be discussed further in Chapter 5.

1.3.4 Distance matrix

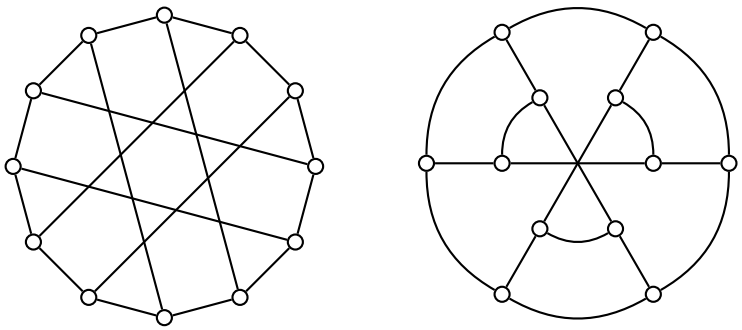
Recall that the distance (or geodesic distance) $d(v, w)$ between two vertices $v, w \in V$ in a connected graph $G = (V, E)$ is the number of edges in a shortest path connecting them. The $n \times n$ matrix $[d(v_i, v_j)]$ is the *distance matrix* of G. Sage helps you to compute the distance matrix of a graph:

```
sage: G = Graph({1:[2,4], 2:[1,4], 3:[2,6], 4:[1,3], 5:[4,2], 6:[3,1]})
sage: d = [[G.distance(i,j) for i in range(1,7)] for j in range(1,7)]
sage: matrix(d)
[0 1 2 1 2 1]
[1 0 1 1 1 2]
[2 1 0 1 2 1]
[1 1 1 0 1 2]
[2 1 2 1 0 3]
[1 2 1 2 3 0]
```

The distance matrix is an important quantity which allows one to better understand the "connectivity" of a graph. Distance and connectivity will be discussed in more detail in Chapters 5 and 10.

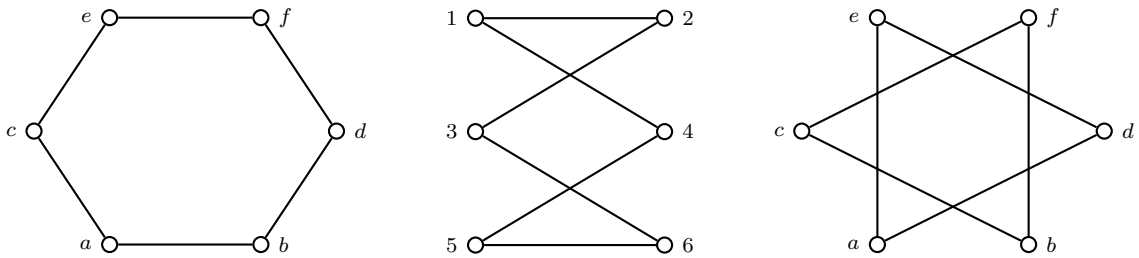
1.4 Isomorphic graphs

Determining whether or not two graphs are, in some sense, the "same" is a hard but important problem. Two graphs G and H are *isomorphic* if there is a bijection $f : V(G) \to V(H)$ such that whenever $uv \in E(G)$ then $f(u)f(v) \in E(H)$. The function f is an *isomorphism* between G and H. Otherwise, G and H are non-isomorphic. If G and H are isomorphic, we write $G \cong H$.



(a) (b)

Figure 1.23: Two representations of the Franklin graph.



(a) C_6 (b) G_1 (c) G_2

Figure 1.24: Isomorphic and nonisomorphic graphs.

A graph G is isomorphic to a graph H if these two graphs can be labelled in such a way that if u and v are adjacent in G, then their counterparts in $V(H)$ are also adjacent in H. To determine whether or not two graphs are isomorphic is to determine if they are structurally equivalent. Graphs G and H may be drawn differently so that they seem different. However, if $G \cong H$ then the isomorphism $f : V(G) \to V(H)$ shows that both of these graphs are fundamentally the same. In particular, the order and size of G are equal to those of H, the isomorphism f preserves adjacencies, and $\deg(v) = \deg(f(v))$ for all $v \in G$. Since f preserves adjacencies, then adjacencies along a given geodesic path are preserved as well. That is, if $v_1, v_2, v_3, \ldots, v_k$ is a shortest path between $v_1, v_k \in V(G)$, then $f(v_1), f(v_2), f(v_3), \ldots, f(v_k)$ is a geodesic path between $f(v_1), f(v_k) \in V(H)$. For example, the two graphs in Figure 1.23 are isomorphic to each other.

Example 1.29. *Consider the graphs in Figure 1.24. Which pair of graphs are isomorphic, and which two graphs are non-isomorphic?*

Solution. If `G` is a Sage graph, one can use the method `G.is_isomorphic()` to determine whether or not the graph `G` is isomorphic to another graph. The following Sage session illustrates how to use `G.is_isomorphic()`.

```
sage: C6 = Graph({"a":["b","c"], "b":["a","d"], "c":["a","e"], \
...   "d":["b","f"], "e":["c","f"], "f":["d","e"]})
sage: G1 = Graph({1:[2,4], 2:[1,3], 3:[2,6], 4:[1,5], \
...   5:[4,6], 6:[3,5]})
sage: G2 = Graph({"a":["d","e"], "b":["c","f"], "c":["b","f"], \
...   "d":["a","e"], "e":["a","d"], "f":["b","c"]})
sage: C6.is_isomorphic(G1)
True
sage: C6.is_isomorphic(G2)
False
sage: G1.is_isomorphic(G2)
False
```

Thus, for the graphs C_6, G_1 and G_2 in Figure 1.24, C_6 and G_1 are isomorphic, but G_1 and G_2 are not isomorphic. ■

An important notion in graph theory is the idea of an "invariant". An *invariant* is an object $f = f(G)$ associated to a graph G which has the property

$$G \cong H \implies f(G) = f(H).$$

For example, the number of vertices of a graph, $f(G) = |V(G)|$, is an invariant.

1.4.1 Adjacency matrices

Two $n \times n$ matrices A_1 and A_2 are *permutation equivalent* if there is a permutation matrix P such that $A_1 = PA_2P^{-1}$. In other words, A_1 is the same as A_2 after a suitable

re-ordering of the rows and a corresponding re-ordering of the columns. This notion of permutation equivalence is an equivalence relation.

To show that two undirected graphs are isomorphic depends on the following result.

Theorem 1.30. *Consider two directed or undirected graphs G_1 and G_2 with respective adjacency matrices A_1 and A_2. Then G_1 and G_2 are isomorphic if and only if A_1 is permutation equivalent to A_2.*

This says that the permutation equivalence class of the adjacency matrix is an invariant.

Define an ordering on the set of $n \times n$ $(0,1)$-matrices as follows: we say $A_1 < A_2$ if the list of entries of A_1 is less than or equal to the list of entries of A_2 in the lexicographical ordering. Here, the list of entries of a $(0,1)$-matrix is obtained by concatenating the entries of the matrix, row-by-row. For example,

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} < \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

Algorithm 1.1 is an immediate consequence of Theorem 1.30. The lexicographically maximal element of the permutation equivalence class of the adjacency matrix of G is called the *canonical label* of G. Thus, to check if two undirected graphs are isomorphic, we simply check if their canonical labels are equal. This idea for graph isomorphism checking is presented in Algorithm 1.1.

Algorithm 1.1: Computing graph isomorphism using canonical labels.

Input: Two undirected simple graphs G_1 and G_2, each having n vertices.
Output: True if $G_1 \cong G_2$; False otherwise.

```
1 for i ← 1, 2 do
2     A_i ← adjacency matrix of G_i
3     p_i ← permutation equivalence class of A_i
4     A'_i ← lexicographically maximal element of p_i
5 if A'_1 = A'_2 then
6     return True
7 return False
```

1.4.2 Degree sequence

Let G be a graph with n vertices. The *degree sequence* of G is the ordered n-tuple of the vertex degrees of G arranged in non-increasing order.

The degree sequence of G may contain the same degrees, repeated as often as they occur. For example, the degree sequence of C_6 is $2, 2, 2, 2, 2, 2$ and the degree sequence of the house graph in Figure 1.3 is $3, 3, 2, 2, 2$. If $n \geq 3$ then the cycle graph C_n has the degree sequence

$$\underbrace{2, 2, 2, \ldots, 2}_{n \text{ copies of } 2}.$$

The path P_n, for $n \geq 3$, has the degree sequence

$$\underbrace{2, 2, 2, \ldots, 2, 1, 1}_{n-2 \text{ copies of } 2}.$$

For positive integer values of n and m, the complete graph K_n has the degree sequence

$$\underbrace{n-1, n-1, n-1, \ldots, n-1}_{n \text{ copies of } n-1}$$

and the complete bipartite graph $K_{m,n}$ has the degree sequence

$$\underbrace{n, n, n, \ldots, n}_{m \text{ copies of } n}, \underbrace{m, m, m, \ldots, m}_{n \text{ copies of } m}.$$

Let S be a non-increasing sequence of non-negative integers. Then S is said to be *graphical* if it is the degree sequence of some graph. If G is a graph with degree sequence S, we say that G *realizes* S.

Let $S = (d_1, d_2, \ldots, d_n)$ be a graphical sequence, i.e. $d_i \geq d_j$ for all $i \leq j$ such that $1 \leq i, j \leq n$. From Corollary 1.11 we see that $\sum_{d_i \in S} d_i = 2k$ for some integer $k \geq 0$. In other words, the sum of a graphical sequence is nonnegative and even. In 1961, Erdős and Gallai [**?**] used this observation as part of a theorem that provides necessary and sufficient conditions for a sequence to be realized by a simple graph. The result is stated in Theorem 1.31, but the original paper of Erdős and Gallai [**?**] does not provide an algorithm to construct a simple graph with a given degree sequence. For a simple graph that has a degree sequence with repeated elements, e.g. the degree sequences of C_n, P_n, K_n, and $K_{m,n}$, it is redundant to verify inequality (1.11) for repeated elements of that sequence. In 2003, Tripathi and Vijay [**?**] showed that one only needs to verify inequality (1.11) for as many times as there are distinct terms in S.

Theorem 1.31. ***Erdős & Gallai 1961 [?].*** *Let $d = (d_1, d_2, \ldots, d_n)$ be a sequence of positive integers such that $d_i \geq d_{i+1}$. Then d is realized by a simple graph if and only if $\sum_i d_i$ is even and*

$$\sum_{i=1}^{k} d_i \leq k(k+1) + \sum_{j=k+1}^{n} \min\{k, d_i\} \tag{1.11}$$

for all $1 \leq k \leq n-1$.

As noted above, Theorem 1.31 is an existence result showing that something exists without providing a construction of the object under consideration. Havel [**?**] and Hakimi [**?**, **?**] independently provided an algorithmic approach that allows for constructing a simple graph with a given degree sequence. See Sierksma and Hoogeveen [**?**] for a coverage of seven criteria for a sequence of integers to be graphic. See Erdős et al. [**?**] for an extension of the Havel-Hakimi theorem to digraphs.

Theorem 1.32. ***Havel 1955 [?] & Hakimi 1962–3 [?, ?].*** *Consider the non-increasing sequence $S_1 = (d_1, d_2, \ldots, d_n)$ of nonnegative integers, where $n \geq 2$ and $d_1 \geq 1$. Then S_1 is graphical if and only if the sequence*

$$S_2 = (d_2 - 1,\ d_3 - 1, \ldots, d_{d_1+1} - 1,\ d_{d_1+2}, \ldots, d_n)$$

is graphical.

Proof. Suppose S_2 is graphical. Let $G_2 = (V_2, E_2)$ be a graph of order $n-1$ with vertex set $V_2 = \{v_2, v_3, \ldots, v_n\}$ such that

$$\deg(v_i) = \begin{cases} d_i - 1, & \text{if } 2 \leq i \leq d_1 + 1, \\ d_i, & \text{if } d_1 + 2 \leq i \leq n. \end{cases}$$

Construct a new graph G_1 with degree sequence S_1 as follows. Add another vertex v_1 to V_2 and add to E_2 the edges v_1v_i for $2 \leq i \leq d_1 + 1$. It is clear that $\deg(v_1) = d_1$ and $\deg(v_i) = d_i$ for $2 \leq i \leq n$. Thus G_1 has the degree sequence S_1.

On the other hand, suppose S_1 is graphical and let G_1 be a graph with degree sequence S_1 such that

(i) The graph G_1 has the vertex set $V(G_1) = \{v_1, v_2, \ldots, v_n\}$ and $\deg(v_i) = d_i$ for $i = 1, \ldots, n$.

(ii) The degree sum of all vertices adjacent to v_1 is a maximum.

To obtain a contradiction, suppose v_1 is not adjacent to vertices having degrees

$$d_2, d_3, \ldots, d_{d_1+1}.$$

Then there exist vertices v_i and v_j with $d_j > d_i$ such that $v_1v_i \in E(G_1)$ but $v_1v_j \notin E(G_1)$. As $d_j > d_i$, there is a vertex v_k such that $v_jv_k \in E(G_1)$ but $v_iv_k \notin E(G_1)$. Replacing the edges v_1v_i and v_jv_k with v_1v_j and v_iv_k, respectively, results in a new graph H whose degree sequence is S_1. However, the graph H is such that the degree sum of vertices adjacent to v_1 is greater than the corresponding degree sum in G_1, contradicting property (ii) in our choice of G_1. Consequently, v_1 is adjacent to d_1 other vertices of largest degree. Then S_2 is graphical because $G_1 - v_1$ has degree sequence S_2. ■

The proof of Theorem 1.32 can be adapted into an algorithm to determine whether or not a sequence of nonnegative integers can be realized by a simple graph. If G is a simple graph, the degree of any vertex in $V(G)$ cannot exceed the order of G. By the handshaking lemma (Theorem 1.9), the sum of all terms in the sequence cannot be odd. Once the sequence passes these two preliminary tests, we then adapt the proof of Theorem 1.32 to successively reduce the original sequence to a smaller sequence. These ideas are summarized in Algorithm 1.2.

Algorithm 1.2: Havel-Hakimi test for sequences realizable by simple graphs.

Input: A nonincreasing sequence $S = (d_1, d_2, \ldots, d_n)$ of nonnegative integers, where $n \geq 2$.

Output: True if S is realizable by a simple graph; False otherwise.

```
1  if ∑_i d_i is odd then
2      return False
3  while True do
4      if min(S) < 0 then
5          return False
6      if max(S) = 0 then
7          return True
8      if max(S) > length(S) − 1 then
9          return False
10     S ← (d_2 − 1, d_3 − 1, ..., d_{d_1+1} − 1, d_{d_1+2}, ..., d_{length(S)})
11     sort S in nonincreasing order
```

We now show that Algorithm 1.2 determines whether or not a sequence of integers is realizable by a simple graph. Our input is a sequence $S = (d_1, d_2, \ldots, d_n)$ arranged

in non-increasing order, where each $d_i \geq 0$. The first test as contained in the if block, otherwise known as a conditional, on line 1 uses the handshaking lemma (Theorem 1.9). During the first run of the while loop, the conditional on line 4 ensures that the sequence S only consists of nonnegative integers. At the conditional on line 6, we know that S is arranged in non-increasing order and has nonnegative integers. If this conditional holds true, then S is a sequence of zeros and it is realizable by a graph with only isolated vertices. Such a graph is simple by definition. The conditional on line 8 uses the following property of simple graphs: If G is a simple graph, then the degree of each vertex of G is less than the order of G. By the time we reach line 10, we know that S has n terms, $\max(S) > 0$, and $0 \leq d_i \leq n-1$ for all $i = 1, 2, \ldots, n$. After applying line 10, S is now a sequence of $n-1$ terms with $\max(S) > 0$ and $0 \leq d_i \leq n-2$ for all $i = 1, 2, \ldots, n-1$. In general, after k rounds of the while loop, S is a sequence of $n-k$ terms with $\max(S) > 0$ and $0 \leq d_i \leq n-k-1$ for all $i = 1, 2, \ldots, n-k$. And after $n-1$ rounds of the while loop, the resulting sequence has one term whose value is zero. In other words, eventually Algorithm 1.2 produces a sequence with a negative term or a sequence of zeros.

1.4.3 Invariants revisited

In some cases, one can distinguish non-isomorphic graphs by considering graph invariants. For instance, the graphs C_6 and G_1 in Figure 1.24 are isomorphic so they have the same number of vertices and edges. Also, G_1 and G_2 in Figure 1.24 are non-isomorphic because the former is connected, while the latter is not connected. To prove that two graphs are non-isomorphic, one could show that they have different values for a given graph invariant. The following list contains some items to check off when showing that two graphs are non-isomorphic:

1. the number of vertices,
2. the number of edges,
3. the degree sequence,
4. the length of a geodesic path,
5. the length of the longest path,
6. the number of connected components of a graph.

1.5 New graphs from old

This section provides a brief survey of operations on graphs to obtain new graphs from old graphs. Such graph operations include unions, products, edge addition, edge deletion, vertex addition, and vertex deletion. Several of these are briefly described below.

1.5.1 Union, intersection, and join

The *disjoint union* of graphs is defined as follows. For two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with disjoint vertex sets, their disjoint union is the graph

$$G_1 \cup G_2 = (V_1 \cup V_2,\ E_1 \cup E_2).$$

For example, Figure 1.25 shows the vertex disjoint union of the complete bipartite graph $K_{1,5}$ with the wheel graph W_4. The adjacency matrix A of the disjoint union of two graphs G_1 and G_2 is the diagonal block matrix obtained from the adjacency matrices A_1 and A_2, respectively. Namely,

$$A = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}.$$

Sage can compute graph unions, as the following example shows.

```
sage: G1 = Graph({1:[2,4], 2:[1,3], 3:[2,6], 4:[1,5], 5:[4,6], 6:[3,5]})
sage: G2 = Graph({7:[8,10], 8:[7,10], 9:[8,12], 10:[7,9], 11:[10,8], 12:[9,7]})
sage: G1u2 = G1.union(G2)
sage: G1u2.adjacency_matrix()
[0 1 0 1 0 0 0 0 0 0 0 0]
[1 0 1 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 0 1 0 0 0 0 0 0]
[0 0 1 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 1 0 1]
[0 0 0 0 0 0 1 0 1 1 1 0]
[0 0 0 0 0 0 0 1 0 1 0 1]
[0 0 0 0 0 0 1 1 1 0 1 0]
[0 0 0 0 0 0 0 1 0 1 0 0]
[0 0 0 0 0 0 1 0 1 0 0 0]
```

In the case where $V_1 = V_2$, then $G_1 \cup G_2$ is simply the graph consisting of all edges in G_1 or in G_2. In general, the union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is defined as

$$G_1 \cup G_2 = (V_1 \cup V_2,\ E_1 \cup E_2)$$

where $V_1 \subseteq V_2$, $V_2 \subseteq V_1$, $V_1 = V_2$, or $V_1 \cap V_2 = \emptyset$. Figure 1.26(c) illustrates the graph union where one vertex set is a proper subset of the other. If $G_1, G_2, \ldots, G_n$ are the components of a graph G, then G is obtained by the disjoint union of its components, i.e. $G = \bigcup G_i$.

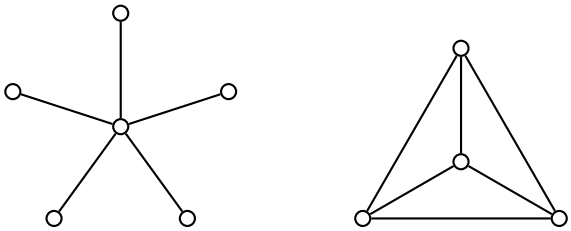
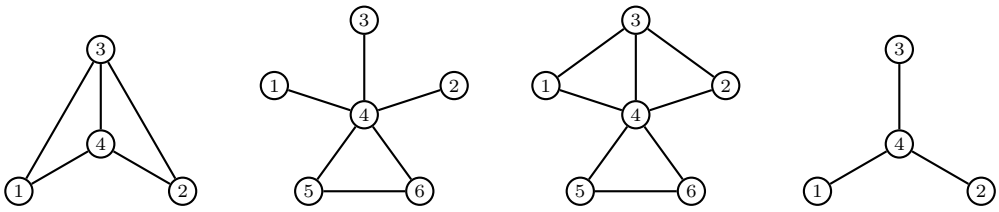


Figure 1.25: The vertex disjoint union $K_{1,5} \cup W_4$.



(a) G_1 (b) G_2 (c) $G_1 \cup G_2$ (d) $G_1 \cap G_2$

Figure 1.26: The union and intersection of graphs with overlapping vertex sets.

The *intersection* of graphs is defined as follows. For two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, their intersection is the graph

$$G_1 \cap G_2 = (V_1 \cap V_2,\ E_1 \cap E_2).$$

Figure 1.26(d) illustrates the intersection of two graphs whose vertex sets overlap.

The *symmetric difference* of graphs is defined as follows. For two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, their symmetric difference is the graph

$$G_1 \Delta G_2 = (V, E)$$

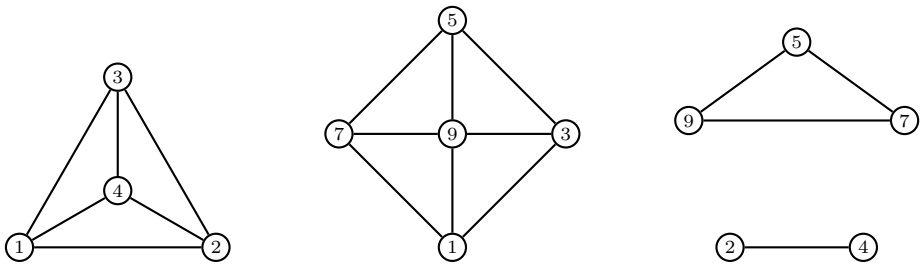
where $V = V_1 \Delta V_2$ and the edge set is given by

$$E = (E_1 \Delta E_2) \backslash \{uv \mid u \in V_1 \cap V_2 \quad \text{or} \quad v \in V_1 \cap V_2\}.$$

Recall that the symmetric difference of two sets S_1 and S_2 is defined by

$$S_1 \Delta S_2 = \{x \in S_1 \cup S_2 \mid x \notin S_1 \cap S_2\}.$$

In the case where $V_1 = V_2$, then $G_1 \Delta G_2$ is simply the empty graph. See Figure 1.27 for an illustration of the symmetric difference of two graphs.



(a) G_1 (b) G_2 (c) $G_1 \Delta G_2$

Figure 1.27: The symmetric difference of graphs.

The *join* of two disjoint graphs G_1 and G_2, denoted $G_1 + G_2$, is their graph union, with each vertex of one graph connecting to each vertex of the other graph. For example, the join of the cycle graph C_{n-1} with a single vertex graph is the *wheel graph* W_n. Figure 1.28 shows various wheel graphs.

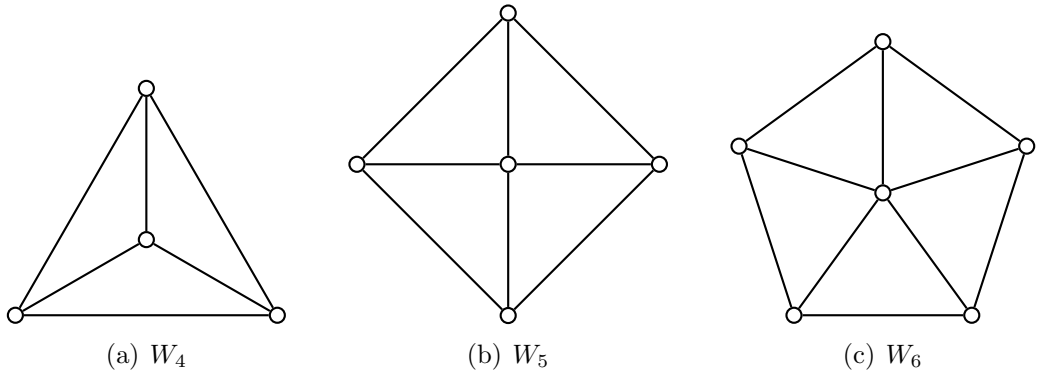
1.5.2 Edge or vertex deletion/insertion

Vertex deletion subgraph

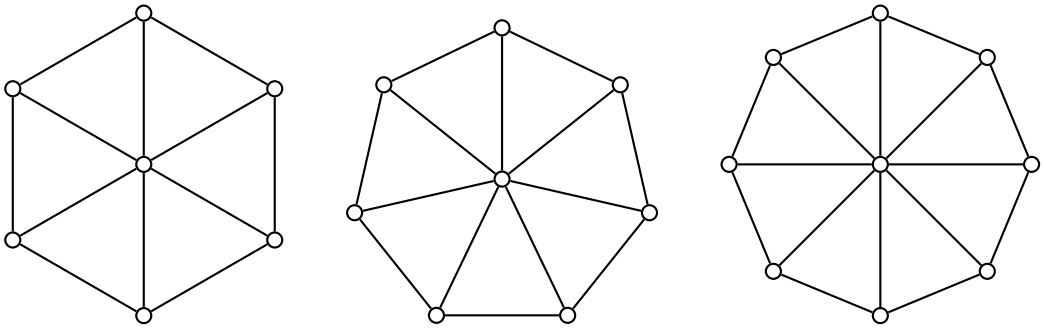
If $G = (V, E)$ is any graph with at least 2 vertices, then the *vertex deletion subgraph* is the subgraph obtained from G by deleting a vertex $v \in V$ and also all the edges incident to that vertex. The vertex deletion subgraph of G is sometimes denoted $G - \{v\}$. Sage can compute vertex deletions, as the following example shows.

```
sage: G = Graph({1:[2,4], 2:[1,4], 3:[2,6], 4:[1,3], 5:[4,2], 6:[3,1]})
sage: G.vertices()
[1, 2, 3, 4, 5, 6]
sage: E1 = Set(G.edges(labels=False)); E1
{(1, 2), (4, 5), (1, 4), (2, 3), (3, 6), (1, 6), (2, 5), (3, 4), (2, 4)}
sage: E4 = Set(G.edges_incident(vertices=[4], labels=False)); E4
{(4, 5), (3, 4), (2, 4), (1, 4)}
sage: G.delete_vertex(4)
sage: G.vertices()
[1, 2, 3, 5, 6]
sage: E2 = Set(G.edges(labels=False)); E2
{(1, 2), (1, 6), (2, 5), (2, 3), (3, 6)}
sage: E1.difference(E2) == E4
True
```

Figure 1.29 presents a sequence of subgraphs obtained by repeatedly deleting vertices. As the figure shows, when a vertex is deleted from a graph, all edges incident on that vertex are deleted as well.

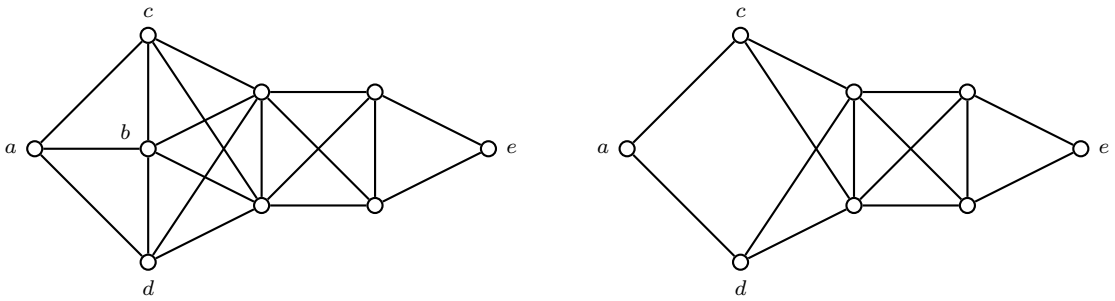


(a) W_4 (b) W_5 (c) W_6

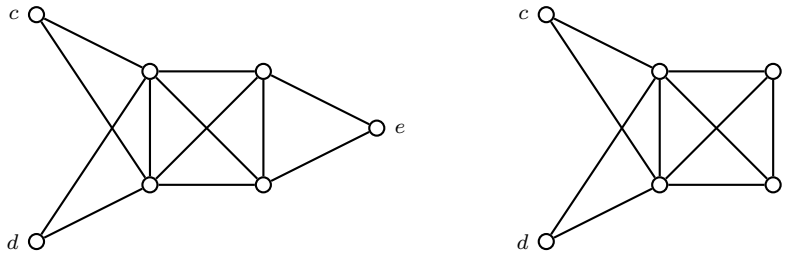


(d) W_7 (e) W_8 (f) W_9

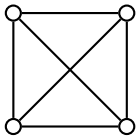
Figure 1.28: The wheel graphs W_n for $n = 4, \ldots, 9$.



(a) G (b) $G - \{b\}$



(c) $G - \{a, b\}$ (d) $G - \{a, b, e\}$



(e) $G - \{a, b, c, d, e\}$

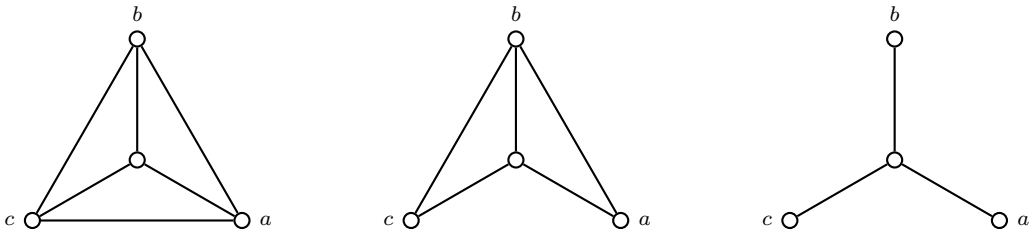
Figure 1.29: Obtaining subgraphs via repeated vertex deletion.

Edge deletion subgraph

If $G = (V, E)$ is any graph with at least 1 edge, then the *edge deletion subgraph* is the subgraph obtained from G by deleting an edge $e \in E$, but not the vertices incident to that edge. The edge deletion subgraph of G is sometimes denoted $G - \{e\}$. Sage can compute edge deletions, as the following example shows.

```
sage: G = Graph({1:[2,4], 2:[1,4], 3:[2,6], 4:[1,3], 5:[4,2], 6:[3,1]})
sage: E1 = Set(G.edges(labels=False)); E1
{(1, 2), (4, 5), (1, 4), (2, 3), (3, 6), (1, 6), (2, 5), (3, 4), (2, 4)}
sage: V1 = G.vertices(); V1
[1, 2, 3, 4, 5, 6]
sage: E14 = Set([(1,4)]); E14
{(1, 4)}
sage: G.delete_edge([1,4])
sage: E2 = Set(G.edges(labels=False)); E2
{(1, 2), (4, 5), (2, 3), (3, 6), (1, 6), (2, 5), (3, 4), (2, 4)}
sage: E1.difference(E2) == E14
True
```

Figure 1.30 shows a sequence of graphs resulting from edge deletion. Unlike vertex deletion, when an edge is deleted the vertices incident on that edge are left intact.



(a) G (b) $G - \{ac\}$ (c) $G - \{ab, ac, bc\}$

Figure 1.30: Obtaining subgraphs via repeated edge deletion.

Vertex cut, cut vertex, or cutpoint

A *vertex cut* (or *separating set*) of a connected graph $G = (V, E)$ is a subset $W \subseteq V$ such that the vertex deletion subgraph $G - W$ is disconnected. In fact, if $v_1, v_2 \in V$ are two non-adjacent vertices, then you can ask for a vertex cut W for which v_1, v_2 belong to different components of $G - W$. Sage's `vertex_cut` method allows you to compute a minimal cut having this property. For many connected graphs, the removal of a single vertex is sufficient for the graph to be disconnected (see Figure 1.30(c)).

Edge cut, cut edge, or bridge

If deleting a single, specific edge would disconnect a graph G, that edge is called a *bridge*. More generally, the *edge cut* (or *disconnecting set* or *seg*) of a connected graph $G = (V, E)$ is a set of edges $F \subseteq E$ whose removal yields an edge deletion subgraph $G - F$ that is disconnected. A minimal edge cut is called a *cut set* or a *bond*. In fact, if $v_1, v_2 \in V$ are two vertices, then you can ask for an edge cut F for which v_1, v_2 belong to different components of $G - F$. Sage's `edge_cut` method allows you to compute a minimal cut having this property. For example, any of the three edges in Figure 1.30(c) qualifies as a bridge and those three edges form an edge cut for the graph in question.

Theorem 1.33. *Let G be a connected graph. An edge $e \in E(G)$ is a bridge of G if and only if e does not lie on a cycle of G.*

Proof. First, assume that $e = uv$ is a bridge of G. Suppose for contradiction that e lies on a cycle

$$C : u, v, w_1, w_2, \ldots, w_k, u.$$

Then $G - e$ contains a u-v path $u, w_k, \ldots, w_2, w_1, v$. Let u_1, v_1 be any two vertices in $G - e$. By hypothesis, G is connected so there is a u_1-v_1 path P in G. If e does not lie on P, then P is also a path in $G - e$ so that u_1, v_1 are connected, which contradicts our assumption of e being a bridge. On the other hand, if e lies on P, then express P as

$$u_1, \ldots, u, v, \ldots, v_1 \quad \text{or} \quad u_1, \ldots, v, u, \ldots, v_1.$$

Now

$$u_1, \ldots, u, w_k, \ldots, w_2, w_1, v, \ldots, v_1 \quad \text{or} \quad u_1, \ldots, v, w_1, w_2, \ldots, w_k, u, \ldots, v_1$$

respectively is a u_1-v_1 walk in $G - e$. By Theorem 1.15, $G - e$ contains a u_1-v_1 path, which contradicts our assumption about e being a bridge.

Conversely, let $e = uv$ be an edge that does not lie on any cycles of G. If $G - e$ has no u-v paths, then we are done. Otherwise, assume for contradiction that $G - e$ has a u-v path P. Then P with uv produces a cycle in G. This cycle contains e, in contradiction of our assumption that e does not lie on any cycles of G. ■

Edge contraction

An *edge contraction* is an operation which, like edge deletion, removes an edge from a graph. However, unlike edge deletion, edge contraction also merges together the two vertices the edge used to connect. For a graph $G = (V, E)$ and an edge $uv = e \in E$, the edge contraction G/e is the graph obtained as follows:

1. Delete the vertices u, v from G.
2. In place of u, v is a new vertex v_e.
3. The vertex v_e is adjacent to vertices that were adjacent to u, v, or both u and v.

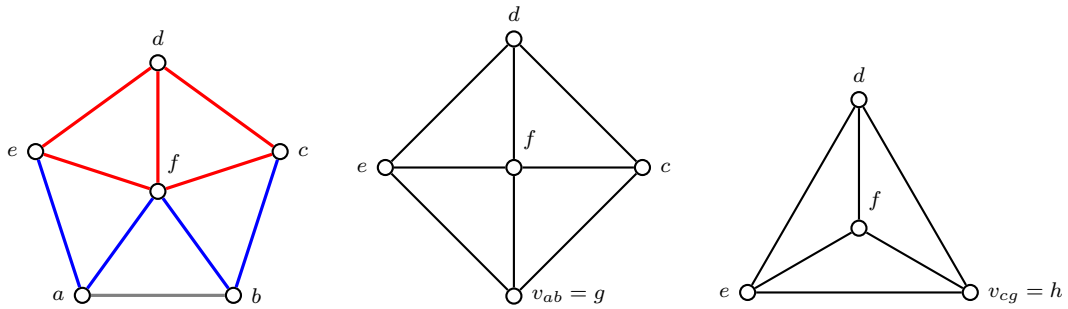
The vertex set of $G/e = (V', E')$ is defined as $V' = \big(V \backslash \{u, v\}\big) \cup \{v_e\}$ and its edge set is

$$E' = \big\{wx \in E \mid \{w, x\} \cap \{u, v\} = \emptyset\big\} \cup \big\{v_e w \mid uw \in E \backslash \{e\} \text{ or } vw \in E \backslash \{e\}\big\}.$$

Make the substitutions

$$E_1 = \big\{wx \in E \mid \{w, x\} \cap \{u, v\} = \emptyset\big\}$$
$$E_2 = \big\{v_e w \mid uw \in E \backslash \{e\} \text{ or } vw \in E \backslash \{e\}\big\}.$$

Let G be the wheel graph W_6 in Figure 1.31(a) and consider the edge contraction G/ab, where ab is the gray colored edge in that figure. Then the edge set E_1 denotes all those edges in G each of which is not incident on a, b, or both a and b. These are precisely those edges that are colored red. The edge set E_2 means that we consider those edges in G each of which is incident on exactly one of a or b, but not both. The blue colored edges in Figure 1.31(a) are precisely those edges that E_2 suggests for consideration. The result of the edge contraction G/ab is the wheel graph W_5 in Figure 1.31(b). Figures 1.31(a) to 1.31(f) present a sequence of edge contractions that starts with W_6 and repeatedly contracts it to the trivial graph K_1.



(a) G_1 (b) $G_2 = G_1/ab$ (c) $G_3 = G_2/cg$

(d) $G_4 = G_3/dh$ (e) $G_5 = G_4/fi$ (f) $G_6 = G_5/ej$

Figure 1.31: Contracting the wheel graph W_6 to the trivial graph K_1.

1.5.3 Complements

The *complement* of a simple graph has the same vertices, but exactly those edges that are not in the original graph. In other words, if $G^c = (V, E^c)$ is the complement of $G = (V, E)$, then two distinct vertices $v, w \in V$ are adjacent in G^c if and only if they are not adjacent in G. We also write the complement of G as $\overline{G}$. The sum of the adjacency matrix of G and that of G^c is the matrix with 1's everywhere, except for 0's on the main diagonal. A simple graph that is isomorphic to its complement is called a *self-complementary graph*. Let H be a subgraph of G. The *relative complement* of G and H is the edge deletion subgraph $G - E(H)$. That is, we delete from G all edges in H. Sage can compute edge complements, as the following example shows.

```
sage: G = Graph({1:[2,4], 2:[1,4], 3:[2,6], 4:[1,3], 5:[4,2], 6:[3,1]})
sage: Gc = G.complement()
sage: EG = Set(G.edges(labels=False)); EG
{(1, 2), (4, 5), (1, 4), (2, 3), (3, 6), (1, 6), (2, 5), (3, 4), (2, 4)}
sage: EGc = Set(Gc.edges(labels=False)); EGc
{(1, 5), (2, 6), (4, 6), (1, 3), (5, 6), (3, 5)}
sage: EG.difference(EGc) == EG
True
sage: EGc.difference(EG) == EGc
True
sage: EG.intersection(EGc)
{}
```

Theorem 1.34. *If $G = (V, E)$ is self-complementary, then the order of G is $|V| = 4k$ or $|V| = 4k + 1$ for some nonnegative integer k. Furthermore, if $n = |V|$ is the order of G, then the size of G is $|E| = n(n-1)/4$.*

Proof. Let G be a self-complementary graph of order n. Each of G and G^c contains half the number of edges in K_n. From (1.6), we have

$$|E(G)| = |E(G^c)| = \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}.$$

Then $n \mid n(n-1)$, with one of n and $n-1$ being even and the other odd. If n is even, $n-1$ is odd so $\gcd(4, n-1) = 1$, hence by [**?**, Theorem 1.9] we have $4 \mid n$ and so $n = 4k$ for some nonnegative $k \in \mathbf{Z}$. If $n-1$ is even, use a similar argument to conclude that $n = 4k+1$ for some nonnegative $k \in \mathbf{Z}$. ■

Theorem 1.35. *A graph and its complement cannot be both disconnected.*

Proof. If G is connected, then we are done. Without loss of generality, assume that G is disconnected and let $\overline{G}$ be the complement of G. Let u, v be vertices in $\overline{G}$. If u, v are in different components of G, then they are adjacent in $\overline{G}$. If both u, v belong to some component C_i of G, let w be a vertex in a different component C_j of G. Then u, w are adjacent in $\overline{G}$, and similarly for v and w. That is, u and v are connected in $\overline{G}$ and therefore $\overline{G}$ is connected. ■

1.5.4 Cartesian product

The *Cartesian product* $G\square H$ of graphs G and H is a graph such that the vertex set of $G\square H$ is the Cartesian product

$$V(G\square H) = V(G) \times V(H).$$

Any two vertices (u, u') and (v, v') are adjacent in $G\square H$ if and only if either

1. $u = v$ and u' is adjacent with v' in H; or
2. $u' = v'$ and u is adjacent with v in G.

The vertex set of $G\square H$ is $V(G\square H)$ and the edge set of $G\square H$ is

$$E(G\square H) = \big(V(G) \times E(H)\big) \cup \big(E(G) \times V(H)\big).$$

Sage can compute Cartesian products, as the following example shows.

```
sage: Z = graphs.CompleteGraph(2); len(Z.vertices()); len(Z.edges())
2
1
sage: C = graphs.CycleGraph(5); len(C.vertices()); len(C.edges())
5
5
sage: P = C.cartesian_product(Z); len(P.vertices()); len(P.edges())
10
15
```

The *path graph* P_n is a tree with n vertices $V = \{v_1, v_2, \ldots, v_n\}$ and edges $E = \{(v_i, v_{i+1}) \mid 1 \le i \le n-1\}$. In this case, $\deg(v_1) = \deg(v_n) = 1$ and $\deg(v_i) = 2$ for $1 < i < n$. The path graph P_n can be obtained from the cycle graph C_n by deleting one edge of C_n. The *ladder graph* L_n is the Cartesian product of path graphs, i.e. $L_n = P_n\square P_1$.

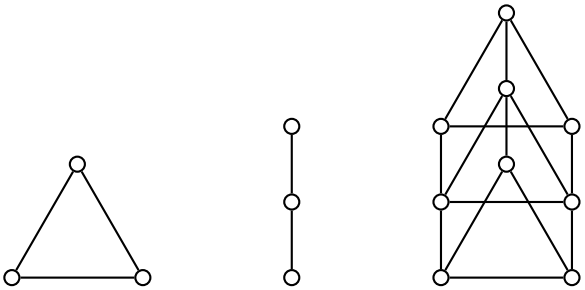
The Cartesian product of two graphs G_1 and G_2 can be visualized as follows. Let $V_1 = \{u_1, u_2, \ldots, u_m\}$ and $V_2 = \{v_1, v_2, \ldots, v_n\}$ be the vertex sets of G_1 and G_2, respectively. Let $H_1, H_2, \ldots, H_n$ be n copies of G_1. Place each H_i at the location of v_i in G_2. Then $u_i \in V(H_j)$ is adjacent to $u_i \in V(H_k)$ if and only if $v_{jk} \in E(G_2)$. See Figure 1.32 for an illustration of obtaining the Cartesian product of K_3 and P_3.

The *hypercube graph* Q_n is the n-regular graph having vertex set

$$V = \big\{(\epsilon_1, \ldots, \epsilon_n) \mid \epsilon_i \in \{0, 1\}\big\}$$

of cardinality 2^n. That is, each vertex of Q_n is a bit string of length n. Two vertices $v, w \in V$ are connected by an edge if and only if v and w differ in exactly one coordinate.[5]

[5] In other words, the *Hamming distance* between v and w is equal to 1.



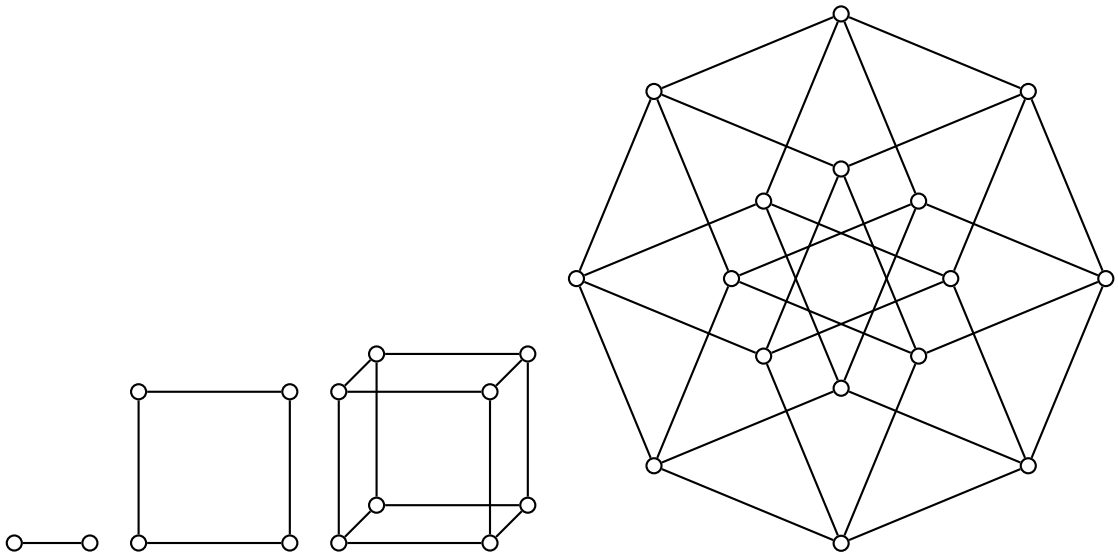
(a) K_3 (b) P_3 (c) $K_3 \square P_3$

Figure 1.32: The Cartesian product of K_3 and P_3.

The Cartesian product of n edge graphs K_2 is a hypercube:

$$(K_2)^{\square n} = Q_n.$$

Figure 1.33 illustrates the hypercube graphs Q_n for $n = 1, \ldots, 4$.



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4

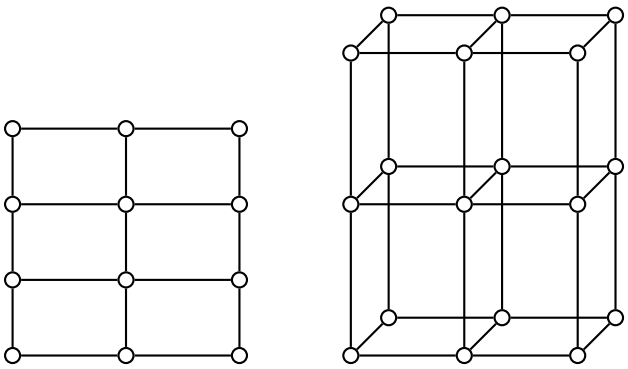
Figure 1.33: Hypercube graphs Q_n for $n = 1, \ldots, 4$.

Example 1.36. The Cartesian product of two hypercube graphs is another hypercube, i.e. $Q_i \square Q_j = Q_{i+j}$. ■

Another family of graphs that can be constructed via Cartesian product is the *mesh*. Such a graph is also referred to as *grid* or *lattice*. The 2-mesh is denoted $M(m, n)$ and is defined as the Cartesian product $M(m, n) = P_m \square P_n$. Similarly, the 3-mesh is defined as $M(k, m, n) = P_k \square P_m \square P_n$. In general, for a sequence $a_1, a_2, \ldots, a_n$ of $n > 0$ positive integers, the n-mesh is given by

$$M(a_1, a_2, \ldots, a_n) = P_{a_1} \square P_{a_2} \square \cdots \square P_{a_n}$$

where the 1-mesh is simply the path graph $M(k) = P_k$ for some positive integer k. Figure 1.34(a) illustrates the 2-mesh $M(3, 4) = P_3 \square P_4$, while the 3-mesh $M(3, 2, 3) = P_3 \square P_2 \square P_3$ is presented in Figure 1.34(b).



(a) $M(3,4)$ (b) $M(3,2,3)$

Figure 1.34: The 2-mesh $M(3,4)$ and the 3-mesh $M(3,2,3)$.

1.5.5 Graph minors

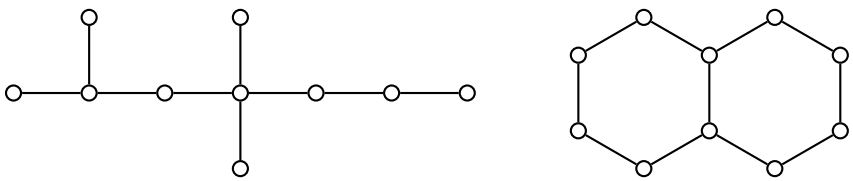
A graph H is called a *minor* of a graph G if H is isomorphic to a graph obtained by a sequence of edge contractions on a subgraph of G. The order in which a sequence of such contractions is performed on G does not affect the resulting graph H. A graph minor is not in general a subgraph. However, if G_1 is a minor of G_2 and G_2 is a minor of G_3, then G_1 is a minor of G_3. Therefore, the relation "being a minor of" is a partial ordering on the set of graphs. For example, the graph in Figure 1.31(c) is a minor of the graph in Figure 1.31(a).

The following non-intuitive fact about graph minors was proven by Neil Robertson and Paul Seymour in a series of 20 papers spanning 1983 to 2004. This result is known by various names including the Robertson-Seymour theorem, the graph minor theorem, or Wagner's conjecture (named after Klaus Wagner).

Theorem 1.37. ***Robertson & Seymour 1983–2004.*** *If an infinite list G_1, G_2, ... of finite graphs is given, then there always exist two indices $i < j$ such that G_i is a minor of G_j.*

Many classes of graphs can be characterized by *forbidden minors*: a graph belongs to the class if and only if it does not have a minor from a certain specified list. We shall see examples of this in Chapter 7.

1.6 Common applications



(a) 2,4,4-trimethylheptane (b) naphthalene

Figure 1.35: Two molecular graphs.

Graph theory, and especially undirected graphs, is used in chemistry to study the structure of molecules. The graph theoretical representation of the structure of a molecule

is called a *molecular graph*; two such examples are illustrated in Figure 1.35. Below we list a few common problems arising in applications of graph theory. See Foulds [**?**] and Walther [**?**] for surveys of applications of graph theory in science, engineering, social sciences, economics, and operation research.

- If the edge weights are all nonnegative, find a "cheapest" closed path which contains all the vertices. This is related to the famous traveling salesman problem and is further discussed in Chapters 2 and 6.

- Find a walk that visits each vertex, but contains as few edges as possible and contains no cycles. This type of problem is related to *spanning trees* and is discussed in further details in Chapter 3.

- Determine which vertices are "more central" than others. This is connected with various applications to *social network analysis* and is covered in more details in Chapters 5 and 10. An example of a social network is shown in Figure 1.36, which illustrates the marriage ties among Renaissance Florentine families [**?**]. Note that one family has been removed because its inclusion would create a disconnected graph.

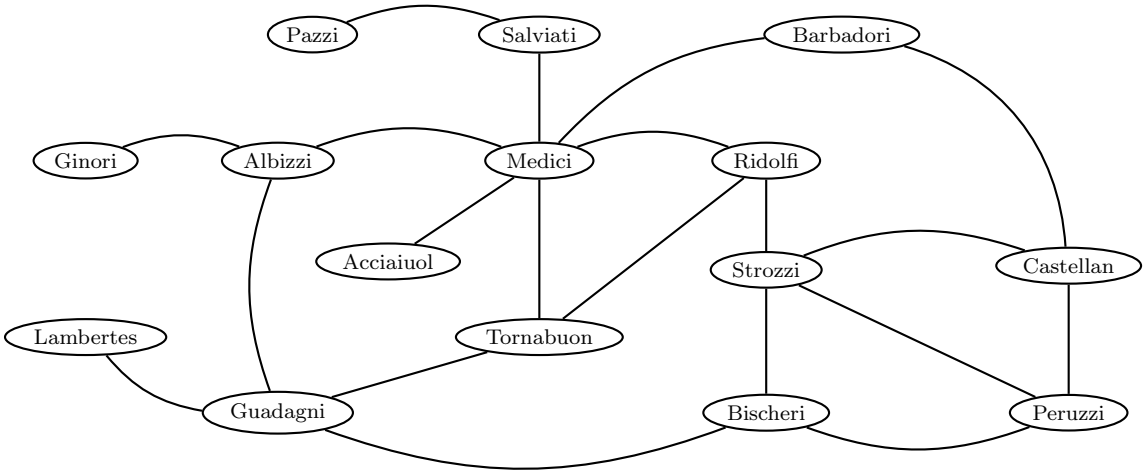


Figure 1.36: Marriage ties among Renaissance Florentine families.

- A *planar graph* is a graph that can be drawn on the plane in such a way that its edges intersect only at their endpoints. Can a graph be drawn entirely in the plane, with no crossing edges? In other words, is a given graph planar? This problem is important for designing computer chips and wiring diagrams. Further discussion is contained in Chapter 7.

- Can you label or color all the vertices of a graph in such a way that no adjacent vertices have the same color? If so, this is called a *vertex coloring*. Can you label or color all the edges of a graph in such a way that no incident edges have the same color? If so, this is called an *edge coloring*. Figure 1.37(a) shows a vertex coloring of the wheel graph W_4 using two colors; Figure 1.37(b) shows a vertex coloring of the Petersen graph using three colors. Graph coloring has several remarkable applications, one of which is to scheduling of jobs relying on a shared resource. This is discussed further in Chapter 8.

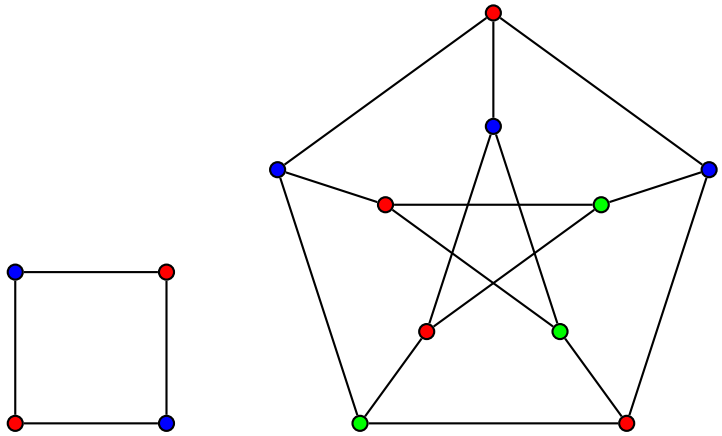


Figure 1.37: Vertex coloring with two and three colors.

- In some fields, such as operations research, a directed graph with nonnegative edge weights is called a *network*, the vertices are called *nodes*, the edges are called *arcs*, and the weight on an edge is called its *capacity*. A *network flow* must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, except when it is a *source node*, which has more outgoing flow, or a *sink node*, which has more incoming flow. The flow along an edge must not exceed the capacity. What is the maximum flow on a network and how to you find it? This problem, which has many industrial applications, is discussed in Chapter 9.

1.7 Application: finite automata

In theoretical computer science, automata are used as idealized mathematical models of computation. The studies of computability (i.e. what can be computed) and complexity (i.e. the time and space requirements of a computation) are based on automata theory to provide precise mathematical models of computers. For an intuitive appreciation of automata, consider a vending machine that dispenses food or beverages. We insert a fixed amount of money into the vending machine and make our choice of food or beverage by pressing buttons that correspond to our choice. If the amount of money inserted is sufficient to cover the cost of our choice of food or beverage, the machine dispenses the item of our choice. Otherwise we need to insert more money until the required amount is reached and then make our selection again. Embodied in the above vending machine example are notions of input (money), machine states (has a selection been made? has sufficient money been inserted?), state transition (move from money insertion state to food/beverage selection state), and output (dispense item of choice).

In the above vending machine example, we should note that the vending machine only accepts a finite number of objects as legitimate input. The vending machine can accept dollar bills and coins of a fixed variety of denominations and belonging to a specific locale, e.g. Australia. Thus we say that the vending machine has finite input and the automaton that models the vending machine is referred to as a finite automaton having a finite alphabet.

δ	20¢	50¢
0¢	20¢	50¢
20¢	40¢	70¢
40¢	60¢	90¢
50¢	70¢	$\geq$ \$1
60¢	80¢	$\geq$ \$1
70¢	90¢	$\geq$ \$1
80¢	$\geq$ \$1	$\geq$ \$1
90¢	$\geq$ \$1	$\geq$ \$1
$\geq$ \$1	$\geq$ \$1	$\geq$ \$1

Table 1.1: Transition table of a simple vending machine.

1.7.1 Automaton and language

Before presenting a precise definition of finite automata, we take a detour to describe notations associated with valid input to finite automata. Let Σ be a nonempty finite alphabet. By Σ^* we mean the set of all finite strings over Σ. Each element of Σ^* is a string or word of finite length whose components are elements of Σ. That is, if $w \in \Sigma^*$ then $w = w_1 w_2 \cdots w_n$ for some integer $n \geq 0$ and each $w_i \in \Sigma$. It follows that $\Sigma \subseteq \Sigma^*$. We also consider the empty string ε as a valid string over Σ. The string ε is sometimes called the null string.

Definition 1.38. ***Finite automata.*** *Let Q and Σ be nonempty finite sets. A* finite automaton *is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where*

1. *Q is a finite set of states.*
2. *Σ is a finite set of input alphabet.*
3. *$\delta : Q \times \Sigma \to Q$ is the transition function.*
4. *$q_0 \in Q$ is the start or initial state.*
5. *$F \subseteq Q$ is the set of accepting or final states.*

For each possible combination of state and input symbol, the transition function δ specifies exactly one subsequent or next state. The finite automaton $\mathcal{A}$ must have at least one initial state, but this lower bound does not necessarily apply to its set of final states. It is possible that the set of final states be empty, in which case $\mathcal{A}$ has no accepting states.

Example 1.39. Figure 1.38 illustrates a finite-automaton representation of a basic vending machine. The initial state is depicted as a circle with an arrow pointing to it, with no other state at the tail of the arrow. The final state is shown as a circle with two concentric rings. We can consider the visual representation in Figure 1.38, also called a *state diagram*, as a multidigraph where each vertex is a state and each directed edge is a transition from one state to another. The state diagram can also be represented in tabular form as shown in Table 1.1. ■

Let $q_1, q_2 \in Q$. The finite-state automaton $\mathcal{A}$ is said to be a *deterministic finite-state automaton* (DFA) if for all $(q, a) \in Q \times \Sigma$, the mappings $(q, a) \mapsto q_1$ and $(q, a) \mapsto q_2$ imply that $q_1 = q_2$. Furthermore, for each state $q \in Q$ and each input symbol $a \in \Sigma$, we have $(q, a) \mapsto q'$ for some $q' \in Q$. In other words $|\delta(q, a)| = 1$.

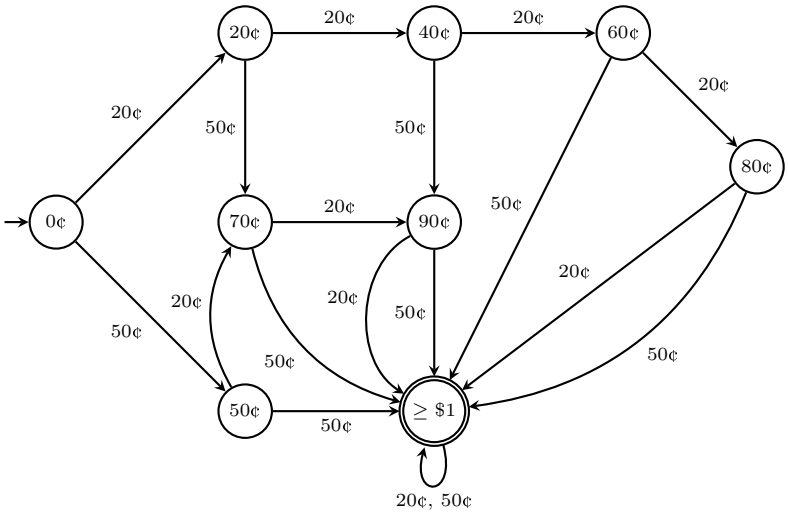


Figure 1.38: State diagram of a simple vending machine.

Definition 1.40. ***Nondeterministic finite-state automata.*** *A* nondeterministic finite-state automaton *(NFA) is a* 5*-tuple* $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ *where*

1. Q *is a finite set of states.*
2. Σ *is a finite set of input alphabet.*
3. δ *is a transition function defined by* $\delta : Q \times \Sigma \to 2^Q$*, where* 2^Q *is the power set of* Q*.*
4. $Q_0 \subseteq Q$ *is a set of initial states.*
5. $F \subseteq Q$ *is a set of accepting or final states.*

Intuitively, $\mathcal{A}$ is said to be an NFA if there exist some $(q, a) \in Q \times \Sigma$ and $q_1, q_2 \in Q$ such that the transitions $(q, a) \mapsto q_1$ and $(q, a) \mapsto q_2$ imply $q_1 \neq q_2$. That is, corresponding to each state/input pair is a multitude of subsequent states. Note the contrast to DFA, where it is required that each state/input pair has at most one subsequent state.

Example 1.41. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be defined by $Q = \{1, 2\}$, $\Sigma = \{a, b\}$, $q_0 = 1$, $F = \{2\}$ and the transition function δ given by

$$\delta(1, a) = 1, \quad \delta(1, b) = 2, \quad \delta(2, a) = 2, \quad \delta(2, b) = 2.$$

Figure 1.39 shows a digraph representation of $\mathcal{A}$. It is easily verifiable by definition that $\mathcal{A}$ is indeed a DFA. ■

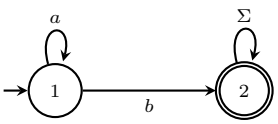


Figure 1.39: A deterministic finite-state automaton.

Example 1.42. Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be defined by $Q = \{1, 2\}$, $\Sigma = \{a, b\}$, $Q_0 = \{1\}$, $F = \{2\}$, and the transition function δ given by

$$\delta(1, a) = 1, \quad \delta(1, a) = 2, \quad \delta(2, a) = 2, \quad \delta(2, b) = 2.$$

Figure 1.40 shows a digraph representation of $\mathcal{A}$. Note that $\delta(1, a) = 1$ and $\delta(1, a) = 2$. It follows by definition that $\mathcal{A}$ is an NFA. ■

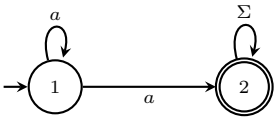


Figure 1.40: A nondeterministic finite-state automaton.

We can inductively define a transition function $\hat{\delta}$ of a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ operating on finite strings over Σ. That is,

$$\hat{\delta} : Q \times \Sigma^* \to Q. \tag{1.12}$$

Let $q \in Q$ and let $s = s_1 s_2 \cdots s_n \in \Sigma^*$. In the case of the empty string, define $\hat{\delta}(q, \varepsilon) = q$. When $i = 1$, we have $\hat{\delta}(q, s_1) = \delta(q, s_1)$. For $1 < i \leq n$, define

$$\hat{\delta}(q,\ s_1 s_2 \cdots s_i) = \hat{\delta}\left(\hat{\delta}(q,\ s_1 s_2 \cdots s_{i-1}),\ s_i\right).$$

For convenience, we write $\delta(q, s)$ instead of $\hat{\delta}(q, s)$. Where $\delta(q_0, s) \in F$, we say that the string s is *accepted* by $\mathcal{A}$. Any subset of Σ^* is said to be a *language* over Σ. The language $\mathcal{L}$ of $\mathcal{A}$ is the set of all finite strings accepted by $\mathcal{A}$, i.e.

$$\mathcal{L}(\mathcal{A}) = \{s \in \Sigma^* \mid \delta(q_0, s) \in F\}.$$

The special language $\mathcal{L}(\mathcal{A})$ is also referred to as a *regular language*. Referring back to example 1.41, any string accepted by $\mathcal{A}$ has zero or more a, followed by exactly one b, and finally zero or more occurrences of a or b. We can describe this language using the regular expression $a^*b(a|b)^*$.

For NFAs, we can similarly define a transition function operating on finite strings. Each input is a string over Σ and the transition function $\hat{\delta}$ returns a subset of Q. Formally, our transition function for NFAs operating on finite strings is the map

$$\hat{\delta} : Q \times \Sigma^* \to 2^Q.$$

Let $q \in Q$ and let $w = xa$, where $x \in \Sigma^*$ and $a \in \Sigma$. The input symbol a can be interpreted as being the very last symbol in the string w. Then x is interpreted as being the substring of w excluding the symbol a. In the case of the empty string, we have $\hat{\delta}(q, \varepsilon) = \{q\}$. For the inductive case, assume that $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$ where each $p_i \in Q$. Then $\hat{\delta}(q, w)$ is defined by

$$\begin{aligned}\hat{\delta}(q, w) &= \hat{\delta}\left(\hat{\delta}(q, x),\ a\right) \\ &= \hat{\delta}\left(\{p_1, p_2, \ldots, p_k\}, a\right) \\ &= \bigcup_{i=1}^{k} \delta(p_i, a).\end{aligned}$$

It may happen that for some state p_i, there are no transitions from p_i with input a. We cater for this possibility by writing $\delta(p_i, a) = \emptyset$.

1.7.2 Simulating NFAs using DFAs

Any NFA can be simulated by a DFA. One way of accomplishing this is to allow the DFA to keep track of all the states that the NFA can be in after reading an input symbol. The formal proof depends on this construction of an equivalent DFA and then showing that the language of the DFA is the same as that of the NFA.

Theorem 1.43. ***Determinize an NFA.*** *If $\mathcal{A}$ is a nondeterministic finite-state automaton, then there exists a deterministic finite-state automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.*

Proof. Let the NFA $\mathcal{A}$ be defined by $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and define a DFA $\mathcal{A}' = (Q', \Sigma, \delta', q_0', F')$ as follows. The state space of $\mathcal{A}'$ is the power set of Q, i.e. $Q' = 2^Q$. The accepting state space F' of $\mathcal{A}'$ is a subset of Q', where each $f \in F'$ is a set containing at least an accepting state of $\mathcal{A}$. In symbols, we write $F' \subseteq Q'$ where

$$F' = \{q \in Q' \mid p \in F \text{ for some } p \in q\}.$$

Denote each element $q \in Q'$ by $q = [q_1, q_2, \ldots, q_i]$ where $q_1, q_2, \ldots, q_i \in Q$. Thus the initial state of $\mathcal{A}'$ is $q_0' = [Q_0]$. Now define the transition function δ' by

$$\begin{aligned} &\delta'([q_1, q_2, \ldots, q_i], s) = [p_1, p_2, \ldots, p_j] \\ \iff \quad &\delta(\{q_1, q_2, \ldots, q_i\}, s) = \bigcup_{k=1}^{i} \delta(q_k, s) = \{p_1, p_2, \ldots, p_j\}. \end{aligned} \tag{1.13}$$

For any input string w, we now show by induction on the length of w that

$$\begin{aligned} &\delta'(q_0', w) = [q_1, q_2, \ldots, q_i] \\ \iff \quad &\delta(Q_0, w) = \{q_1, q_2, \ldots, q_i\}. \end{aligned} \tag{1.14}$$

For the basis step, let $|w| = 0$ so that $w = \varepsilon$. Then it is clear that

$$\begin{aligned} &\delta'(q_0', w) = \delta'(q_0', \varepsilon) = [q_0'] \\ \iff \quad &\delta(Q_0, w) = \delta(Q_0, \varepsilon) = Q_0. \end{aligned}$$

Next, assume for induction that statement (1.14) holds for all strings of length less than or equal to $m > 0$. Let w be a string of length m and let $a \in \Sigma$ so that $|wa| = m + 1$. Then $\delta'(q_0', wa) = \delta'\big(\delta'(q_0', w),\, a\big)$. By our inductive hypothesis, we have

$$\begin{aligned} &\delta'(q_0', w) = [p_1, p_2, \ldots, p_j] \\ \iff \quad &\delta(Q_0, w) = \{p_1, p_2, \ldots, p_j\} \end{aligned}$$

and applying (1.14) we get

$$\begin{aligned} &\delta'([p_1, p_2, \ldots, p_j], a) = [r_1, r_2, \ldots, r_k] \\ \iff \quad &\delta(\{p_1, p_2, \ldots, p_j\}, a) = \{r_1, r_2, \ldots, r_k\}. \end{aligned}$$

Hence

$$\begin{aligned} &\delta'(q_0', wa) = [r_1, r_2, \ldots, r_k] \\ \iff \quad &\delta(Q_0, wa) = \{r_1, r_2, \ldots, r_k\} \end{aligned}$$

which establishes that statement (1.14) holds for all finite strings over Σ. Finally, $\delta'(q_0', w) \in F'$ if and only if there is some $p \in \delta(Q_0, w)$ such that $p \in F$. Therefore $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. ∎

Theorem 1.43 tells us that any NFA corresponds to some DFA that accepts the same language. For this reason, the theorem is said to provide us with a procedure for *determinizing* NFAs. The actual procedure itself is contained in the proof of the theorem, although it must be noted that the procedure is inefficient since it potentially yields transitions from states that are unreachable from the initial state. If $q \in Q'$ is a state of $\mathcal{A}'$ that is unreachable from q_0', then there are no input strings w such that $\delta'(q_0', w) = q$. Such unreachable states are redundant insofar as they do not affect $\mathcal{L}(\mathcal{A}')$.

Another inefficiency of the procedure in the proof of Theorem 1.43 is the problem of state space explosion. As $Q' = 2^Q$ is the power set of Q, the resulting DFA can potentially have exponentially more states than the NFA it is simulating. In the worse case, each element of Q' is a state of the resulting DFA that is reachable from $q_0' = [Q_0]$. The best-case scenario is when each state of the DFA is a singleton, hence the DFA has the same number of states as its corresponding NFA. However, according to the procedure in the proof of Theorem 1.43, we generate all the possible 2^n states of the DFA, where $n = |Q|$. After considering all the transitions whose starting states are singletons, we then consider all transitions starting from each of the remaining $2^n - n$ elements in Q'. In the best-case, none of those remaining $2^n - n$ states are reachable from q_0', hence it is redundant to generate transitions starting at each of those $2^n - n$ states. Example 1.44 concretizes our discussion.

Example 1.44. *Use the procedure in Theorem 1.43 to determinize the NFA in Figure 1.41.*

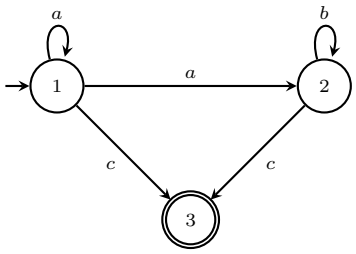


Figure 1.41: An NFA with 3 states and 3 input symbols.

Solution. The NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ in Figure 1.41 has the states $Q = \{1, 2, 3\}$, the initial state $q_0 = 1$, the final state set $F = \{3\}$, and the input alphabet $\Sigma = \{a, b, c\}$. Its transitions are contained in Table 1.2. To determinize $\mathcal{A}$, we construct a DFA $\mathcal{A}' =$

δ	a	b	c
1	$\{1, 2\}$	$\emptyset$	$\{3\}$
2	$\emptyset$	$\{2\}$	$\{3\}$
3	$\emptyset$	$\emptyset$	$\emptyset$

Table 1.2: Transition table for the NFA in Figure 1.41.

$(Q', \Sigma, \delta', q_0', F')$. As Q' is the power set of Q, then all the possible states of $\mathcal{A}'$ are contained in $Q' = \{\emptyset, [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]\}$. The alphabet of $\mathcal{A}'$ is the same as the alphabet of $\mathcal{A}$, namely Σ. The initial state of $\mathcal{A}'$ is $q_0' = [q_0] = [1]$. All the possible accepting states of $\mathcal{A}'$ are contained in $F' = \{[3], [1, 3], [2, 3], [1, 2, 3]\}$. Next,

we apply (1.13) to construct all the possible transitions of $\mathcal{A}'$. These transitions are contained in Table 1.3. Using those transitions, we obtain the digraph representation in Figure 1.42, from which it is clear that the states [1], [2], [3], and $[1,2]$ are the only states

δ'	a	b	c
$[1]$	$[1,2]$	$\emptyset$	$[3]$
$[2]$	$\emptyset$	$[2]$	$[3]$
$[3]$	$\emptyset$	$\emptyset$	$\emptyset$
$[1,2]$	$[1,2]$	$[2]$	$[3]$
$[1,3]$	$[1,2]$	$\emptyset$	$[3]$
$[2,3]$	$\emptyset$	$[2]$	$[3]$
$[1,2,3]$	$[1,2]$	$[2]$	$[3]$

Table 1.3: Transition table of a deterministic version of the NFA in Figure 1.41.

reachable from the initial state $q_0' = [1]$. The remaining states $[1,3]$, $[2,3]$, and $[1,2,3]$ are not reachable from $q_0' = [1]$. In other words, starting at $q_0' = [1]$ there are no input strings that would result in a transition to any of $[1,3]$, $[2,3]$, and $[1,2,3]$. Therefore these states, and the transitions starting from them, can be deleted from Figure 1.42 without affecting the language of $\mathcal{A}'$. Figure 1.43 shows an equivalent DFA with redundant states removed. ■

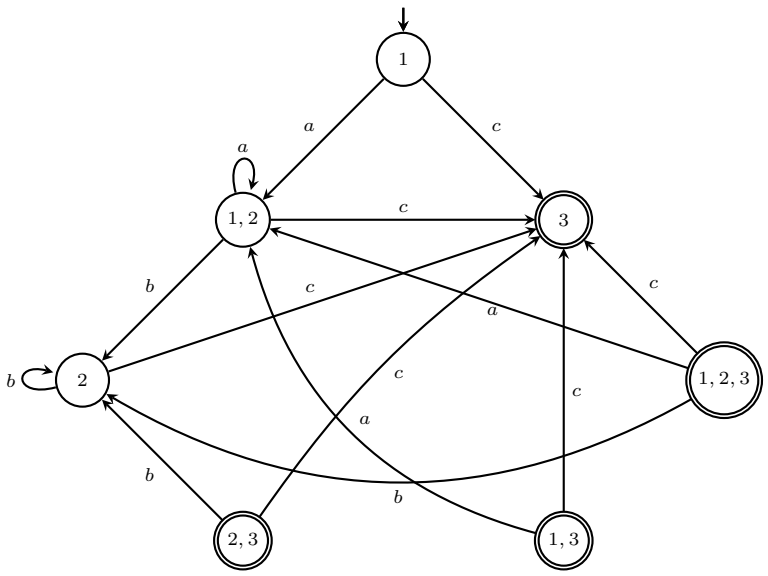


Figure 1.42: A DFA accepting the same language as the NFA in Figure 1.41.

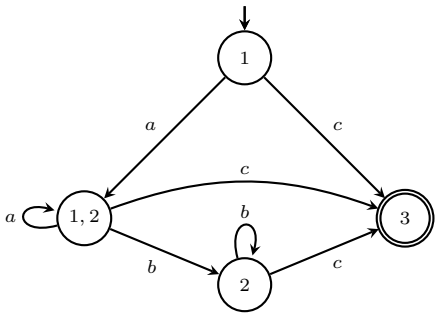


Figure 1.43: A DFA equivalent to that in Figure 1.42, with redundant states removed.

1.8 Problems

> A problem left to itself dries up or goes rotten. But fertilize a problem with a solution—you'll hatch out dozens.
> — N. F. Simpson, *A Resounding Tinkle*, 1958

1.1. For each graph in Figure 1.7, do the following:

(a) Construct the graph using Sage.

(b) Find its adjacency matrix.

(c) Find its node and edge sets.

(d) How many nodes and edges are in the graph?

(e) If applicable, find all of each node's in-coming and out-going edges. Hence find the node's indegree and outdegree.

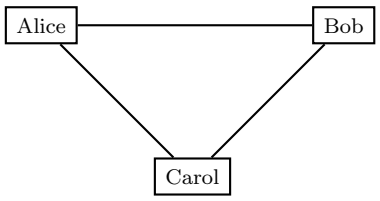


Figure 1.44: Graph representation of a social network.

1.2. In the friendship network of Figure 1.44, Carol is a mutual friend of Alice and Bob. How many possible ways are there to remove exactly one edge such that, in the resulting network, Carol is no longer a mutual friend of Alice and Bob?

1.3. The routing network of German cities in Figure 1.45 shows that each pair of distinct cities are connected by a flight path. The weight of each edge is the flight distance in kilometers between the two corresponding cities. In particular, there is a flight path connecting Karlsruhe and Stuttgart. What is the shortest route between Karlsruhe and Stuttgart? Suppose we can remove at least one edge from this network. How many possible ways are there to remove edges such that, in the resulting network, Karlsruhe is no longer connected to Stuttgart via a flight path?

1.4. Let $D = (V, E)$ be a digraph of size q. Show that

$$\sum_{v \in V} \mathrm{id}(v) = \sum_{v \in V} \mathrm{od}(v) = q.$$

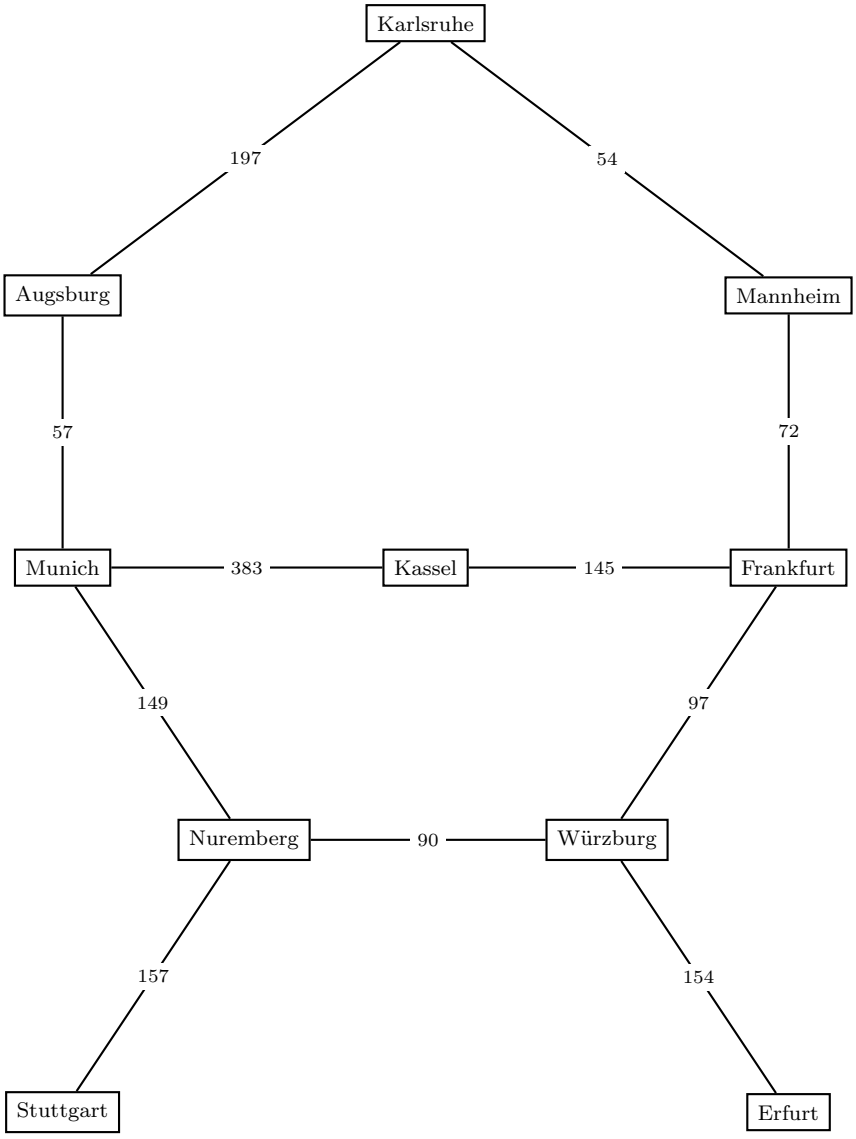


Figure 1.45: Graph representation of a routing network.

1.5. If G is a simple graph of order $n > 0$, show that $\deg(v) < n$ for all $v \in V(G)$.

1.6. Let G be a graph of order n and size m. Then G is called an *overfull graph* if $m > \Delta(G) \cdot \lfloor n/2 \rfloor$. If $m = \Delta(G) \cdot \lfloor n/2 \rfloor + 1$, then G is said to be *just overfull*. It can be shown that overfull graphs have odd order. Equivalently, let G be of odd order. We can define G to be overfull if $m > \Delta(G) \cdot (n-1)/2$, and G is just overfull if $m = \Delta(G) \cdot (n-1)/2 + 1$. Find an overfull graph and a graph that is just overfull. Some basic results on overfull graphs are presented in [**?**].

1.7. Fix a positive integer n and denote by $\Gamma(n)$ the number of simple graphs on n vertices. Show that

$$\Gamma(n) = 2^{\binom{n}{2}} = 2^{n(n-1)/2}.$$

1.8. Let G be an undirected graph whose unoriented incidence matrix is M_u and whose oriented incidence matrix is M_o.

(a) Show that the sum of the entries in any row of M_u is the degree of the corresponding vertex.

(b) Show that the sum of the entries in any column of M_u is equal to 2.

(c) If G has no self-loops, show that each column of M_o sums to zero.

1.9. Let G be a loopless digraph and let M be its incidence matrix.

(a) If r is a row of M, show that the number of occurrences of -1 in r counts the outdegree of the vertex corresponding to r. Show that the number of occurrences of 1 in r counts the indegree of the vertex corresponding to r.

(b) Show that each column of M sums to 0.

1.10. Let G be a digraph and let M be its incidence matrix. For any row r of M, let m be the frequency of -1 in r, let p be the frequency of 1 in r, and let t be twice the frequency of 2 in r. If v is the vertex corresponding to r, show that the degree of v is $\deg(v) = m + p + t$.

1.11. Let G be an undirected graph without self-loops and let M and its oriented incidence matrix. Show that the Laplacian matrix $\mathcal{L}$ of G satisfies $\mathcal{L} = M \times M^T$, where M^T is the transpose of M.

1.12. Let J_1 denote the incidence matrix of G_1 and let J_2 denote the incidence matrix of G_2. Find matrix theoretic criteria on J_1 and J_2 which hold if and only if $G_1 \cong G_2$. In other words, find the analog of Theorem 1.30 for incidence matrices.

1.13. Show that the complement of an edgeless graph is a complete graph.

1.14. Let $G \square H$ be the Cartesian product of two graphs G and H. Show that $|E(G \square H)| = |V(G)| \cdot |E(H)| + |E(G)| \cdot |V(H)|$.

1.15. In 1751, Leonhard Euler posed a problem to Christian Goldbach, a problem that now bears the name "Euler's polygon division problem". Given a plane convex polygon having n sides, how many ways are there to divide the polygon into triangles using only diagonals? For our purposes, we consider only regular polygons having n sides for $n \geq 3$ and any two diagonals must not cross each other. For

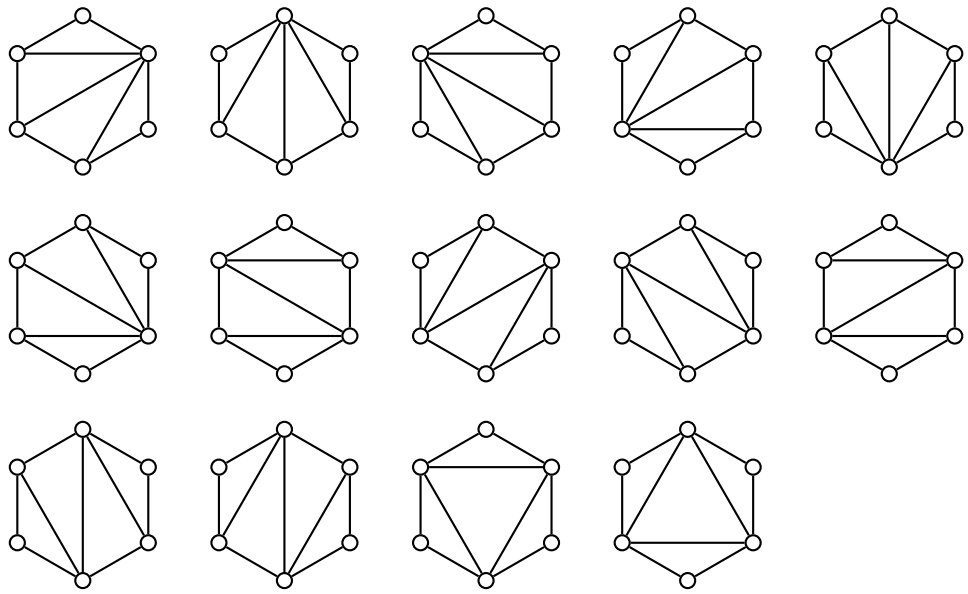


Figure 1.46: Euler's polygon division problem for the hexagon.

example, the triangle is a regular 3-gon, the square a regular 4-gon, the pentagon a regular 5-gon, etc. In the case of the hexagon considered as the cycle graph C_6, there are 14 ways to divide it into triangles, as shown in Figure 1.46, resulting in 14 graphs. However, of those 14 graphs only 3 are nonisomorphic to each other.

(a) What is the number of ways to divide a pentagon into triangles using only diagonals? List all such divisions. If each of the resulting so divided pentagons is considered a graph, how many of those graphs are nonisomorphic to each other?

(b) Repeat the above exercise for the heptagon.

(c) Let E_n be the number of ways to divide an n-gon into triangles using only diagonals. For $n \geq 1$, the *Catalan numbers* C_n are defined as

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Dörrie [**?**, pp.21–27] showed that E_n is related to the Catalan numbers via the equation $E_n = C_{n-1}$. Show that

$$C_n = \frac{1}{4n+2}\binom{2n+2}{n+1}.$$

For $k \geq 2$, show that the Catalan numbers satisfy the recurrence relation

$$C_k = \frac{4k-2}{k+1}C_{k-1}.$$

1.16. A graph is said to be *planar* if it can be drawn on the plane in such a way that no two edges cross each other. For example, the complete graph K_n is planar for $n = 1, 2, 3, 4$, but K_5 is not planar (see Figure 1.13). Draw a planar version of K_4 as presented in Figure 1.13(b). Is the graph in Figure 1.9 planar? For $n = 1, 2, \ldots, 5$, enumerate all simple nonisomorphic graphs on n vertices that are planar; only work with undirected graphs.

1.17. If $n \geq 3$, show that the join of C_n and K_1 is the wheel graph W_{n+1}. In other words, show that $C_n + K_1 = W_{n+1}$.

1.18. A common technique for generating "random" numbers is the linear congruential method, a generalization of the Lehmer generator [**?**] introduced in 1949. First, we choose four integers:

$$\begin{array}{lll} m, & \text{modulus}, & 0 < m \\ a, & \text{multiplier}, & 0 \leq a < m \\ c, & \text{increment}, & 0 \leq c < m \\ X_0, & \text{seed}, & 0 \leq X_0 < m \end{array}$$

where the value X_0 is also referred to as the starting value. Then iterate the relation

$$X_{n+1} = (aX_n + c) \bmod m, \qquad n \geq 0$$

and halt when the relation produces the seed X_0 or when it produces an integer X_k such that $X_k = X_i$ for some $0 \leq i < k$. The resulting sequence

$$S = (X_0, X_1, \ldots, X_n)$$

is called a *linear congruential sequence*. Define a graph theoretic representation of S as follows: let the vertex set be $V = \{X_0, X_1, \ldots, X_n\}$ and let the edge set be $E = \{X_iX_{i+1} \mid 0 \leq i < n\}$. The resulting graph $G = (V, E)$ is called the *linear congruential graph* of the linear congruential sequence S. See chapter 3 of [**?**] for other techniques for generating "random" numbers.

(a) Compute the linear congruential sequences S_i with the following parameters:
   (i) S_1: $m = 10$, $a = c = X_0 = 7$
   (ii) S_2: $m = 10$, $a = 5$, $c = 7$, $X_0 = 0$
   (iii) S_3: $m = 10$, $a = 3$, $c = 7$, $X_0 = 2$
   (iv) S_4: $m = 10$, $a = 2$, $c = 5$, $X_0 = 3$

(b) Let G_i be the linear congruential graph of S_i. Draw each of the graphs G_i. Draw the graph resulting from the union

$$\bigcup_i G_i.$$

(c) Let m, a, c, and X_0 be the parameters of a linear congruential sequence where
   (i) c is relatively prime to m;
   (ii) $b = a - 1$ is a multiple of p for each prime p that divides m; and
   (iii) 4 divides b if 4 divides m.

   Show that the corresponding linear congruential graph is the wheel graph W_m on m vertices.

1.19. We want to generate a random bipartite graph whose first and second partitions have n_1 and n_2 vertices, respectively. Describe and present pseudocode to generate the required random bipartite graph. What is the worst-case runtime of your algorithm? Modify your algorithm to account for a third parameter m that specifies the number of edges in the resulting bipartite graph.

1.20. Describe and present pseudocode to generate a random regular graph. What is the worst-case runtime of your algorithm?

1.21. The Cantor-Schröder-Bernstein theorem states that if A, B are sets and we have an injection $f : A \to B$ and an injection $g : B \to A$, then there is a bijection between A and B, thus proving that A and B have the same cardinality. Here we use bipartite graphs and other graph theoretic concepts to prove the Cantor-Schröder-Bernstein theorem. The full proof can be found in [**?**].

(a) Is it possible for A and B to be bipartitions of V and yet satisfy $A \cap B \neq \emptyset$?

(b) Now assume that $A \cap B = \emptyset$ and define a bipartite graph $G = (V, E)$ with A and B being the two partitions of V, where for any $x \in A$ and $y \in B$ we have $xy \in E$ if and only if either $f(x) = y$ or $g(y) = x$. Show that $\deg(v) = 1$ or $\deg(v) = 2$ for each $v \in V$.

(c) Let C be a component of G and let $A' \subseteq A$ and $B' \subseteq B$ contain all vertices in the component C. Show that $|A'| = |B'|$.

1.22. Fermat's little theorem states that if p is prime and a is an integer not divisible by p, then p divides $a^p - a$. Here we cast the problem within the context of graph theory and prove it using graph theoretic concepts. The full proof can be found in [**?**, **?**].

(a) Let $G = (V, E)$ be a graph with V being the set of all sequences $(a_1, a_2, \ldots, a_p)$ of integers $1 \leq a_i \leq a$ and $a_j \neq a_k$ for some $j \neq k$. Show that G has $a^p - a$ vertices.

(b) Define the edge set of G as follows. If $u, v \in V$ such that $u = (u_1, u_2, \ldots, u_p)$ and $v = (u_p, u_1, \ldots, u_{p-1})$, then $uv \in E$. Show that each component of G is a cycle of length p.

(c) Show that G has $(a^p - a)/p$ components.

1.23. For the finite automaton in Figure 1.38, identify the following:

(a) The states set Q.

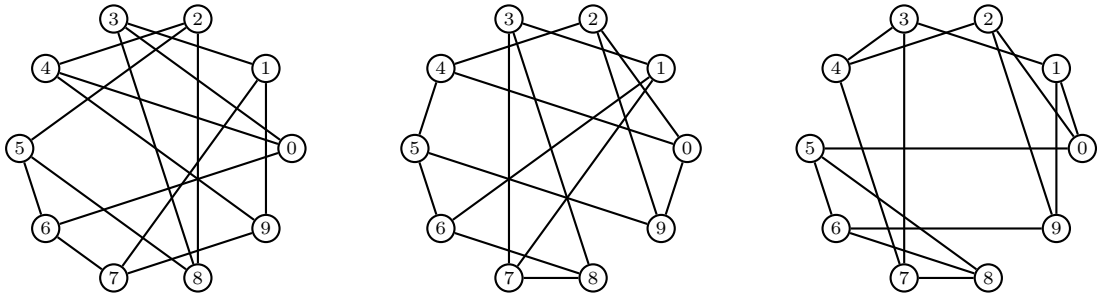
(b) The alphabet set Σ.

(c) The transition function $\delta : Q \times \Sigma \to Q$.

(d) The initial state $q_0 \in Q$.

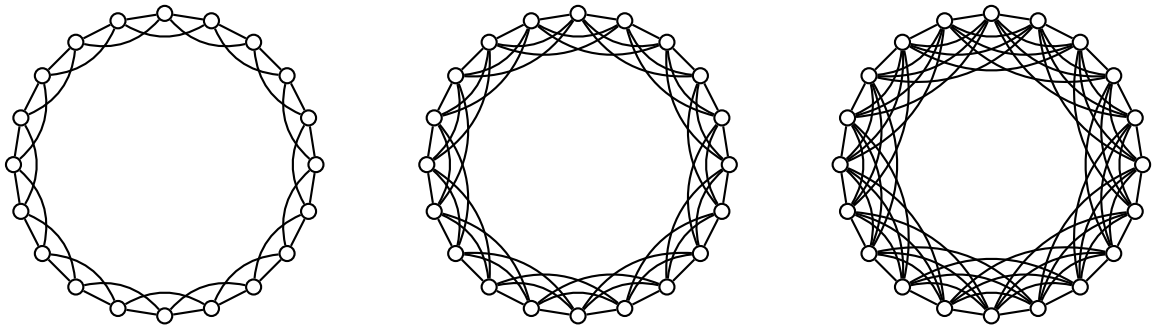
(e) The set of final states $F \subseteq Q$.

1.24. The cycle graph C_n is a 2-regular graph. If $2 < r < n/2$, unlike the cycle graph there are various realizations of an r-regular graph; see Figure 1.47 for the case of $r = 3$ and $n = 10$. The k-circulant graph on n vertices can be considered as an intermediate graph between C_n and a k-regular graph. Let k and n be positive integers satisfying $k < n/2$ with k being even. Suppose $G = (V, E)$ is a simple undirected graph with vertex set $V = \{0, 1, \ldots, n-1\}$. Define the edge set of G as follows. Each $i \in V$ is incident with each of $i + j \bmod n$ and $i - j \bmod n$ for $j \in \{1, 2, \ldots, k/2\}$. With the latter edge set, G is said to be a *k-circulant graph*, a type of graphs used in constructing small-world networks (see section 10.4). Refer to Figure 1.48 for examples of k-circulant graphs.



(a) (b) (c)

Figure 1.47: Various 3-regular graphs on 10 vertices.



(a) $k = 4$ (b) $k = 6$ (c) $k = 8$

Figure 1.48: Various k-circulant graphs for $k = 4, 6, 8$.

(a) Describe and provide pseudocode of an algorithm to construct a k-circulant graph on n vertices.

(b) Show that the cycle graph C_n is 2-circulant.

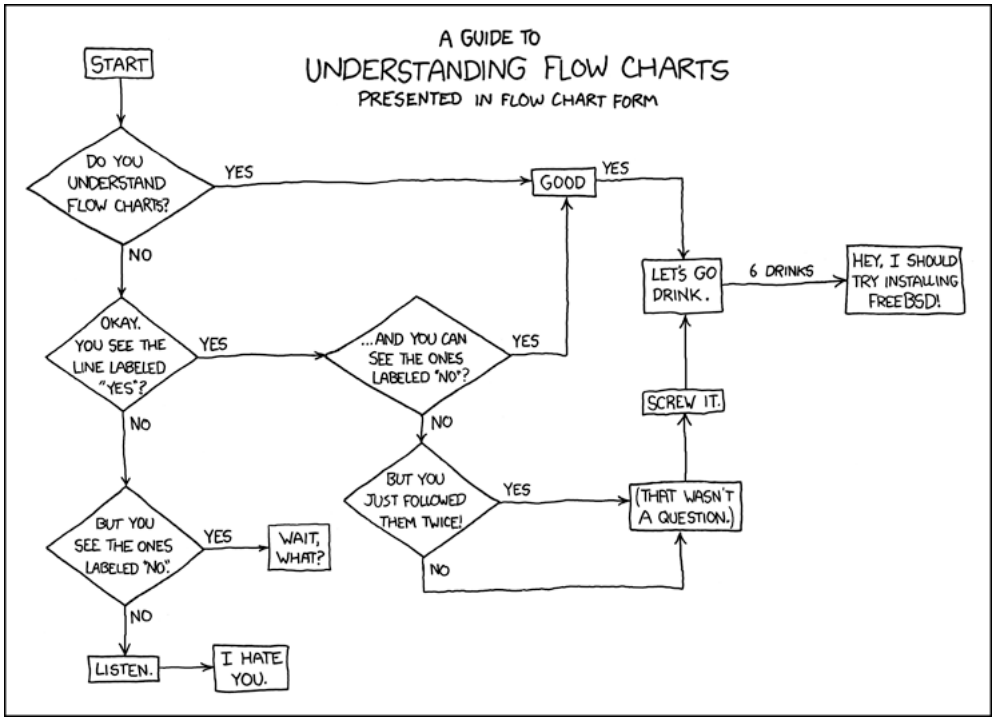
(c) Show that the sum of all degrees of a k-circulant graph on n vertices is nk.

(d) Show that a k-circulant graph is k-regular.

(e) Let $\mathcal{C}$ be the collection of all k-regular graphs on n vertices. If each k-regular graph from $\mathcal{C}$ is equally likely to be chosen, what is the probability that a k-circulant graph be chosen from $\mathcal{C}$?

Chapter 2

Graph algorithms



— Randall Munroe, xkcd, http://xkcd.com/518/

Graph algorithms have many applications. Suppose you are a salesman with a product you would like to sell in several cities. To determine the cheapest travel route from city-to-city, you must effectively search a graph having weighted edges for the "cheapest" route visiting each city once. Each vertex denotes a city you must visit and each edge has a weight indicating either the distance from one city to another or the cost to travel from one city to another.

Shortest path algorithms are some of the most important algorithms in algorithmic graph theory. In this chapter, we first examine several common graph traversal algorithms and some basic data structures underlying these algorithms. A data structure is a combination of methods for structuring a collection of data (e.g. vertices and edges) and protocols for accessing the data. We then consider a number of common shortest path algorithms, which rely in one way or another on graph traversal techniques and basic data structures for organizing and managing vertices and edges.

2.1 Representing graphs in a computer

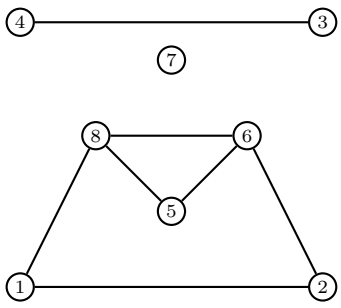
> To err is human but to really foul things up requires a computer.
> — Anonymous, *Farmers' Almanac for 1978*, "Capsules of Wisdom"

In section 1.3, we discussed how to use matrices for representing graphs and digraphs. If $A = [a_{ij}]$ is an $m \times n$ matrix, the adjacency matrix representation of a graph would require representing all the mn entries of A. Alternative graph representations exist that are much more efficient than representing all entries of a matrix. The graph representation used can be influenced by the size of a graph or the purpose of the representation. Section 2.1.1 discusses the adjacency list representation that can result in less storage space requirement than the adjacency matrix representation. The `graph6` format discussed in section 2.1.3 provides a compact means of storing graphs for archival purposes.

2.1.1 Adjacency lists

A *list* is a sequence of objects. Unlike sets, a list may contain multiple copies of the same object. Each object in a list is referred to as an *element* of the list. A list L of $n \geq 0$ elements is written as $L = [a_1, a_2, \ldots, a_n]$, where the i-th element a_i can be indexed as $L[i]$. In case $n = 0$, the list $L = [\,]$ is referred to as the *empty list*. Two lists are equivalent if they both contain the same elements at exactly the same positions.

Define the adjacency lists of a graph as follows. Let G be a graph with vertex set $V = \{v_1, v_2, \ldots, v_n\}$. Assign to each vertex v_i a list L_i containing all the vertices that are adjacent to v_i. The list L_i associated with v_i is referred to as the *adjacency list* of v_i. Then $L_i = [\,]$ if and only if v_i is an isolated vertex. We say that L_i is *the* adjacency list of v_i because any permutation of the elements of L_i results in a list that contains the same vertices adjacent to v_i. We are mainly concerned with the neighbors of v_i, but disregard the position where each neighbor is located in L_i. If each adjacency list L_i contains s_i elements where $0 \leq s_i \leq n$, we say that L_i has *length* s_i. The adjacency list representation of the graph G requires that we represent $\sum_i s_i = 2 \cdot |E(G)| \leq n^2$ elements in a computer's memory, since each edge appears twice in the adjacency list representation. An adjacency list is explicit about which vertices are adjacent to a vertex and implicit about which vertices are not adjacent to that same vertex. Without knowing the graph G, given the adjacency lists $L_1, L_2, \ldots, L_n$, we can reconstruct G. For example, Figure 2.1 shows a graph and its adjacency list representation.



$L_1 = [2, 8]$ $L_5 = [6, 8]$
$L_2 = [1, 6]$ $L_6 = [2, 5, 8]$
$L_3 = [4]$ $L_7 = [\,]$
$L_4 = [3]$ $L_8 = [1, 5, 6]$

Figure 2.1: A graph and its adjacency lists.

Example 2.1. *The Kneser graph with parameters (n, k), also known as the (n, k)-Kneser graph, is the graph whose vertices are all the k-subsets of $\{1, 2, \ldots, n\}$. Furthermore, two*

vertices are adjacent if their corresponding sets are disjoint. Draw the $(5,2)$*-Kneser graph and find its order and adjacency lists. In general, if* n *and* k *are positive, what is the order of the* (n,k)*-Kneser graph?*

Solution. The $(5,2)$-Kneser graph is the graph whose vertices are the 2-subsets

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

of $\{1,2,3,4,5\}$. That is, each vertex of the $(5,2)$-Kneser graph is a 2-combination of the set $\{1,2,3,4,5\}$ and therefore the graph itself has order $\binom{5}{2} = \frac{5\times 4}{2!} = 10$. The edges of this graph are

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

from which we obtain the following adjacency lists:

$$L_{\{1,2\}} = [\{3,4\},\ \{3,5\},\ \{4,5\}], \quad L_{\{1,3\}} = [\{2,4\},\ \{2,5\},\ \{4,5\}],$$
$$L_{\{1,4\}} = [\{2,3\},\ \{3,5\},\ \{2,5\}], \quad L_{\{1,5\}} = [\{2,4\},\ \{3,4\},\ \{2,3\}],$$
$$L_{\{2,3\}} = [\{1,5\},\ \{1,4\},\ \{4,5\}], \quad L_{\{2,4\}} = [\{1,3\},\ \{1,5\},\ \{3,5\}],$$
$$L_{\{2,5\}} = [\{1,3\},\ \{3,4\},\ \{1,4\}], \quad L_{\{3,4\}} = [\{1,2\},\ \{1,5\},\ \{2,5\}],$$
$$L_{\{3,5\}} = [\{2,4\},\ \{1,2\},\ \{1,4\}], \quad L_{\{4,5\}} = [\{1,3\},\ \{1,2\},\ \{2,3\}].$$

The $(5,2)$-Kneser graph itself is shown in Figure 2.2. Using Sage, we have

```
sage: K = graphs.KneserGraph(5, 2); K
Kneser graph with parameters 5,2: Graph on 10 vertices
sage: for v in K.vertices():
...       print(v, K.neighbors(v))
...
({4, 5}, [{1, 3}, {1, 2}, {2, 3}])
({1, 3}, [{2, 4}, {2, 5}, {4, 5}])
({2, 5}, [{1, 3}, {3, 4}, {1, 4}])
({2, 3}, [{1, 5}, {1, 4}, {4, 5}])
({3, 4}, [{1, 2}, {1, 5}, {2, 5}])
({3, 5}, [{2, 4}, {1, 2}, {1, 4}])
({1, 4}, [{2, 3}, {3, 5}, {2, 5}])
({1, 5}, [{2, 4}, {3, 4}, {2, 3}])
({1, 2}, [{3, 4}, {3, 5}, {4, 5}])
({2, 4}, [{1, 3}, {1, 5}, {3, 5}])
```

If n and k are positive integers, then the (n,k)-Kneser graph has

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!}$$

vertices. ■

We can categorize a graph $G = (V,E)$ as *dense* or *sparse* based upon its size. A dense graph has size $|E|$ that is close to $|V|^2$, i.e. $|E| = \Omega\big(|V|^2\big)$, in which case it is feasible to represent G as an adjacency matrix. The size of a sparse graph is much less than $|V|^2$, i.e. $|E| = \Omega\big(|V|\big)$, which renders the adjacency matrix representation as unsuitable. For a sparse graph, an adjacency list representation can require less storage space than an adjacency matrix representation of the same graph.

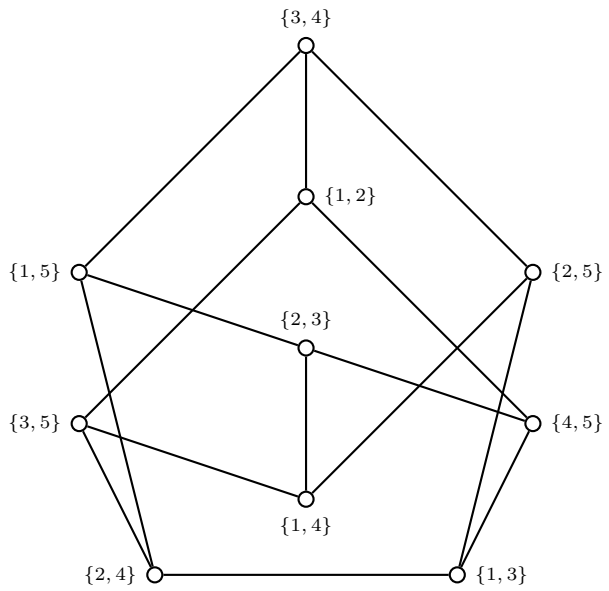


Figure 2.2: The $(5, 2)$-Kneser graph.

2.1.2 Edge lists

Lists can also be used to store the edges of a graph. To create an edge list L for a graph G, if uv is an edge of G then we let uv or the ordered pair (u, v) be an element of L. In general, let

$$v_0v_1,\ v_2v_3,\ \ldots,\ v_kv_{k+1}$$

be all the edges of G, where k is even. Then the edge list of G is given by

$$L = [v_0v_1,\ v_2v_3,\ \ldots,\ v_kv_{k+1}].$$

In some cases, it is desirable to have the edges of G be in contiguous list representation. If the edge list L of G is as given above, the contiguous edge list representation of the edges of G is

$$[v_0,\ v_1,\ v_2,\ v_3,\ \ldots,\ v_k,\ v_{k+1}].$$

That is, if $0 \leq i \leq k$ is even then v_iv_{i+1} is an edge of G.

2.1.3 The `graph6` format

The graph formats `graph6` and `sparse6` were developed by Brendan McKay [**?**] at The Australian National University as a compact way to represent graphs. These two formats use bit vectors and printable characters of the American Standard Code for Information Interchange (ASCII) encoding scheme. The 64 printable ASCII characters used in `graph6` and `sparse6` are those ASCII characters with decimal codes from 63 to 126, inclusive, as shown in Table 2.1. This section shall only cover the `graph6` format. For full specification on both of the `graph6` and `sparse6` formats, see McKay [**?**].

Bit vectors

Before discussing how `graph6` and `sparse6` represent graphs using printable ASCII characters, we first present encoding schemes used by these two formats. A *bit vector* is, as

binary	decimal	glyph	binary	decimal	glyph
0111111	63	?	1011111	95	_
1000000	64	@	1100000	96	`
1000001	65	A	1100001	97	a
1000010	66	B	1100010	98	b
1000011	67	C	1100011	99	c
1000100	68	D	1100100	100	d
1000101	69	E	1100101	101	e
1000110	70	F	1100110	102	f
1000111	71	G	1100111	103	g
1001000	72	H	1101000	104	h
1001001	73	I	1101001	105	i
1001010	74	J	1101010	106	j
1001011	75	K	1101011	107	k
1001100	76	L	1101100	108	l
1001101	77	M	1101101	109	m
1001110	78	N	1101110	110	n
1001111	79	O	1101111	111	o
1010000	80	P	1110000	112	p
1010001	81	Q	1110001	113	q
1010010	82	R	1110010	114	r
1010011	83	S	1110011	115	s
1010100	84	T	1110100	116	t
1010101	85	U	1110101	117	u
1010110	86	V	1110110	118	v
1010111	87	W	1110111	119	w
1011000	88	X	1111000	120	x
1011001	89	Y	1111001	121	y
1011010	90	Z	1111010	122	z
1011011	91	[	1111011	123	{
1011100	92	\	1111100	124	|
1011101	93	]	1111101	125	}
1011110	94	^	1111110	126	~

Table 2.1: ASCII printable characters used by `graph6` and `sparse6`.

its name suggests, a vector whose elements are 1's and 0's. It can be represented as a list of bits, e.g. `E` can be represented as the ASCII bit vector $[1, 0, 0, 0, 1, 0, 1]$. For brevity, we write a bit vector in a compact form such as `1000101`. The *length* of a bit vector is its number of bits. The *most significant bit* of a bit vector v is the bit position with the largest value among all the bit positions in v. Similarly, the *least significant bit* is the bit position in v having the least value among all the bit positions in v. The least significant bit of v is usually called the parity bit because when v is interpreted as an integer the parity bit determines whether the integer is even or odd. Reading `1000101` from left to right, the first bit `1` is the most significant bit, followed by the second bit `0` which is the second most significant bit, and so on all the way down to the seventh bit `1` which is the least significant bit.

The order in which we process the bits of a bit vector

$$v = b_{n-1}b_{n-2}\cdots b_0 \tag{2.1}$$

is referred to as *endianness*. Processing v in *big-endian* order means that we first process the most significant bit of v, followed by the second most significant bit, and so on all the way down to the least significant bit of v. Thus, in big-endian order we read the bits b_i of v from left to right in increasing order of powers of 2. Table 2.2 illustrates the big-endian interpretation of the ASCII binary representation of `E`. *Little-endian* order means that we first process the least significant bit, followed by the second least significant bit, and so on all the way up to the most significant bit. In little-endian order, the bits b_i are read from right to left in increasing order of powers of 2. Table 2.3 illustrates the little-endian interpretation of the ASCII binary representation of `E`. In his novel *Gulliver's Travels* first published in 1726, Jonathan Swift used the terms big- and little-endian in satirizing politicians who squabbled over whether to break an egg at the big end or the little end. Danny Cohen [**?**, **?**] first used the terms in 1980 as an April fool's joke in the context of computer architecture.

Suppose the bit vector (2.1) is read in big-endian order. To determine the integer representation of v, multiply each bit value by its corresponding position value, then add up all the results. Thus, if v is read in big-endian order, the integer representation of v is obtained by evaluating the polynomial

$$p(x) = \sum_{i=0}^{n-1} x^i b_i = x^{n-1}b_{n-1} + x^{n-2}b_{n-2} + \cdots + xb_1 + b_0. \tag{2.2}$$

at $x = 2$. See problem 2.2 for discussion of an efficient method to compute the integer representation of a bit vector.

position	0	1	2	3	4	5	6
bit value	1	0	0	0	1	0	1
position value	2^0	2^1	2^2	2^3	2^4	2^5	2^6

Table 2.2: Big-endian order of the ASCII binary code of `E`.

In `graph6` and `sparse6` formats, the length of a bit vector must be a multiple of 6. Suppose v is a bit vector of length k such that $6 \nmid k$. To transform v into a bit vector having length a multiple of 6, let $r = k \mod 6$ be the remainder upon dividing k by 6, and pad $6 - r$ zeros to the right of v.

position	0	1	2	3	4	5	6
bit value	1	0	0	0	1	0	1
position value	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Table 2.3: Little-endian order of the ASCII binary code of `E`.

Suppose $v = b_1 b_2 \cdots b_k$ is a bit vector of length k, where $6 \mid k$. We split v into $k/6$ bit vectors v_i, each of length 6. For $0 \leq i \leq k/6$, the i-th bit vector is given by

$$v_i = b_{6i-5} b_{6i-4} b_{6i-3} b_{6i-2} b_{6i-1} b_{6i}.$$

Consider each v_i as the big-endian binary representation of a positive integer. Use (2.2) to obtain the integer representation N_i of each v_i. Then add 63 to each N_i to obtain N_i' and store N_i' in one byte of memory. That is, each N_i' can be represented as a bit vector of length 8. Thus the required number of bytes to store v is $\lceil k/6 \rceil$. Let B_i be the byte representation of N_i' so that

$$R(v) = B_1 B_2 \cdots B_{\lceil k/6 \rceil} \tag{2.3}$$

denotes the representation of v as a sequence of $\lceil k/6 \rceil$ bytes.

We now discuss how to encode an integer n in the range $0 \leq n \leq 2^{36} - 1$ using (2.3) and denote such an encoding of n as $N(n)$. Let v be the big-endian binary representation of n. Then $N(n)$ is given by

$$N(n) = \begin{cases} n + 63, & \text{if } 0 \leq n \leq 62, \\ 126\, R(v), & \text{if } 63 \leq n \leq 258047, \\ 126\,126\, R(v), & \text{if } 258048 \leq n \leq 2^{36} - 1. \end{cases} \tag{2.4}$$

Note that $n + 63$ requires one byte of storage memory, while $126\, R(v)$ and $126\,126\, R(v)$ require 4 and 8 bytes, respectively.

The `graph6` format

The `graph6` format is used to represent simple, undirected graphs of order from 0 to $2^{36} - 1$, inclusive. Let G be a simple, undirected graph of order $0 \leq n \leq 2^{36} - 1$. If $n = 0$, then G is represented in `graph6` format as "`?`". Suppose $n > 0$. Let $M = [a_{ij}]$ be the adjacency matrix of G. Consider the upper triangle of M, excluding the main diagonal, and write that upper triangle as the bit vector

$$v = \underbrace{a_{0,1}}_{c_1} \underbrace{a_{0,2} a_{1,2}}_{c_2} \underbrace{a_{0,3} a_{1,3} a_{2,3}}_{c_3} \cdots \underbrace{a_{0,i} a_{1,i} \cdots a_{i-1,i}}_{c_i} \cdots \underbrace{a_{0,n} a_{1,n} \cdots a_{n-1,n}}_{c_n}$$

where c_i denotes the entries $a_{0,i} a_{1,i} \cdots a_{i-1,i}$ in column i of M. Then the `graph6` representation of G is $N(n)R(v)$, where $R(v)$ and $N(n)$ are as in (2.3) and (2.4), respectively. That is, $N(n)$ encodes the order of G and $R(v)$ encodes the edges of G.

2.2 Graph searching

> Errors, like straws, upon the surface flow;
> He who would search for pearls must dive below.
> — John Dryden, *All for Love*, 1678

This section discusses two fundamental algorithms for graph traversal: breadth-first search and depth-first search. The word "search" used in describing these two algorithms is rather misleading. It would be more accurate to describe them as algorithms for constructing trees using the adjacency information of a given graph. However, the names "breadth-first search" and "depth-first search" are entrenched in literature on graph theory and computer science. From hereon, we use these two names as given above, bearing in mind their intended purposes.

2.2.1 Breadth-first search

Breadth-first search (BFS) is a strategy for running through the vertices of a graph. It was presented by Moore [**?**] in 1959 within the context of traversing mazes. Lee [**?**] independently discovered the same algorithm in 1961 in his work on routing wires on circuit boards. In the physics literature, BFS is also known as a "burning algorithm" in view of the analogy of a fire burning and spreading through an area, a piece of paper, fabric, etc.

The basic BFS algorithm can be described as follows. Starting from a given vertex v of a graph G, we first explore the neighborhood of v by visiting all vertices that are adjacent to v. We then apply the same strategy to each of the neighbors of v. The strategy of exploring the neighborhood of a vertex is applied to all vertices of G. The result is a tree rooted at v and this tree is a subgraph of G. Algorithm 2.1 presents a general template for the BFS strategy. The tree resulting from the BFS algorithm is called a *breadth-first search tree*.

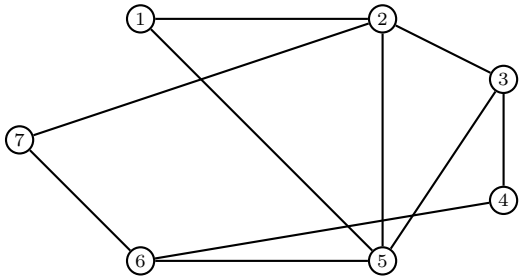
Algorithm 2.1: A general breadth-first search template.

Input: A directed or undirected graph $G = (V, E)$ of order $n > 0$. A vertex s from which to start the search. The vertices are numbered from 1 to $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.

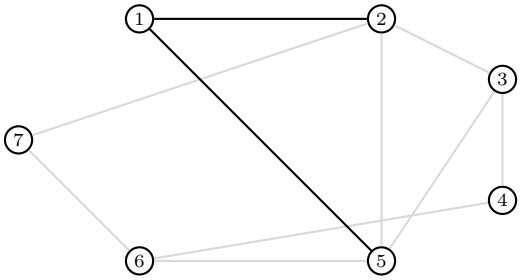
Output: A list D of distances of all vertices from s. A tree T rooted at s.

```
1  Q ← [s]                          /* queue of nodes to visit */
2  D ← [∞, ∞, ..., ∞]                        /* n copies of ∞ */
3  D[s] ← 0
4  T ← [ ]
5  while length(Q) > 0 do
6      v ← dequeue(Q)
7      for each w ∈ adj(v) do
8          if D[w] = ∞ then
9              D[w] ← D[v] + 1
10             enqueue(Q, w)
11             append(T, vw)
12 return (D, T)
```

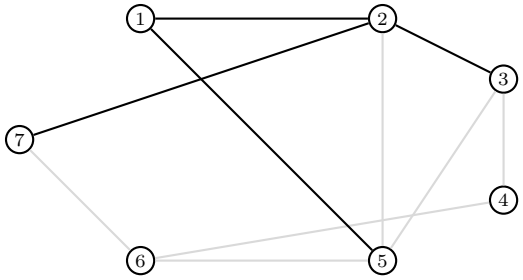
The breadth-first search algorithm makes use of a special type of list called a *queue*. This is analogous to a queue of people waiting in line to be served. A person may enter the queue by joining the rear of the queue. The person who is in the queue the longest amount of time is served first, followed by the person who has waited the second longest time, and so on. Formally, a queue Q is a list of elements. At any time, we only have access to the first element of Q, known as the *front* or *start* of the queue. We insert a new element into Q by appending the new element to the *rear* or *end* of the queue. The operation of removing the front of Q is referred to as *dequeue*, while the operation of appending to the rear of Q is called *enqueue*. That is, a queue implements a first-in first-out (FIFO) protocol for adding and removing elements. As with lists, the *length* of a queue is its total number of elements.



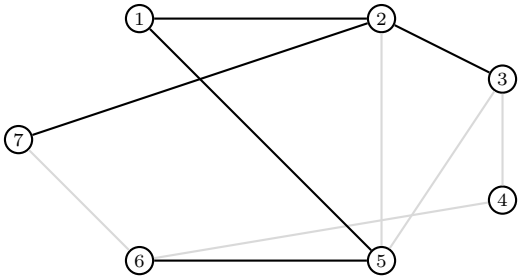
(a) Original undirected graph.



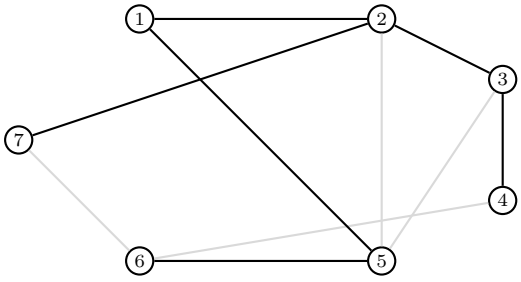
(b) First iteration of while loop.



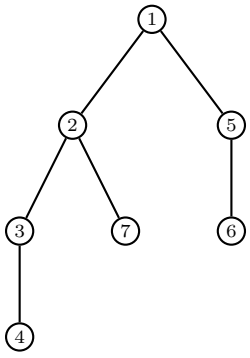
(c) Second iteration of while loop.



(d) Third iteration of while loop.



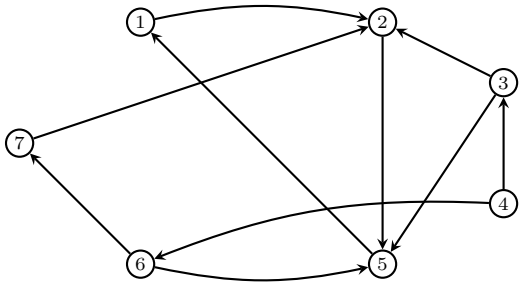
(e) Fourth iteration of while loop.



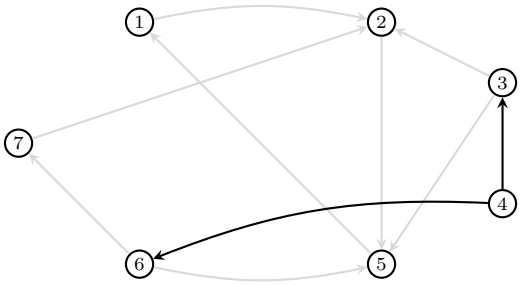
(f) Final BFS tree.

Figure 2.3: Breadth-first search tree for an undirected graph.

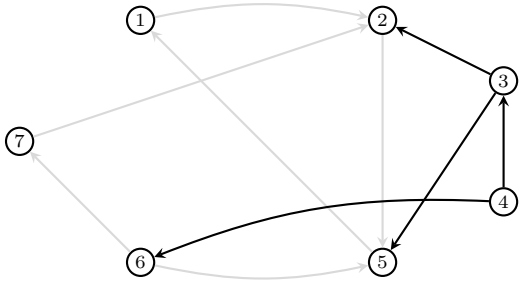
Note that the BFS Algorithm 2.1 works on both undirected and directed graphs. For an undirected graph, line 7 means that we explore all the neighbors of vertex v, i.e. the set $\text{adj}(v)$ of vertices adjacent to v. In the case of a digraph, we replace "$w \in \text{adj}(v)$" on line 7 with "$w \in \text{oadj}(v)$" because we only want to explore all vertices that are out-neighbors of v. The algorithm returns two lists D and T. The list T contains a subset



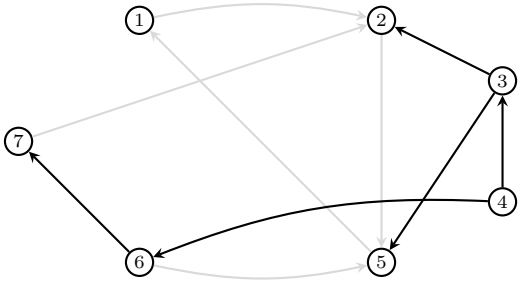
(a) Original digraph.



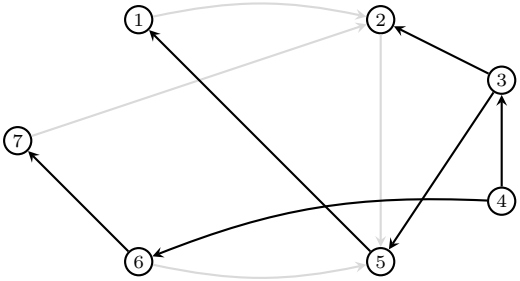
(b) First iteration of while loop.



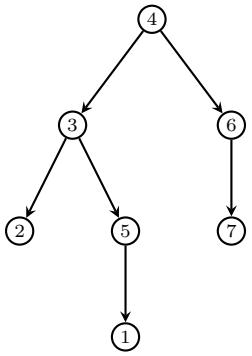
(c) Second iteration of while loop.



(d) Third iteration of while loop.



(e) Fourth iteration of while loop.



(f) Final BFS tree.

Figure 2.4: Breadth-first search tree for a digraph.

of edges in $E(G)$ that make up a tree rooted at the given start vertex s. As trees are connected graphs without cycles, we may take the vertices comprising the edges of T to be the vertex set of the tree. It is clear that T represents a tree by means of a list of edges, which allows us to identify the tree under consideration as the edge list T. The list D has the same number of elements as the order of $G = (V, E)$, i.e. $\text{length}(D) = |V|$. The i-th element $D[i]$ counts the number of edges in T between the vertices s and v_i. In other words, $D[i]$ is the length of the s-v_i path in T. It can be shown that $D[i] = \infty$ if and only if G is disconnected. After one application of Algorithm 2.1, it may happen that $D[i] = \infty$ for at least one vertex $v_i \in V$. To traverse those vertices that are unreachable from s, again we apply Algorithm 2.1 on G with starting vertex v_i. Repeat this algorithm as often as necessary until all vertices of G are visited. The result may be a tree that contains all the vertices of G or a collection of trees, each of which contains a subset of $V(G)$. Figures 2.3 and 2.4 present BFS trees resulting from applying Algorithm 2.1 on an undirected graph and a digraph, respectively.

Theorem 2.2. *The worst-case time complexity of Algorithm 2.1 is $O(|V| + |E|)$.*

Proof. Without loss of generality, we can assume that $G = (V, E)$ is connected. The initialization steps in lines 1 to 4 take $O(|V|)$ time. After initialization, all but one vertex are labelled ∞. Line 8 ensures that each vertex is enqueued at most once and hence dequeued at most once. Each of enqueuing and dequeuing takes constant time. The total time devoted to queue operations is $O(|V|)$. The adjacency list of a vertex is scanned after dequeuing that vertex, so each adjacency list is scanned at most once. Summing the lengths of the adjacency lists, we have $\Theta(|E|)$ and therefore we require $O(|E|)$ time to scan the adjacency lists. After the adjacency list of a vertex is scanned, at most k edges are added to the list T, where k is the length of the adjacency list under consideration. Like queue operations, appending to a list takes constant time, hence we require $O(|E|)$ time to build the list T. Therefore, BFS runs in $O(|V| + |E|)$ time. ■

Theorem 2.3. *For the list D resulting from Algorithm 2.1, let s be a starting vertex and let v be a vertex such that $D[v] \neq \infty$. Then $D[v]$ is the length of any shortest path from s to v.*

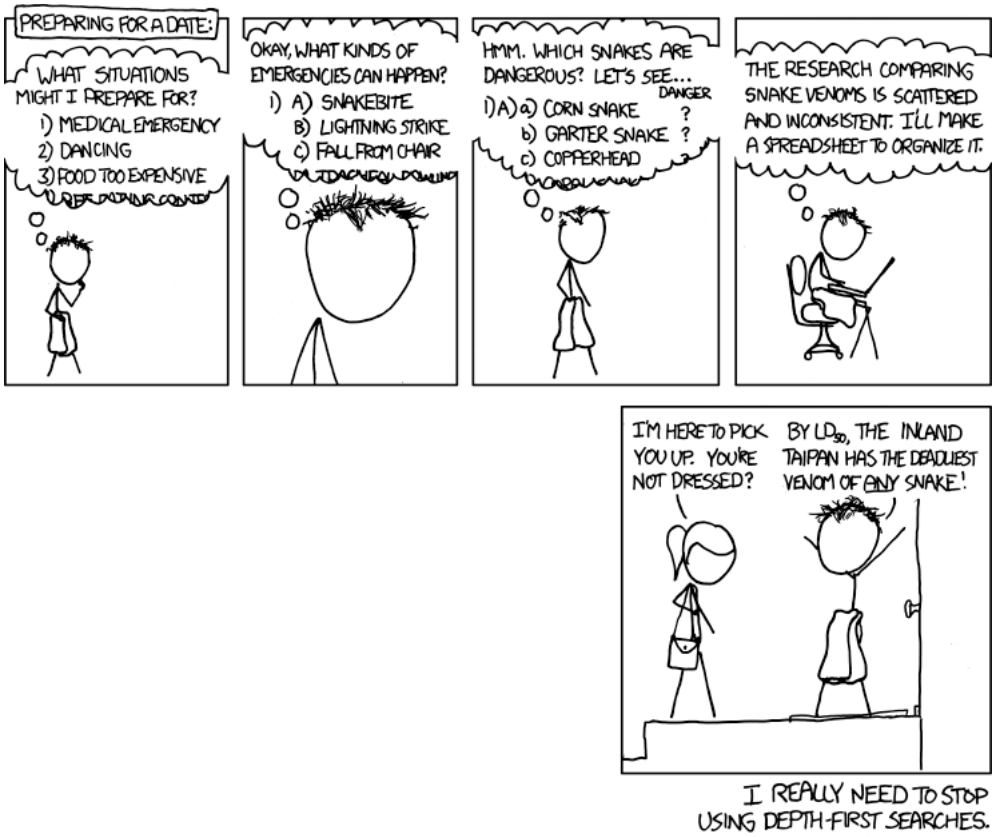
Proof. It is clear that $D[v] = \infty$ if and only if there are no paths from s to v. Let v be a vertex such that $D[v] \neq \infty$. As v can be reached from s by a path of length $D[v]$, the length $d(s, v)$ of any shortest s-v path satisfies $d(s, v) \leq D[v]$. Use induction on $d(s, v)$ to show that equality holds. For the base case $s = v$, we have $d(s, v) = D[v] = 0$ since the trivial path has length zero. Assume for induction that if $d(s, v) = k$, then $d(s, v) = D[v]$. Let $d(s, u) = k + 1$ with the corresponding shortest s-u path being $(s, v_1, v_2, \ldots, v_k, u)$. By our induction hypothesis, $(s, v_1, v_2, \ldots, v_k)$ is a shortest path from s to v_k of length $d(s, v_k) = D[v_k] = k$. In other words, $D[v_k] < D[u]$ and the while loop spanning lines 5 to 11 processes v_k before processing u. The graph under consideration has the edge $v_k u$. When examining the adjacency list of v_k, BFS reaches u (if u is not reached earlier) and so $D[u] \leq k + 1$. Hence, $D[u] = k + 1$ and therefore $d(s, u) = D[u] = k + 1$. ■

In the proof of Theorem 2.3, we used $d(u, v)$ to denote the length of the shortest path from u to v. This shortest path length is also known as the *distance* from u to v, and will be discussed in further details in section 2.3 and Chapter 5. The *diameter* $\text{diam}(G)$ of a graph $G = (V, E)$ is defined as

$$\text{diam}(G) = \max_{\substack{u,v \in V \\ u \neq v}} d(u, v). \tag{2.5}$$

Using the above definition, to find the diameter we first determine the distance between each pair of distinct vertices, then we compute the maximum of all such distances. Breadth-first search is a useful technique for finding the diameter: we simply run breadth-first search from each vertex. An interesting application of the diameter appears in the *small-world phenomenon* [**?**, **?**, **?**], which contends that a certain special class of sparse graphs have low diameter.

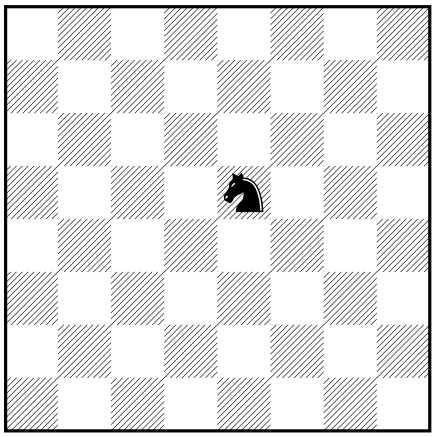
2.2.2 Depth-first search



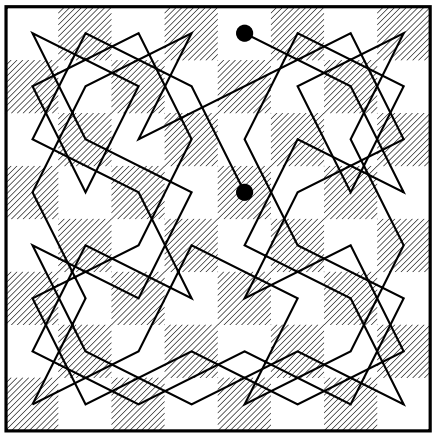
— Randall Munroe, xkcd, http://xkcd.com/761/

A depth-first search (DFS) is a graph traversal strategy similar to breadth-first search. Both BFS and DFS differ in how they explore each vertex. Whereas BFS explores the neighborhood of a vertex v before moving on to explore the neighborhoods of the neighbors, DFS explores as deep as possible a path starting at v. One can think of BFS as exploring the immediate surrounding, while DFS prefers to see what is on the other side of the hill. In the 19th century, Lucas [**?**] and Tarry [**?**] investigated DFS as a strategy for traversing mazes. Fundamental properties of DFS were discovered in the early 1970s by Hopcroft and Tarjan [**?**, **?**].

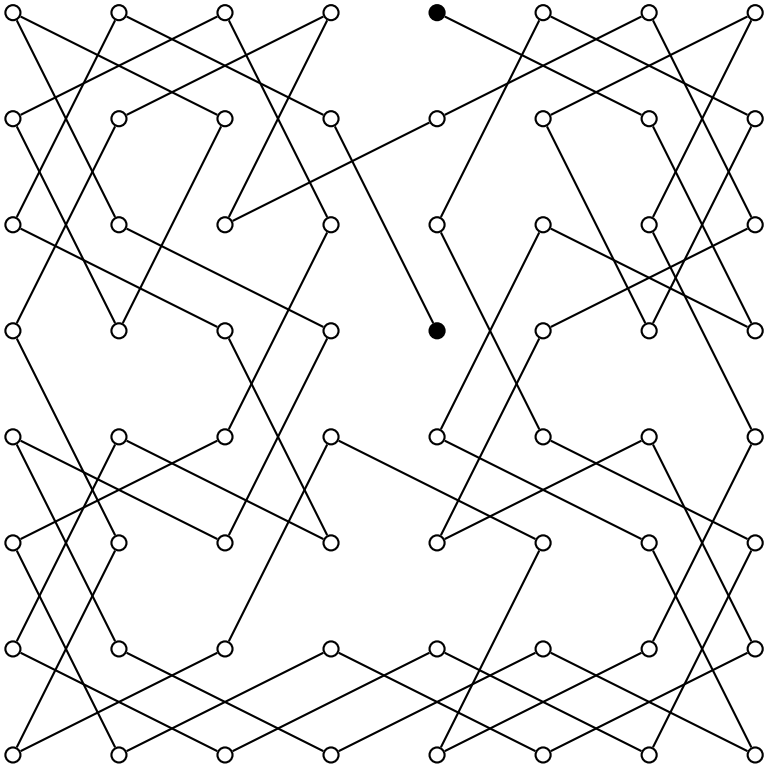
To get an intuitive appreciation for DFS, suppose we have an 8×8 chessboard in front of us. We place a single knight piece on a fixed square of the board, as shown in Figure 2.5(a). Our objective is to find a sequence of knight moves that visits each and every square exactly once, while obeying the rules of chess that govern the movement of the knight piece. Such a sequence of moves, if one exists, is called a *knight's tour*. How do we find such a tour? We could make one knight move after another, recording each move to ensure that we do not step on a square that is already visited, until we could not make any more moves. Acknowledging defeat when encountering a dead end, it might



(a) The knight's initial position.



(b) A knight's tour.



(c) Graph representation of the tour.

Figure 2.5: The knight's tour from a given starting position.

make sense to *backtrack* a few moves and try again, hoping we would not get stuck. If we fail again, we try backtracking a few more moves and traverse yet another path, hoping to make further progress. Repeat this strategy until a tour is found or until we have exhausted all possible moves. The above strategy for finding a knight's tour is an example of depth-first search, sometimes called *backtracking*. Figure 2.5(b) shows a knight's tour with the starting position as shown in Figure 2.5(a); and Figure 2.5(c) is a graph representation of this tour. The black-filled nodes indicate the endpoints of the tour. A more interesting question is: What is the number of knight's tours on an 8×8 chessboard? Loebbing and Wegener [**?**] announced in 1996 that this number is 33,439,123,484,294. The answer was later corrected by McKay [**?**] to be 13,267,364,410,532. See [**?**] for a discussion of the knight's tour and its relationship to mathematics.

Algorithm 2.2: A general depth-first search template.

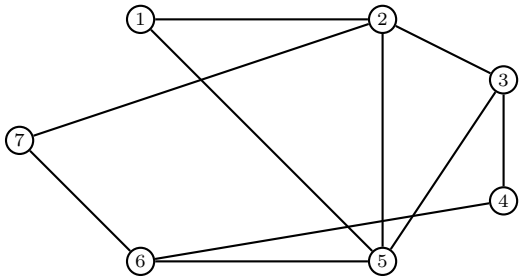
Input: A directed or undirected graph $G = (V, E)$ of order $n > 0$. A vertex s from which to start the search. The vertices are numbered from 1 to $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.

Output: A list D of distances of all vertices from s. A tree T rooted at s.

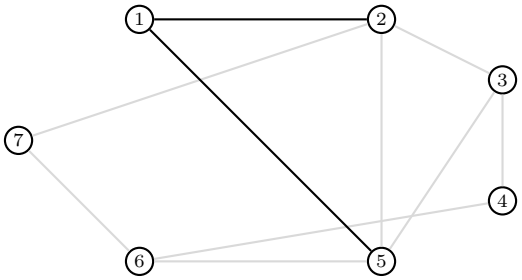
```
1  S ← [s]                          /* stack of nodes to visit */
2  D ← [∞, ∞, ..., ∞]                        /* n copies of ∞ */
3  D[s] ← 0
4  T ← [ ]
5  while length(S) > 0 do
6      v ← pop(S)
7      for each w ∈ adj(v) do
8          if D[w] = ∞ then
9              D[w] ← D[v] + 1
10             push(S, w)
11             append(T, vw)
12 return (D, T)
```

Algorithm 2.2 formalizes the above description of depth-first search. The tree resulting from applying DFS on a graph is called a *depth-first search tree*. The general structure of this algorithm bears close resemblance to Algorithm 2.1. A significant difference is that instead of using a queue to structure and organize vertices to be visited, DFS uses another special type of list called a *stack*. To understand how elements of a stack are organized, we use the analogy of a stack of cards. A new card is added to the stack by placing it on top of the stack. Any time we want to remove a card, we are only allowed to remove the top-most card that is on the top of the stack. A list $L = [a_1, a_2, \ldots, a_k]$ of k elements is a stack when we impose the same rules for element insertion and removal. The top and bottom of the stack are $L[k]$ and $L[1]$, respectively. The operation of removing the top element of the stack is referred to as *popping* the element off the stack. Inserting an element into the stack is called *pushing* the element onto the stack. In other words, a stack implements a last-in first-out (LIFO) protocol for element insertion and removal, in contrast to the FIFO policy of a queue. We also use the term *length* to refer to the number of elements in the stack.

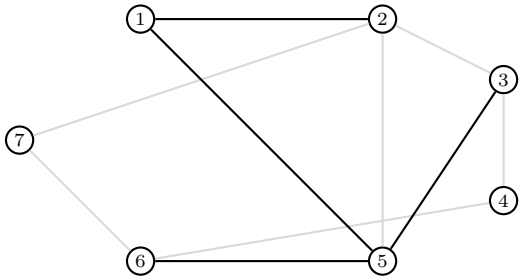
The depth-first search Algorithm 2.2 can be analyzed similar to how we analyzed Algorithm 2.3. Just as BFS is applicable to both directed and undirected graphs, we



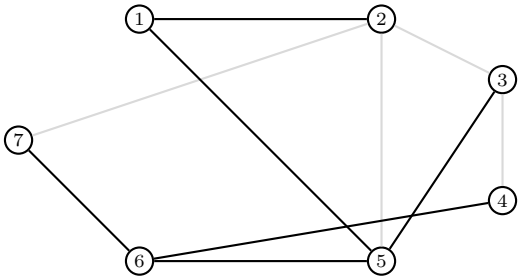
(a) Original undirected graph.



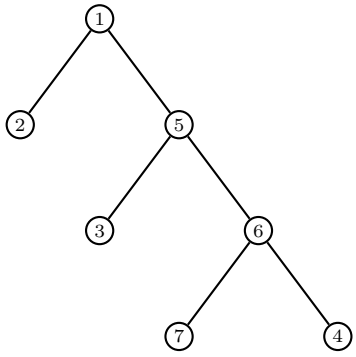
(b) First iteration of while loop.



(c) Second iteration of while loop.

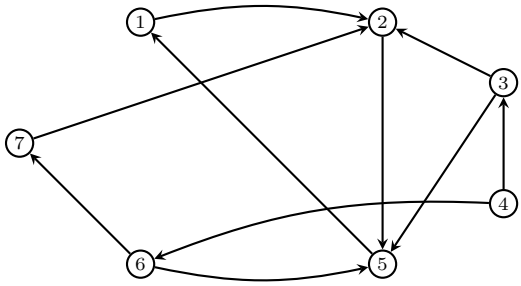


(d) Third iteration of while loop.

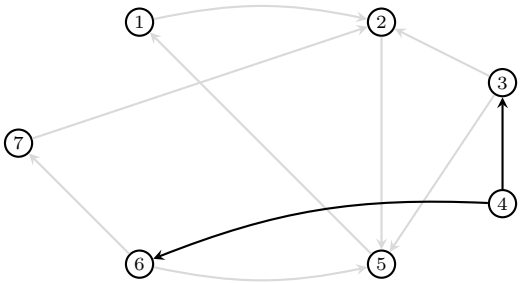


(e) Final DFS tree.

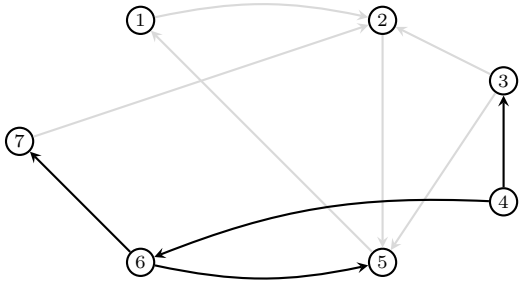
Figure 2.6: Depth-first search tree for an undirected graph.



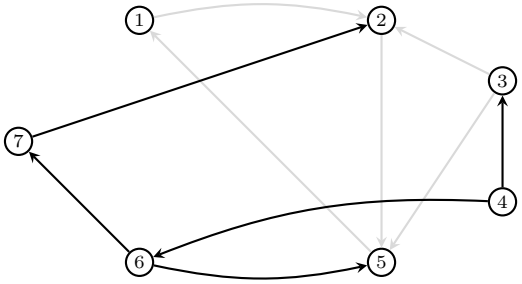
(a) Original digraph.



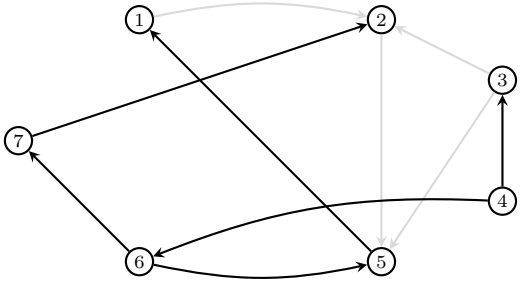
(b) First iteration of while loop.



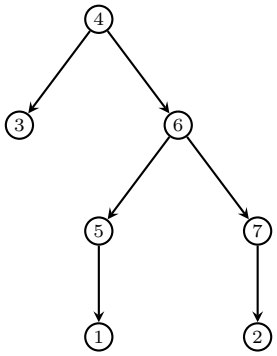
(c) Second iteration of while loop.



(d) Third iteration of while loop.



(e) Fourth iteration of while loop.



(f) Final DFS tree.

Figure 2.7: Depth-first search tree for a digraph.

can also have undirected graphs and digraphs as input to DFS. For the case of an undirected graph, line 7 of Algorithm 2.2 considers all vertices adjacent to the current vertex v. In case the input graph is directed, we replace "$w \in \text{adj}(v)$" on line 7 with "$w \in \text{oadj}(v)$" to signify that we only want to consider the out-neighbors of v. If any neighbors (respectively, out-neighbors) of v are labelled as ∞, we know that we have not explored any paths starting from any of those vertices. So we label each of those unexplored vertices with a positive integer and push them onto the stack S, where they will wait for later processing. We also record the paths leading from v to each of those unvisited neighbors, i.e. the edges vw for each vertex $w \in \text{adj}(v)$ (respectively, $w \in \text{oadj}(v)$) are appended to the list T. The test on line 8 ensures that we do not push onto S any vertices on the path that lead to v. When we resume another round of the while loop that starts on line 5, the previous vertex v have been popped off S and the neighbors (respectively, out-neighbors) of v have been pushed onto S. For example, in step 2 of Figure 2.6, vertex 5 is considered in DFS (in contrast to the vertex 2 in step 2 of the BFS in the graph in Figure 2.3) because DFS is organized by the LIFO protocol (in contrast to the FIFO protocol of BFS). To explore a path starting at v, we choose any unexplored neighbors of v by popping an element off S and repeat the for loop starting on line 7. Repeat the DFS algorithm as often as required in order to traverse all vertices of the input graph. The output of DFS consists of two lists D and T: T is a tree rooted at the starting vertex s; and each $D[i]$ counts the length of the s-v_i path in T. Figures 2.6 and 2.7 show the DFS trees resulting from running Algorithm 2.2 on an undirected graph and a digraph, respectively. The worst-case time complexity of DFS can be analyzed using an argument similar to that in Theorem 2.2. Arguing along the same lines as in the proof of Theorem 2.3, we can also show that the list D returned by DFS contains lengths of any shortest paths in the tree T from the starting vertex s to any other vertex in T (but not necessarily for shortest paths in the original graph G).

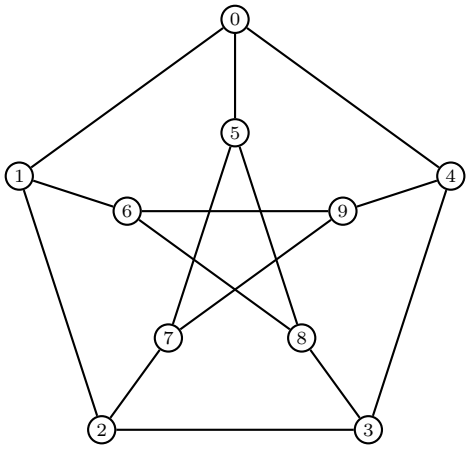


Figure 2.8: The Petersen graph.

Example 2.4. *In 1898, Julius Petersen published [?] a graph that now bears his name: the Petersen graph shown in Figure 2.8. Compare the search trees resulting from running breadth- and depth-first searches on the Petersen graph with starting vertex* 0.

Solution. The Petersen graph in Figure 2.8 can be constructed and searched as follows.

```
sage: g = graphs.PetersenGraph(); g
Petersen graph: Graph on 10 vertices
```

```
sage: list(g.breadth_first_search(0))
[0, 1, 4, 5, 2, 6, 3, 9, 7, 8]
sage: list(g.depth_first_search(0))
[0, 5, 8, 6, 9, 7, 2, 3, 4, 1]
```

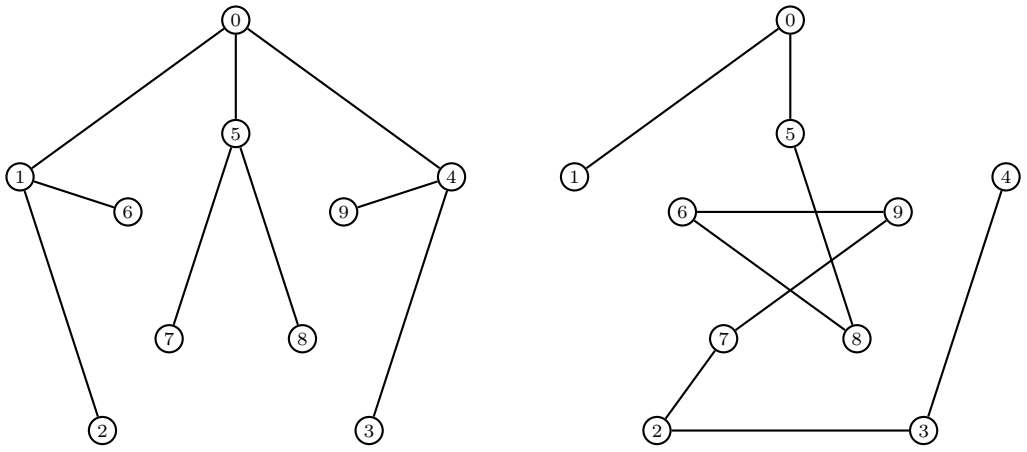
From the above Sage session, we see that starting from vertex 0 breadth-first search yields the edge list

$$[01, 04, 05, 12, 16, 43, 49, 57, 58]$$

and depth-first search produces the corresponding edge list

$$[05, 58, 86, 69, 97, 72, 23, 34, 01].$$

Our results are illustrated in Figure 2.9. ■



(a) Breadth-first search. (b) Depth-first search.

Figure 2.9: Traversing the Petersen graph starting from vertex 0.

2.2.3 Connectivity of a graph

Both BFS and DFS can be used to determine if an undirected graph is connected. Let $G = (V, E)$ be an undirected graph of order $n > 0$ and let s be an arbitrary vertex of G. We initialize a counter $c \leftarrow 1$ to mean that we are starting our exploration at s, hence we have already visited one vertex, i.e. s. We apply either BFS or DFS, treating G and s as input to any of these algorithms. Each time we visit a vertex that was previously unvisited, we increment the counter c. At the end of the algorithm, we compare c with n. If $c = n$, we know that we have visited all vertices of G and conclude that G is connected. Otherwise, we conclude that G is disconnected. This procedure is summarized in Algorithm 2.3.

Note that Algorithm 2.3 uses the BFS template of Algorithm 2.1, with some minor changes. Instead of initializing the list D with $n = |V|$ copies of ∞, we use n copies of 0. Each time we have visited a vertex w, we make the assignment $D[w] \leftarrow 1$, instead of incrementing the value $D[v]$ of w's parent vertex and assign that value to $D[w]$. At the end of the while loop, we have the equality $c = \sum_{d \in D} d$. The value of this sum could be used in the test starting from line 12. However, the value of the counter c is incremented immediately after we have visited an unvisited vertex. An advantage is that we do not need to perform a separate summation outside of the while loop. To use the DFS template for determining graph connectivity, we simply replace the queue implementation in Algorithm 2.3 with a stack implementation (see problem 2.20).

Algorithm 2.3: Determining whether an undirected graph is connected.

Input: An undirected graph $G = (V, E)$ of order $n > 0$. A vertex s from which to start the search. The vertices are numbered from 1 to $n = |V|$, i.e. $V = \{1, 2, \ldots, n\}$.

Output: True if G is connected; False otherwise.

```
1  Q ← [s]                          /* queue of nodes to visit */
2  D ← [0, 0, ..., 0]                       /* n copies of 0 */
3  D[s] ← 1
4  c ← 1
5  while length(Q) > 0 do
6      v ← dequeue(Q)
7      for each w ∈ adj(v) do
8          if D[w] = 0 then
9              D[w] ← 1
10             c ← c + 1
11             enqueue(Q, w)
12 if c = |V| then
13     return True
14 return False
```

2.3 Weights and distances

In Chapter 1, we briefly mentioned some applications of weighted graphs, but we did not define the concept of weighted graphs. A graph is said to be *weighted* when we assign a numeric label or weight to each of its edges. Depending on the application, we can let the vertices represent physical locations and interpret the weight of an edge as the distance separating two adjacent vertices. There might be a cost involved in traveling from a vertex to one of its neighbors, in which case the weight assigned to the corresponding edge can represent such a cost. The concept of *weighted digraphs* can be similarly defined. When no explicit weights are assigned to the edges of an undirected graph or digraph, it is usually convenient to consider each edge as having a weight of one or unit weight.

Based on the concept of weighted graphs, we now define what it means for a path to be a shortest path. Let $G = (V, E)$ be a (di)graph with nonnegative edge weights $w(e) \in \mathbf{R}$ for each edge $e \in E$. The *length* or *distance* $d(P)$ of a u-v path P from $u \in V$ to $v \in V$ is the sum of the edge weights for edges in P. Denote by $d(u, v)$ the smallest value of $d(P)$ for all paths P from u to v. When we regard edge weights as physical distances, a u-v path that realizes $d(u, v)$ is sometimes called a *shortest path* from u to v. The above definitions of distance and shortest path also apply to graphs with negative edge weights. Unless otherwise specified, where the weight of an edge is not explicitly given, we usually consider the edge to have unit weight.

The distance function d on a graph with nonnegative edge weights is known as a *metric function*. Intuitively, the distance between two physical locations is greater than zero. When these two locations coincide, i.e. they are one and the same location, the distance separating them is zero. Regardless of whether we are measuring the distance from location a to b or from b to a, we would obtain the same distance. Imagine now a third location c. The distance from a to b plus the distance from b to c is greater

than or equal to the distance from a to c. The latter principle is known as the *triangle inequality*. In summary, given three vertices u, v, w in a graph G, the distance function d on G satisfies the following property.

Lemma 2.5. ***Path distance as metric function.*** *Let $G = (V, E)$ be a graph with weight function $w : E \to \mathbf{R}$. Define a distance function $d : V \times V \to \mathbf{R}$ given by*

$$d(u, v) = \begin{cases} \infty, & \text{if there are no paths from } u \text{ to } v, \\ \min\{w(W) \mid W \text{ is a } u\text{-}v \text{ walk}\}, & \text{otherwise.} \end{cases}$$

Then d is a metric on V if it satisfies the following properties:

1. *Nonnegativity: $d(u, v) \geq 0$ with $d(u, v) = 0$ if and only if $u = v$.*
2. *Symmetry: $d(u, v) = d(v, u)$.*
3. *Triangle inequality: $d(u, v) + d(v, w) \geq d(u, w)$.*

The pair (V, d) is called a *metric space*, where the word "metric" refers to the distance function d. Any graphs we consider are assumed to have finite sets of vertices. For this reason, (V, d) is also known as a *finite metric space*. The distance matrix $D = [d(v_i, v_j)]$ of a connected graph is the distance matrix of its finite metric space. The topic of metric space is covered in further details in topology texts such as Runde [**?**] and Shirali and Vasudeva [**?**]. See Buckley and Harary [**?**] for an in-depth coverage of the distance concept in graph theory.

Many different algorithms exist for computing a shortest path in a weighted graph. Some only work if the graph has no negative weight cycles. Some assume that there is a single start or source vertex. Some compute the shortest paths from any vertex to any other and also detect if the graph has a negative weight cycle. No matter what algorithm is used for the special case of nonnegative weights, the length of the shortest path can neither equal nor exceed the order of the graph.

Lemma 2.6. *Fix a vertex v in a connected graph $G = (V, E)$ of order $n = |V|$. If there are no negative weight cycles in G, then there exists a shortest path from v to any other vertex $w \in V$ that uses at most $n - 1$ edges.*

Proof. Suppose that G contains no negative weight cycles. Observe that at most $n - 1$ edges are required to construct a path from v to any vertex w (Proposition 1.13). Let P denote such a path:

$$P : v_0 = v,\ v_1,\ v_2, \ldots, v_k = w.$$

Since G has no negative weight cycles, the weight of P is no less than the weight of P', where P' is the same as P except that all cycles have been removed. Thus, we can remove all cycles from P and obtain a v-w path P' of lower weight. Since the final path is acyclic, it must have no more than $n - 1$ edges. ■

Having defined weights and distances, we are now ready to discuss shortest path algorithms for weighted graphs. The breadth-first search Algorithm 2.1 can be applied where each edge has unit weight. Moving on to the general case of graphs with positive edge weights, algorithms for determining shortest paths in such graphs can be classified as *weight-setting* or *weight-correcting* [**?**]. A weight-setting method traverses a graph

Algorithm 2.4: A template for shortest path algorithms.

Input: A weighted graph or digraph $G = (V, E)$, where the vertices are numbered as $V = \{1, 2, \ldots, n\}$. A starting vertex s.

Output: A list D of distances from s to all other vertices. A list P of parent vertices such that $P[v]$ is the parent of v.

```
1  D ← [∞, ∞, ..., ∞]                    /* n copies of ∞ */
2  C ← list of candidate vertices to visit
3  while length(C) > 0 do
4      select v ∈ C
5      C ← remove(C, v)
6      for each u ∈ adj(v) do
7          if D[u] > D[v] + w(vu) then
8              D[u] ← D[v] + w(vu)
9              P[u] ← v
10             if u ∉ C then
11                 add u to C
12 return (D, P)
```

and assigns weights that, once assigned, remain unchanged for the duration of the algorithm. Weight-setting algorithms cannot deal with negative weights. On the other hand, a weight-correcting method is able to change the value of a weight many times while traversing a graph. In contrast to a weight-setting algorithm, a weight-correcting algorithm is able to deal with negative weights, provided that the weight sum of any cycle is nonnegative. The term *negative cycle* refers to the weight sum s of a cycle such that $s < 0$. Some algorithms halt upon detecting a negative cycle; examples of such algorithms include the Bellman-Ford and Johnson's algorithms.

Algorithm 2.4 is a general template for many shortest path algorithms. With a tweak here and there, one could modify it to suit the problem at hand. Note that $w(vu)$ is the weight of the edge vu. If the input graph is undirected, line 6 considers all the neighbors of v. For digraphs, we are interested in out-neighbors of v and accordingly we replace "$u \in \text{adj}(v)$" in line 6 with "$u \in \text{oadj}(v)$". The general flow of Algorithm 2.4 follows the same pattern as depth-first and breadth-first searches.

2.4 Dijkstra's algorithm

— Randall Munroe, xkcd, http://xkcd.com/342/

Dijkstra's algorithm [**?**], discovered by E. W. Dijkstra in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge weights. The algorithm is a generalization of breadth-first search. Imagine that the vertices of a weighted graph represent cities and edge weights represent distances between pairs of cities connected by a direct road. Dijkstra's algorithm can be used to find a shortest route from a fixed city to any other city.

Let $G = (V, E)$ be a (di)graph with nonnegative edge weights. Fix a start or source vertex $s \in V$. Dijkstra's Algorithm 2.5 performs a number of steps, basically one step for each vertex in V. First, we initialize a list D with n copies of ∞ and then assign 0 to $D[s]$. The purpose of the symbol ∞ is to denote the largest possible value. The list D is to store the distances of all shortest paths from s to any other vertices in G, where we take the distance of s to itself to be zero. The list P of parent vertices is initially empty and the queue Q is initialized to all vertices in G. We now consider each vertex in Q, removing any vertex after we have visited it. The while loop starting on line 5 runs until we have visited all vertices. Line 6 chooses which vertex to visit, preferring a vertex v whose distance value $D[v]$ from s is minimal. After we have determined such a vertex v, we remove it from the queue Q to signify that we have visited v. The for loop starting on line 8 adjusts the distance values of each neighbor u of v such that u is also in Q. If G is directed, we only consider out-neighbors of v that are also in Q. The conditional starting on line 9 is where the adjustment takes place. The expression $D[v] + w(vu)$ sums the distance from s to v and the distance from v to u. If this total sum is less than the distance $D[u]$ from s to u, we assign this lesser distance to $D[u]$ and let v be the parent vertex of u. In this way, we are choosing a neighbor vertex that results in minimal distance from s. Each pass through the while loop decreases the number of elements in Q by one without adding any elements to Q. Eventually, we would exit the while loop and the algorithm returns the lists D and P.

Example 2.7. *Apply Dijkstra's algorithm to the graph in Figure 2.10(a), with starting vertex v_1.*

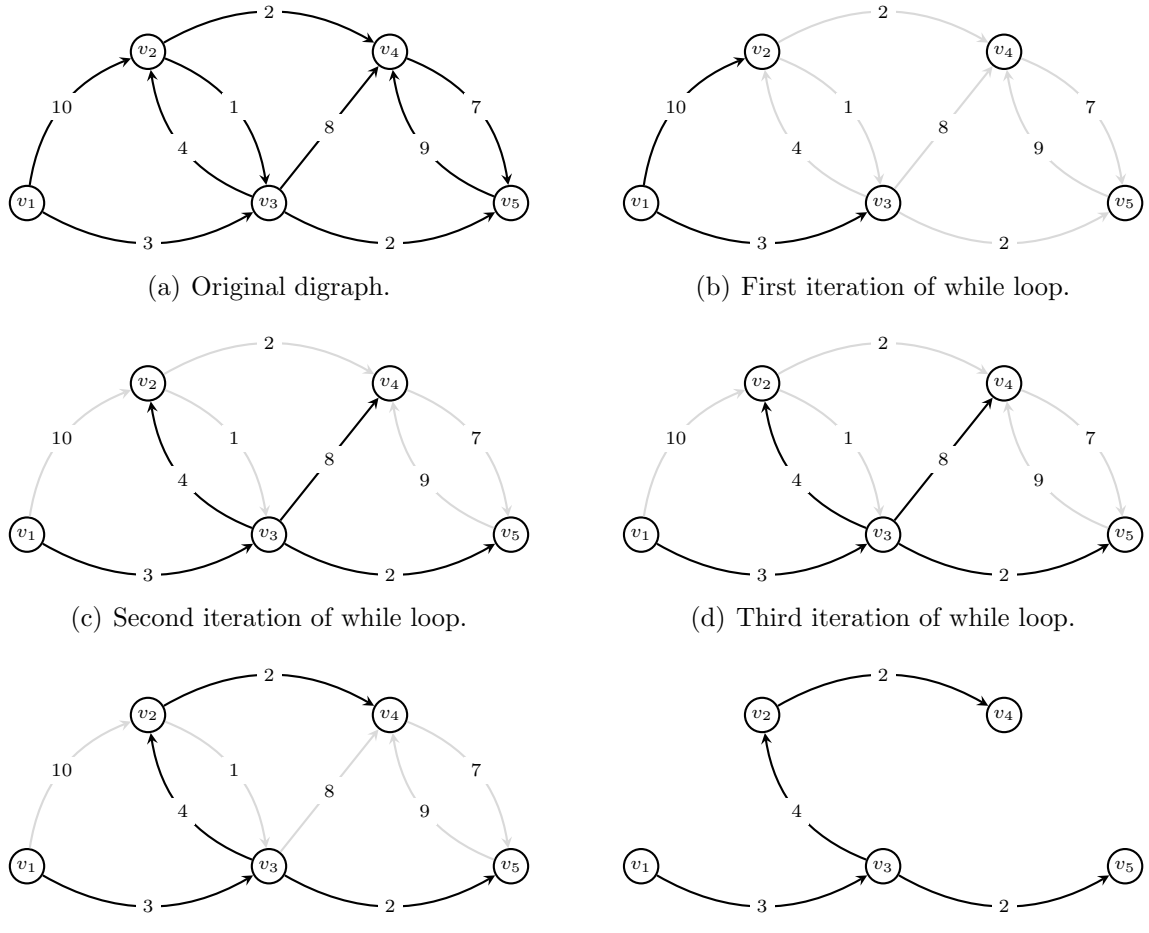
Solution. Dijkstra's Algorithm 2.5 applied to the graph in Figure 2.10(a) yields the sequence of intermediary graphs shown in Figure 2.10, culminating in the final shortest paths graph of Figure 2.10(f) and Table 2.4. For any column v_i in the table, each 2-tuple represents the distance and parent vertex of v_i. As we move along the graph, processing vertices according to Dijkstra's algorithm, the distance and parent vertex of a column

Algorithm 2.5: A general template for Dijkstra's algorithm.

Input: An undirected or directed graph $G = (V, E)$ that is weighted and has no self-loops. The order of G is $n > 0$. A vertex $s \in V$ from which to start the search. Vertices are numbered from 1 to n, i.e. $V = \{1, 2, \ldots, n\}$.

Output: A list D of distances such that $D[v]$ is the distance of a shortest path from s to v. A list P of vertex parents such that $P[v]$ is the parent of v, i.e. v is adjacent from $P[v]$.

```
1  D ← [∞, ∞, ..., ∞]                    /* n copies of ∞ */
2  D[s] ← 0
3  P ← [ ]
4  Q ← V                                  /* list of nodes to visit */
5  while length(Q) > 0 do
6      find v ∈ Q such that D[v] is minimal
7      Q ← remove(Q, v)
8      for each u ∈ adj(v) ∩ Q do
9          if D[u] > D[v] + w(vu) then
10             D[u] ← D[v] + w(vu)
11             P[u] ← v
12 return (D, P)
```



(a) Original digraph.

(b) First iteration of while loop.

(c) Second iteration of while loop.

(d) Third iteration of while loop.

(e) Fourth iteration of while loop.

(f) Final shortest paths graph.

Figure 2.10: Searching a weighted digraph using Dijkstra's algorithm.

v_1	v_2	v_3	v_4	v_5
$\underline{(0,-)}$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$
	$(10,v_1)$	$\underline{(3,v_1)}$	$(11,v_3)$	$\underline{(5,v_3)}$
	$\underline{(7,v_3)}$		$\underline{(9,v_2)}$	

Table 2.4: Stepping through Dijkstra's algorithm.

are updated. The underlined 2-tuple represents the final distance and parent vertex produced by Dijkstra's algorithm. From Table 2.4, we have the following shortest paths and distances:

$$\begin{aligned} &v_1\text{-}v_2 : v_1, v_3, v_2 && d(v_1, v_2) = 7 \\ &v_1\text{-}v_3 : v_1, v_3 && d(v_1, v_3) = 3 \\ &v_1\text{-}v_4 : v_1, v_3, v_2, v_4 && d(v_1, v_4) = 9 \\ &v_1\text{-}v_5 : v_1, v_3, v_5 && d(v_1, v_5) = 5 \end{aligned}$$

Intermediary vertices for a u-v path are obtained by starting from v and work backward using the parent of v, then the parent of the parent, and so on. ■

Dijkstra's algorithm is an example of a *greedy algorithm*. Whenever it tries to find the next vertex, it chooses only that vertex that minimizes the total weight so far. Greedy algorithms may not produce the best possible result. However, as the following theorem shows, Dijkstra's algorithm does indeed produce shortest paths.

Theorem 2.8. ***Correctness of Algorithm 2.5.*** *Let $G = (V, E)$ be a weighted (di)graph with a nonnegative weight function w. When Dijkstra's algorithm is applied to G with source vertex $s \in V$, the algorithm terminates with $D[u] = d(s, u)$ for all $u \in V$. Furthermore, if $D[v] \neq \infty$ and $v \neq s$, then $s = u_1, u_2, \ldots, u_k = v$ is a shortest s-v path such that $u_{i-1} = P[u_i]$ for $i = 2, 3, \ldots, k$.*

Proof. If G is disconnected, then any $v \in V$ that cannot be reached from s has distance $D[v] = \infty$ upon algorithm termination. Hence, it suffices to consider the case where G is connected. Let $V = \{s = v_1, v_2, \ldots, v_n\}$ and use induction on i to show that after visiting v_i we have

$$D[v] = d(s, v) \qquad \text{for all } v \in V_i = \{v_1, v_2, \ldots, v_i\}. \tag{2.6}$$

For $i = 1$, equality holds. Assume for induction that (2.6) holds for some $1 \leq i \leq n-1$, so that now our task is to show that (2.6) holds for $i+1$. To verify $D[v_{i+1}] = d(s, v_{i+1})$, note that by our inductive hypothesis,

$$D[v_{i+1}] = \min\{\, d(s, v) + w(vu) \mid v \in V_i \text{ and } u \in \text{adj}(v) \cap (Q \backslash V_i)\}$$

and respectively

$$D[v_{i+1}] = \min\{\, d(s, v) + w(vu) \mid v \in V_i \text{ and } u \in \text{oadj}(v) \cap (Q \backslash V_i)\}$$

if G is directed. Therefore, $D[v_{i+1}] = d(s, v_{i+1})$.

Let $v \in V$ such that $D[v] \neq \infty$ and $v \neq s$. We now construct an s-v path. When Algorithm 2.5 terminates, we have $D[v] = D[v_1] + w(v_1 v)$, where $P[v] = v_1$ and $d(s, v) = d(s, v_1) + w(v_1 v)$. This means that v_1 is the second-to-last vertex in a shortest s-v path. Repeated application of this process using the parent list P, we eventually produce a shortest s-v path $s = v_m, v_{m-1}, \ldots, v_1, v$, where $P[v_i] = v_{i+1}$ for $i = 1, 2, \ldots, m-1$. ■

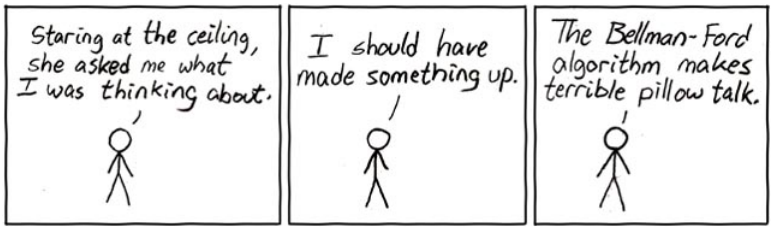
To analyze the worst case time complexity of Algorithm 2.5, note that initializing D takes $O(n+1)$ and initializing Q takes $O(n)$, for a total of $O(n)$ devoted to initialization. Each extraction of a vertex v with minimal $D[v]$ requires $O(n)$ since we search through the entire list Q to determine the minimum value, for a total of $O(n^2)$. Each insertion into D requires constant time and the same holds for insertion into P. Thus, insertion into D and P takes $O(|E| + |E|) = O(|E|)$, which require at most $O(n)$ time. In the worst case, Dijkstra's Algorithm 2.5 has running time $O(n^2 + n) = O(n^2)$.

Can we improve the run time of Dijkstra's algorithm? The time complexity of Dijkstra's algorithm depends on its implementation. With a simple list implementation as presented in Algorithm 2.5, we have a worst case time complexity of $O(n^2)$, where n is the order of the graph under consideration. Let m be the size of the graph. Table 2.5 presents time complexities of Dijkstra's algorithm for various implementations. Out of all the four implementations in this table, the heap implementations are much more efficient than the list implementation presented in Algorithm 2.5. A heap is a type of tree, a topic which will be covered in Chapter 3. Of all the heap implementations in Table 2.5, the Fibonacci heap implementation [**?**] yields the best runtime. Chapter 4 discusses how to use trees for efficient implementations of priority queues via heaps.

Implementation	Time complexity
list	$O(n^2)$
binary heap	$O\big((n+m)\ln n\big)$
k-ary heap	$O\big((kn+m)\frac{\ln n}{\ln k}\big)$
Fibonacci heap	$O(n\ln n + m)$

Table 2.5: Implementation specific worst-case time complexity of Dijkstra's algorithm.

2.5 Bellman-Ford algorithm



— Randall Munroe, xkcd, http://xkcd.com/69/

A disadvantage of Dijkstra's Algorithm 2.5 is that it cannot handle graphs with negative edge weights. The Bellman-Ford algorithm computes single-source shortest paths in a weighted graph or digraph, where some of the edge weights may be negative. This algorithm is a modification of the one published in 1957 by Richard E. Bellman [**?**] and that by Lester Randolph Ford, Jr. [**?**] in 1956. Shimbel [**?**] independently discovered the same method in 1955, and Moore [**?**] in 1959. In contrast to the "greedy" approach that Dijkstra's algorithm takes, i.e. searching for the "cheapest" path, the Bellman-Ford algorithm searches over all edges and keeps track of the shortest one found as it searches.

The Bellman-Ford Algorithm 2.6 runs in time $O(mn)$, where m and n are the size and order of an input graph, respectively. To see this, note that the initialization on

Algorithm 2.6: The Bellman-Ford algorithm.

Input: An undirected or directed graph $G = (V, E)$ that is weighted and has no self-loops. Negative edge weights are allowed. The order of G is $n > 0$. A vertex $s \in V$ from which to start the search. Vertices are numbered from 1 to n, i.e. $V = \{1, 2, \ldots, n\}$.

Output: A list D of distances such that $D[v]$ is the distance of a shortest path from s to v. A list P of vertex parents such that $P[v]$ is the parent of v, i.e. v is adjacent from $P[v]$. If G has negative-weight cycles, then return False. Otherwise, return D and P.

```
1  D ← [∞, ∞, ..., ∞]                    /* n copies of ∞ */
2  D[s] ← 0
3  P ← [ ]
4  for i ← 1, 2, ..., n − 1 do
5      for each edge uv ∈ E do
6          if D[v] > D[u] + w(uv) then
7              D[v] ← D[u] + w(uv)
8              P[v] ← u
9  for each edge uv ∈ E do
10     if D[v] > D[u] + w(uv) then
11         return False
12 return (D, P)
```

lines 1 to 3 takes $O(n)$. Each of the $n - 1$ rounds of the for loop starting on line 4 takes $O(m)$, for a total of $O(mn)$ time. Finally, the for loop starting on line 9 takes $O(m)$.

The loop starting on line 4 performs at most $n - 1$ updates of the distance $D[v]$ of each head of an edge. Many graphs have sizes that are less then $n - 1$, resulting in a number of redundant rounds of updates. To avoid such redundancy, we could add an extra check in the outer loop spanning lines 4 to 8 to immediately terminate that outer loop after any round that did not result in an update of any $D[v]$. Algorithm 2.7 presents a modification of the Bellman-Ford Algorithm 2.6 that avoids redundant rounds of updates.

2.6 Floyd-Roy-Warshall algorithm

> The shortest distance between two points is not a very interesting journey.
> — R. Goldberg

Let D be a weighted digraph of order n and size m. Dijkstra's Algorithm 2.5 and the Bellman-Ford Algorithm 2.6 can be used to determine shortest paths from a single source vertex to all other vertices of D. To determine a shortest path between each pair of distinct vertices in D, we repeatedly apply either of these algorithms to each vertex of D. Such repeated application of Dijkstra's and the Bellman-Ford algorithms results in algorithms that run in time $O(n^3)$ and $O(n^2m)$, respectively.

The *Floyd-Roy-Warshall algorithm* (FRW), or the Floyd-Warshall algorithm, is an algorithm for finding shortest paths in a weighted, directed graph. Like the Bellman-Ford algorithm, it allows for negative edge weights and detects a negative weight cycle if one exists. Assuming that there are no negative weight cycles, a single execution of

Algorithm 2.7: The Bellman-Ford algorithm with checks for redundant updates.

Input: An undirected or directed graph $G = (V, E)$ that is weighted and has no self-loops. Negative edge weights are allowed. The order of G is $n > 0$. A vertex $s \in V$ from which to start the search. Vertices are numbered from 1 to n, i.e. $V = \{1, 2, \ldots, n\}$.

Output: A list D of distances such that $D[v]$ is the distance of a shortest path from s to v. A list P of vertex parents such that $P[v]$ is the parent of v, i.e. v is adjacent from $P[v]$. If G has negative-weight cycles, then return False. Otherwise, return D and P.

```
1  D ← [∞, ∞, ..., ∞]                    /* n copies of ∞ */
2  D[s] ← 0
3  P ← [ ]
4  for i ← 1, 2, ..., n − 1 do
5      updated ← False
6      for each edge uv ∈ E do
7          if D[v] > D[u] + w(uv) then
8              D[v] ← D[u] + w(uv)
9              P[v] ← u
10             updated ← True
11     if updated = False then
12         exit the loop
13 for each edge uv ∈ E do
14     if D[v] > D[u] + w(uv) then
15         return False
16 return (D, P)
```

the FRW algorithm will find the shortest paths between all pairs of vertices. It was discovered independently by Bernard Roy [**?**] in 1959, Robert Floyd [**?**] in 1962, and by Stephen Warshall [**?**] in 1962.

In some sense, the FRW algorithm is an example of *dynamic programming*, which allows one to break the computation into simpler steps using some sort of recursive procedure. The rough idea is as follows. Temporarily label the vertices of a weighted digraph G as $V = \{1, 2, \ldots, n\}$ with $n = |V(G)|$. Let $W = [w(i, j)]$ be the weight matrix of G where

$$w(i,j) = \begin{cases} w(ij), & \text{if } ij \in E(G), \\ 0, & \text{if } i = j, \\ \infty, & \text{otherwise.} \end{cases} \tag{2.7}$$

Let $P_k(i, j)$ be a shortest path from i to j such that its intermediate vertices are in $\{1, 2, \ldots, k\}$. Let $D_k(i, j)$ be the weight (or distance) of $P_k(i, j)$. If no shortest i-j paths exist, define $P_k(i, j) = \infty$ and $D_k(i, j) = \infty$ for all $k \in \{1, 2, \ldots, n\}$. If $k = 0$, then $P_0(i, j) : i, j$ since no intermediate vertices are allowed in the path and hence $D_0(i, j) = w(i, j)$. In other words, if i and j are adjacent, a shortest i-j path is the edge ij itself and the weight of this path is simply the weight of ij. Now consider $P_k(i, j)$ for $k > 0$. Either $P_k(i, j)$ passes through k or it does not. If k is not on the path $P_k(i, j)$, then the intermediate vertices of $P_k(i, j)$ are in $\{1, 2, \ldots, k - 1\}$, as are the vertices of $P_{k-1}(i, j)$. In case $P_k(i, j)$ contains the vertex k, then $P_k(i, j)$ traverses k exactly once by the definition of path. The i-k subpath in $P_k(i, j)$ is a shortest i-k path whose intermediate vertices are drawn from $\{1, 2, \ldots, k - 1\}$, which is also the set of intermediate vertices for the k-j subpath in $P_k(i, j)$. That is, to obtain $P_k(i, j)$, we take the union of the paths $P_{k-1}(i, k)$ and $P_{k-1}(k, j)$. We compute the weight $D_k(i, j)$ of $P_k(i, j)$ using the expression

$$D_k(i,j) = \begin{cases} w(i,j), & \text{if } k = 0, \\ \min\{D_{k-1}(i,j),\, D_{k-1}(i,k) + D_{k-1}(k,j)\}, & \text{if } k > 0. \end{cases} \tag{2.8}$$

The key to the Floyd-Roy-Warshall algorithm lies in exploiting expression (2.8). If $n = |V|$, then this is a $O(n^3)$ time algorithm. For comparison, the Bellman-Ford algorithm has complexity $O(|V| \cdot |E|)$, which is $O(n^3)$ time for dense graphs. However, Bellman-Ford only yields the shortest paths emanating from a *single* vertex. To achieve comparable output, we would need to iterate Bellman-Ford over *all* vertices, which would be an $O(n^4)$ time algorithm for dense graphs. Except possibly for sparse graphs, Floyd-Roy-Warshall is better than an iterated implementation of Bellman-Ford. Note that $P_k(i, k) = P_{k-1}(i, k)$ and $P_k(k, i) = P_{k-1}(k, i)$, consequently $D_k(i, k) = D_{k-1}(i, k)$ and $D_k(k, i) = D_{k-1}(k, i)$. This observation allows us to replace $P_k(i, j)$ with $P(i, j)$ for $k = 1, 2, \ldots, n$. The final results of $P(i, j)$ and $D(i, k)$ are the same as $P_n(i, j)$ and $D_n(i, j)$, respectively. Algorithm 2.8 summarizes the above discussion into an algorithmic presentation.

Like the Bellman-Ford algorithm, the Floyd-Roy-Warshall algorithm can also detect the presence of negative weight cycles. If G is a weighted digraph without self-loops, by (2.7) we have $D(i, i) = 0$ for $i = 1, 2, \ldots, n$. Any path p starting and ending at i could only improve upon the initial weight of 0 if the weight sum of p is less than zero, i.e. a negative weight cycle. Upon termination of Algorithm 2.8, if $D(i, i) < 0$, we conclude that there is a path starting and ending at i whose weight sum is negative.

Algorithm 2.8: The Floyd-Roy-Warshall algorithm for all-pairs shortest paths.

Input: A weighted digraph $G = (V, E)$ that has no self-loops. Negative edge weights are allowed. The order of G is $n > 0$. Vertices are numbered from 1 to n, i.e. $V = \{1, 2, \ldots, n\}$. The weight matrix $W = [w(i, j)]$ of G as defined in (2.7).

Output: A matrix $P = [a_{ij}]$ of shortest paths in G. A matrix $D = [a_{ij}]$ of distances where $D[i, j]$ is the weight (or distance) of a shortest i-j path in G.

$n \leftarrow |V|$
$P[a_{ij}] \leftarrow$ an $n \times n$ zero matrix
$D[a_{ij}] \leftarrow W[w(i, j)]$
for $k \leftarrow 1, 2, \ldots, n$ **do**
  for $i \leftarrow 1, 2, \ldots, n$ **do**
    for $j \leftarrow 1, 2, \ldots, n$ **do**
      if $D[i, j] > D[i, k] + D[k, j]$ **then**
        $P[i, j] \leftarrow k$
        $D[i, j] \leftarrow D[i, k] + D[k, j]$
return (P, D)

Here is an implementation in Sage.

```
def floyd_roy_warshall(A):
    """
    Shortest paths

    INPUT:

    - A -- weighted adjacency matrix

    OUTPUT:

    - dist -- a matrix of distances of shortest paths.
    - paths -- a matrix of shortest paths.
    """
    G = Graph(A, format="weighted_adjacency_matrix")
    V = G.vertices()
    E = [(e[0],e[1]) for e in G.edges()]
    n = len(V)
    dist = [[0]*n for i in range(n)]
    paths = [[-1]*n for i in range(n)]
    # initialization step
    for i in range(n):
        for j in range(n):
            if (i,j) in E:
                paths[i][j] = j
            if i == j:
                dist[i][j] = 0
            elif A[i][j]<>0:
                dist[i][j] = A[i][j]
            else:
                dist[i][j] = infinity
    # iteratively finding the shortest path
    for j in range(n):
        for i in range(n):
            if i <> j:
                for k in range(n):
                    if k <> j:
                        if dist[i][k]>dist[i][j]+dist[j][k]:
                            paths[i][k] = V[j]
                        dist[i][k] = min(dist[i][k], dist[i][j] +dist[j][k])
    for i in range(n):
        if dist[i][i] < 0:
            raise ValueError, "A negative edge weight cycle exists."
```

```
    return dist, matrix(paths)
```

Here are some examples.

```
sage: A = matrix([[0,1,2,3], [0,0,2,1], [-5,0,0,3], [1,0,1,0]]); A
sage: floyd_roy_warshall(A)
Traceback (click to the left of this block for traceback)
...
ValueError: A negative edge weight cycle exists.
```

The plot of this weighted digraph with four vertices appears in Figure 2.11.

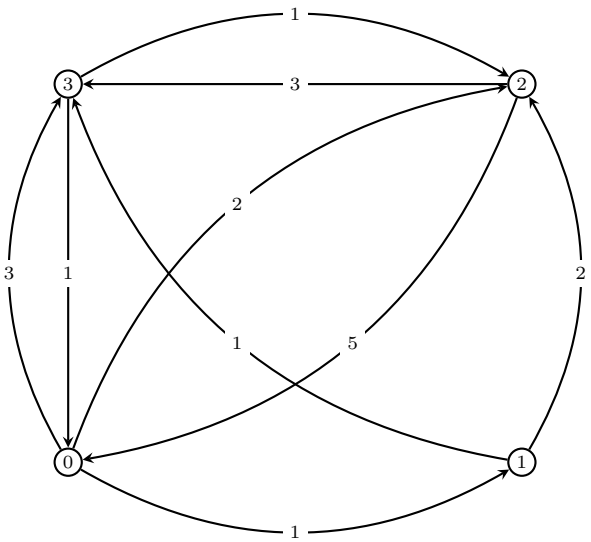


Figure 2.11: Demonstrating the Floyd-Roy-Warshall algorithm.

```
sage: A = matrix([[0,1,2,3], [0,0,2,1], [-1/2,0,0,3], [1,0,1,0]]); A
sage: floyd_roy_warshall(A)
([[0, 1, 2, 2], [3/2, 0, 2, 1], [-1/2, 1/2, 0, 3/2], [1/2, 3/2, 1, 0]],
  [-1  1  2  1]
  [ 2 -1  2  3]
  [-1  0 -1  1]
  [ 2  2 -1 -1])
```

The plot of this weighted digraph with four vertices appears in Figure 2.12.

Example 2.9. *Section 1.6 briefly presented the concept of molecular graphs in chemistry. The Wiener number of a molecular graph was first published in 1947 by Harold Wiener [?] who used it in chemistry to study properties of alkanes. Other applications [?] of the Wiener number to chemistry are now known. If $G = (V, E)$ is a connected graph with vertex set $V = \{v_1, v_2, \ldots, v_n\}$, then the Wiener number W of G is defined by*

$$W(G) = \sum_{i<j} d(v_i, v_j) \tag{2.9}$$

where $d(v_i, v_j)$ is the distance from v_i to v_j. What is the Wiener number of the molecular graph in Figure 2.13?

Solution. Consider the molecular graph in Figure 2.13 as directed with unit weight. To compute the Wiener number of this graph, use the Floyd-Roy-Warshall algorithm to obtain a distance matrix $D = [d_{i,j}]$, where $d_{i,j}$ is the distance from v_i to v_j, and apply the definition (2.9). The distance matrix resulting from the Floyd-Roy-Warshall algorithm

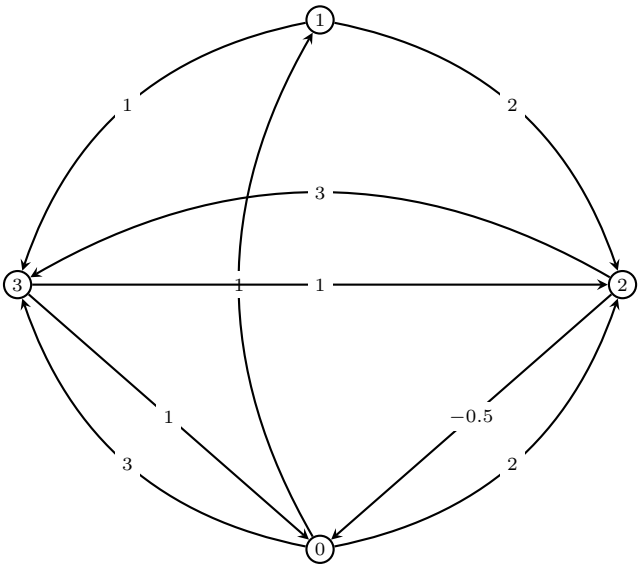


Figure 2.12: Another demonstration of the Floyd-Roy-Warshall algorithm.

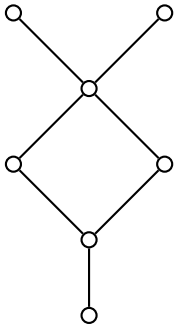


Figure 2.13: Molecular graph of 1,1,3-trimethylcyclobutane.

is

$$M = \begin{bmatrix} 0 & 2 & 1 & 2 & 3 & 2 & 4 \\ 2 & 0 & 1 & 2 & 3 & 2 & 4 \\ 1 & 1 & 0 & 1 & 2 & 1 & 3 \\ 2 & 2 & 1 & 0 & 1 & 2 & 2 \\ 3 & 3 & 2 & 1 & 0 & 1 & 1 \\ 2 & 2 & 1 & 2 & 1 & 0 & 2 \\ 4 & 4 & 3 & 2 & 1 & 2 & 0 \end{bmatrix}.$$

Sum all entries in the upper (or lower) triangular of M to obtain the Wiener number $W = 42$. Using Sage, we have

```
sage: G = Graph({1:[3], 2:[3], 3:[4,6], 4:[5], 6:[5], 5:[7]})
sage: D = G.shortest_path_all_pairs()[0]
sage: M = [D[i][j] for i in D for j in D[i]]
sage: M = matrix(M, nrows=7, ncols=7)
sage: W = 0
sage: for i in range(M.nrows() - 1):
...       for j in range(i+1, M.ncols()):
...           W += M[i,j]
sage: W
42
```

which verifies our computation above. See [**?**] for a survey of some results concerning the Wiener number. ■

2.6.1 Transitive closure

Consider a digraph $G = (V, E)$ of order $n = |V|$. The *transitive closure* of G is defined as the digraph $G^* = (V, E^*)$ having the same vertex set as G. However, the edge set E^* of G^* consists of all edges uv such that there is a u-v path in G and $uv \notin E$. The transitive closure G^* answers an important question about G: If u and v are two distinct vertices of G, are they connected by a path with length ≥ 1?

To compute the transitive closure of G, we let each edge of G be of unit weight and apply the Floyd-Roy-Warshall Algorithm 2.8 on G. By Proposition 1.13, for any i-j path in G we have $D[i, j] < n$, and if there are no paths from i to j in G, we have $D[i, j] = \infty$. This procedure for computing transitive closure runs in time $O(n^3)$.

Modifying the Floyd-Roy-Warshall algorithm slightly, we obtain an algorithm for computing transitive closure that, in practice, is more efficient than Algorithm 2.8 in terms of time and space. Instead of using the operations min and + as is the case in the Floyd-Roy-Warshall algorithm, we replace these operations with the logical operations $\vee$ (logical `OR`) and $\wedge$ (logical `AND`), respectively. For $i, j, k = 1, 2, \ldots, n$, define $T_k(i, j) = 1$ if there is an i-j path in G with all intermediate vertices belonging to $\{1, 2, \ldots, k\}$, and $T_k(i, j) = 0$ otherwise. Thus, the edge ij belongs to the transitive closure G^* if and only if $T_k(i, j) = 1$. The definition of $T_k(i, j)$ can be cast in the form of a recursive definition as follows. For $k = 0$, we have

$$T_0(i, j) = \begin{cases} 0, & \text{if } i \neq j \text{ and } ij \notin E, \\ 1, & \text{if } i = j \text{ or } ij \in E \end{cases}$$

and for $k > 0$, we have

$$T_k(i, j) = T_{k-1}(i, j) \vee \big(T_{k-1}(i, k) \wedge T_{k-1}(k, j)\big).$$

We need not use the subscript k at all and instead let T be a boolean matrix such that $T[i, j] = 1$ if and only if there is an i-j path in G, and $T[i, j] = 0$ otherwise. Using

the above notations, the Floyd-Roy-Warshall algorithm is translated to Algorithm 2.9 for obtaining the boolean matrix T. We can then use T and the definition of transitive closure to obtain the edge set E^* in the transitive closure $G^* = (V, E^*)$ of $G = (V, E)$.

A more efficient transitive closure algorithm can be found in the PhD thesis of Esko Nuutila [**?**]. See also the method of four Russians [**?**, **?**]. The transitive closure algorithm as presented in Algorithm 2.9 is due to Stephen Warshall [**?**]. It is a special case of a more general algorithm in automata theory due to Stephen Kleene [**?**], called Kleene's algorithm.

Algorithm 2.9: Variant of the Floyd-Roy-Warshall algorithm for transitive closure.

Input: A digraph $G = (V, E)$ that has no self-loops. Vertices are numbered from 1 to n, i.e. $V = \{1, 2, \ldots, n\}$.

Output: The boolean matrix T such that $T[i, j] = 1$ if and only if there is an i-j path in G, and $T[i, j] = 0$ otherwise.

```
1 n ← |V|
2 T ← adjacency matrix of G
3 for k ← 1, 2, ..., n do
4     for i ← 1, 2, ..., n do
5         for j ← 1, 2, ..., n do
6             T[i, j] ← T[i, j] ∨ (T[i, k] ∧ T[k, j])
7 return T
```

2.7 Johnson's algorithm

> The shortest distance between two points is under construction.
> — Noelie Altito

Let $G = (V, E)$ be a sparse digraph with edge weights but no negative cycles. *Johnson's algorithm* [**?**] finds a shortest path between each pair of vertices in G. First published in 1977 by Donald B. Johnson, the main insight of Johnson's algorithm is to combine the technique of edge reweighting with the Bellman-Ford and Dijkstra's algorithms. The Bellman-Ford algorithm is first used to ensure that G has no negative cycles. Next, we reweight edges in such a manner as to preserve shortest paths. The final stage makes use of Dijkstra's algorithm for computing shortest paths between all vertex pairs. Pseudocode for Johnson's algorithm is presented in Algorithm 2.10. With a Fibonacci heap implementation of the minimum-priority queue, the time complexity for sparse graphs is $O(|V|^2 \log |V| + |V| \cdot |E|)$, where $n = |V|$ is the number of vertices of the original graph G.

To prove the correctness of Algorithm 2.10, we need to show that the new set of edge weights produced by $\hat{w}$ must satisfy two properties:

1. The reweighted edges preserve shortest paths. That is, let p be a u-v path for $u, v \in V$. Then p is a shortest weighted path using weight function w if and only if p is also a shortest weighted path using weight function $\hat{w}$.

2. The reweight function $\hat{w}$ produces nonnegative weights. In other words, if $u, v \in V$ then $\hat{w}(uv) \geq 0$.

Algorithm 2.10: Johnson's algorithm for sparse graphs.

Input: A sparse weighted digraph $G = (V, E)$, where the vertex set is $V = \{1, 2, \ldots, n\}$.

Output: If G has negative-weight cycles, then return False. Otherwise, return an $n \times n$ matrix D of shortest-path weights and a list P such that $P[v]$ is a parent list resulting from running Dijkstra's algorithm on G with start vertex v.

```
1  s ← vertex not in V
2  V' ← V ∪ {s}
3  E' ← E ∪ {sv | v ∈ V}
4  G' ← digraph (V', E') with weight w(sv) = 0 for all v ∈ V
5  if BellmanFord(G', w, s) = False then
6      return False
7  d ← distance list returned by BellmanFord(G', w, s)
8  for each edge uv ∈ E' do
9      ŵ(uv) ← w(uv) + d[u] − d[v]
10 for each u ∈ V do
11     (δ̂, P̂) ← distance and parent lists returned by Dijkstra(G, ŵ, u)
12     P[u] ← P̂
13     for each v ∈ V do
14         D[u, v] ← δ̂[v] + d[v] − d[u]
15 return (D, P)
```

Both of these properties are proved in Lemma 2.10.

Lemma 2.10. ***Reweighting preserves shortest paths.*** *Let $G = (V, E)$ be a weighted digraph having weight function $w : E \to \mathbf{R}$ and let $h : V \to \mathbf{R}$ be a mapping of vertices to real numbers. Let $\hat{w}$ be another weight function for G such that*

$$\hat{w}(uv) = w(uv) + h(u) - h(v)$$

for all $uv \in E$. Suppose $p : v_0, v_1, \ldots, v_k$ is any path in G. Then we have the following results.

1. *The path p is a shortest v_0-v_k path with weight function w if and only if it is a shortest v_0-v_k path with weight function $\hat{w}$.*

2. *The graph G has a negative cycle using weight function w if and only if G has a negative cycle using $\hat{w}$.*

3. *If G has no negative cycles, then $\hat{w}(uv) \geq 0$ for all $uv \in E$.*

Proof. Write δ and $\hat{\delta}$ for the shortest path weights derived from w and $\hat{w}$, respectively. To prove part 1, we need to show that $w(p) = \delta(v_0, v_k)$ if and only if $\hat{w}(p) = \hat{\delta}(v_0, v_k)$.

First, note that any v_0-v_k path p satisfies $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$ because

$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}v_i) \\
&= \sum_{i=1}^{k} \big(w(v_{i-1}v_i) + h(v_{i-1}) - h(v_i)\big) \\
&= \sum_{i=1}^{k} w(v_{i-1}v_i) + \sum_{i=1}^{k} \big(h(v_{i-1}) - h(v_i)\big) \\
&= \sum_{i=1}^{k} w(v_{i-1}v_i) + h(v_0) - h(v_k) \\
&= w(p) + h(v_0) - h(v_k).
\end{aligned}$$

Any v_0-v_k path shorter than p and using weight function w is also shorter using $\hat{w}$. Therefore, $w(p) = \delta(v_0, v_k)$ if and only if $\hat{w}(p) = \hat{\delta}(v_0, v_k)$.

To prove part 2, consider any cycle $c : v_0, v_1, \ldots, v_k$ where $v_0 = v_k$. Using the proof of part 1, we have

$$\begin{aligned}
\hat{w}(c) &= w(c) + h(v_0) - h(v_k) \\
&= w(c)
\end{aligned}$$

thus showing that c is a negative cycle using w if and only if it is a negative cycle using $\hat{w}$.

To prove part 3, we construct a new graph $G' = (V', E')$ as follows. Consider a vertex $s \notin V$ and let $V' = V \cup \{s\}$ and $E' = E \cup \{sv \mid v \in V\}$. Extend the weight function w to include $w(sv) = 0$ for all $v \in V$. By construction, s has no incoming edges and any path in G' that contains s has s as the source vertex. Thus G' has no negative cycles if and only if G has no negative cycles. Define the function $h : V \to \mathbf{R}$ by $v \mapsto \delta(s, v)$ with domain V'. By the triangle inequality (see Lemma 2.5),

$$\delta(s, u) + w(uv) \geq \delta(s, v) \quad \iff \quad h(u) + w(uv) \geq h(v)$$

thereby showing that $\hat{w}(uv) = w(uv) + h(u) - h(v) \geq 0$. ■

2.8 Problems

> I believe that a scientist looking at nonscientific problems is just as dumb as the next guy.
> — Richard Feynman

2.1. The Euclidean algorithm is one of the oldest known algorithms. Given two positive integers a and b with $a \geq b$, let $a \bmod b$ be the remainder obtained upon dividing a by b. The Euclidean algorithm determines the greatest common divisor $\gcd(a, b)$ of a and b. The procedure is summarized in Algorithm 2.11. Refer to [**?**] for a history of algorithms from ancient to modern times.

(a) Implement Algorithm 2.11 in Sage and use your implementation to compute the greatest common divisors of various pairs of integers. Use the built-in Sage command `gcd` to verify your answer.

Algorithm 2.11: The Euclidean algorithm.

Input: Two integers $a > 0$ and $b \geq 0$ with $a \geq b$.
Output: The greatest common divisor of a and b.

```
1 x ← a
2 y ← b
3 while y ≠ 0 do
4     r ← x mod y
5     x ← y
6     y ← r
7 return x
```

(b) Modify Algorithm 2.11 to compute the greatest common divisor of any pair of integers.

2.2. Given a polynomial $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ of degree n, we can use Horner's method [**?**] to efficiently evaluate p at a specific value $x = x_0$. Horner's method evaluates $p(x)$ by expressing the polynomial as

$$p(x) = \sum_{i=0}^{n} a_i x^i = (\cdots(a_n x + a_{n-1})x + \cdots)x + a_0$$

to obtain Algorithm 2.12.

(a) Compare the runtime of polynomial evaluation using equation (2.2) and Horner's method.

(b) Let v be a bit vector read using big-endian order. Write a Sage function that uses Horner's method to compute the integer representation of v.

(c) Modify Algorithm 2.12 to evaluate the integer representation of a bit vector v read using little-endian order. Hence, write a Sage function to convert v to its integer representation.

Algorithm 2.12: Polynomial evaluation using Horner's method.

Input: A polynomial $p(x) = \sum_{i=0}^{n} a_i x_i$, where $a_n \neq 0$ and $x_0 \in \mathbf{R}$.
Output: An evaluation of p at $x = x_0$.

```
1 b ← a_n
2 for i ← n − 1, n − 2, . . . , 0 do
3     b ← bx_0 + a_i
4 return b
```

2.3. Let $G = (V, E)$ be an undirected graph, let $s \in V$, and D is the list of distances resulting from running Algorithm 2.1 with G and s as input. Show that G is connected if and only if $D[v]$ is defined for each $v \in V$.

2.4. Show that the worst-case time complexity of depth-first search Algorithm 2.2 is $O(|V| + |E|)$.

2.5. Let D be the list of distances returned by Algorithm 2.2, let s be a starting vertex, and let v be a vertex such that $D[v] \neq \infty$. Show that $D[v]$ is the length of any shortest path from s to v.

2.6. Consider the graph in Figure 2.10 as undirected. Run this undirected version through Dijkstra's algorithm with starting vertex v_1.

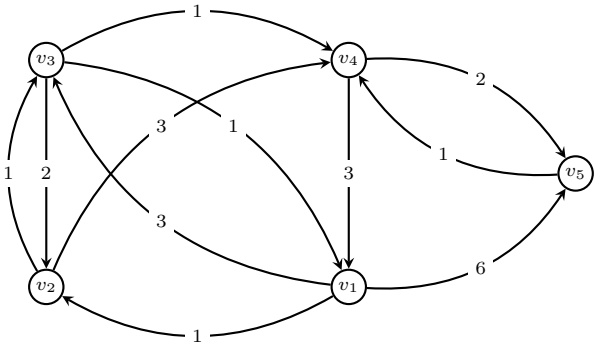


Figure 2.14: Searching a directed house graph using Dijkstra's algorithm.

2.7. Consider the graph in Figure 2.14. Choose any vertex as a starting vertex and run Dijkstra's algorithm over it. Now consider the undirected version of that digraph and repeat the exercise.

2.8. For each vertex v of the graph in Figure 2.14, run breadth-first search over that graph with v as the starting vertex. Repeat the exercise for depth-first search. Compare the graphs resulting from the above exercises.

2.9. A list data structure can be used to manage vertices and edges. If L is a nonempty list of vertices of a graph, we may want to know whether the graph contains a particular vertex. We could search the list L, returning `True` if L contains the vertex in question and `False` otherwise. Linear search is a simple searching algorithm. Given an object E for which we want to search, we consider each element e of L in turn and compare E to e. If at any point during our search we found a match, we halt the algorithm and output an affirmative answer. Otherwise, we have scanned through all elements of L and each of those elements do not match E. In this case, linear search reports that E is not in L. Our discussion is summarized in Algorithm 2.13.

(a) Implement Algorithm 2.13 in Sage and test your implementation using the graphs presented in the figures of this chapter.

(b) What is the maximum number of comparisons during the running of Algorithm 2.13? What is the average number of comparisons?

(c) Why must the input list L be nonempty?

2.10. Binary search is a much faster searching algorithm than linear search. The binary search algorithm assumes that its input list is ordered in some manner. For simplicity, we assume that the input list L consists of positive integers. The main idea of binary search is to partition L into two halves: the left half and the right half. Our task now is to determine whether the object E of interest is in the left half or

Algorithm 2.13: Linear search for lists.

Input: A nonempty list L of vertices or edges. An object E for which we want to search in L.

Output: True if E is in L; False otherwise.

```
1  for each e ∈ L do
2      if E = e then
3          return True
4  return False
```

Algorithm 2.14: Binary search for lists of positive integers.

Input: A nonempty list L of positive integers. Elements of L are sorted in nondecreasing order. An integer i for which we want to search in L.

Output: True if i is in L; False otherwise.

```
1  low ← 0
2  high ← |L| − 1
3  while low ≤ high do
4      mid ← ⌊(low + high)/2⌋
5      if i = L[mid] then
6          return True
7      if i < L[mid] then
8          high ← mid − 1
9      else
10         low ← mid + 1
11 return False
```

the right half, and apply binary search recursively to the half in which E is located. Algorithm 2.14 provides pseudocode of our discussion of binary search.

(a) Implement Algorithm 2.14 in Sage and test your implementation using the graphs presented in the figures of this chapter.

(b) What is the worst case runtime of Algorithm 2.14? How does this compare to the worst case runtime of linear search?

(c) Why must the input list L be sorted in nondecreasing order? Would Algorithm 2.14 work if L is sorted in nonincreasing order? If not, modify Algorithm 2.14 so that it works with an input list that is sorted in nonincreasing order.

(d) Line 4 of Algorithm 2.14 uses the floor function to compute the index of the middle value. Would binary search still work if we use the ceiling function instead of the floor function?

(e) An improvement on the time complexity of binary search is to not blindly use the middle value of the interval of interest, but to guess more precisely where the target falls within this interval. Interpolation search uses this heuristic to improve on the runtime of binary search. Provide an algorithm for interpolation search, analyze its worst-case runtime, and compare its theoretical runtime efficiency with that of binary search (see pages 419–420 in [**?**] and pages 201–202 in [**?**]).

2.11. Let G be a simple undirected graph having distance matrix $D = [d(v_i, v_j)]$, where $d(v_i, v_j) \in \mathbf{R}$ denotes the shortest distance from $v_i \in V(G)$ to $v_j \in V(G)$. If $v_i = v_j$, we set $d(v_i, v_j) = 0$. For each pair of distinct vertices (v_i, v_j), we have $d(v_i, v_j) = d(v_j, v_i)$. The i-j entry of D is also written as $d_{i,j}$ and denotes the entry in row i and column j.

(a) The *total distance* $\mathrm{td}(u)$ of a fixed vertex $u \in V(G)$ is the sum of distances from u to each vertex in G:

$$\mathrm{td}(u) = \sum_{v \in V(G)} d(u, v).$$

If G is connected, i is the row index of vertex u in the distance matrix D, and j is the column index of u in D, show that the total distance of u is

$$\mathrm{td}(u) = \sum_{k} d_{i,k} = \sum_{k} d_{k,j}. \tag{2.10}$$

(b) Let the vertices of G be labeled $V = \{v_1, v_2, \ldots, v_n\}$, where $n = |V(G)|$ is the order of G. The *total distance* $\mathrm{td}(G)$ of G is obtained by summing all the $d(v_i, v_j)$ for $i < j$. If G is connected, show that the total distance of G is equal to the sum of all entries in the upper (or lower) triangular of D:

$$\mathrm{td}(G) = \sum_{i<j} d_{i,j} = \sum_{i>j} d_{i,j} = \frac{1}{2}\left(\sum_{u \in V}\sum_{v \in V} d(u, v)\right). \tag{2.11}$$

Hence show that the total distance of G is equal to its Wiener number:

$$\mathrm{td}(G) = W(G).$$

(c) Would equations (2.10) and (2.11) hold if G is not connected or directed?

2.12. The following result is from [**?**]. Let G_1 and G_2 be graphs with orders $n_i = |V(G_i)|$ and sizes $m_i = |E(G_i)|$, respectively.

(a) If each of G_1 and G_2 is connected, show that the Wiener number of the Cartesian product $G_1 \square G_2$ is

$$W(G_1 \square G_2) = n_2^2 \cdot W(G_1) + n_1^2 \cdot W(G_2).$$

(b) If G_1 and G_2 are arbitrary graphs, show that the Wiener number of the join $G_1 + G_2$ is

$$W(G_1 + G_2) = n_1^2 - n_1 + n_2^2 - n_2 + n_1 n_2 - m_1 - m_2.$$

2.13. The following results originally appeared in [**?**] and independently rediscovered many times since.

(a) If P_n is the path graph on $n \geq 0$ vertices, show that the Wiener number of P_n is $W(P_n) = \frac{1}{6}n(n^2 - 1)$.

(b) If C_n is the cycle graph on $n \geq 0$ vertices, show that the Wiener number of C_n is

$$W(C_n) = \begin{cases} \frac{1}{8}n(n^2 - 1), & \text{if } n \text{ is odd,} \\ \frac{1}{8}n^3, & \text{if } n \text{ is even.} \end{cases}$$

(c) If K_n is the complete graph on n vertices, show that its Wiener number is $W(K_n) = \frac{1}{2}n(n-1)$.

(d) Show that the Wiener number of the complete bipartite graph $K_{m,n}$ is

$$W(K_{m,n}) = mn + m(m-1) + n(n-1).$$

2.14. Consider the world map of major capital cities in Figure 2.16.

(a) Run breadth- and depth-first searches over the graph in Figure 2.16 and compare your results.

(b) Convert the graph in Figure 2.16 to a digraph as follows. Let $0 \leq \alpha \leq 1$ be a fixed threshold probability and let $V = \{v_1, \ldots, v_n\}$ be the vertex set of the graph. For each edge $v_i v_j$, let $0 \leq p \leq 1$ be its orientation probability and define the directedness $\mathrm{dir}(v_i, v_j)$ by

$$\mathrm{dir}(v_i, v_j) = \begin{cases} v_i v_j, & \text{if } p \leq \alpha, \\ v_j v_i, & \text{otherwise.} \end{cases}$$

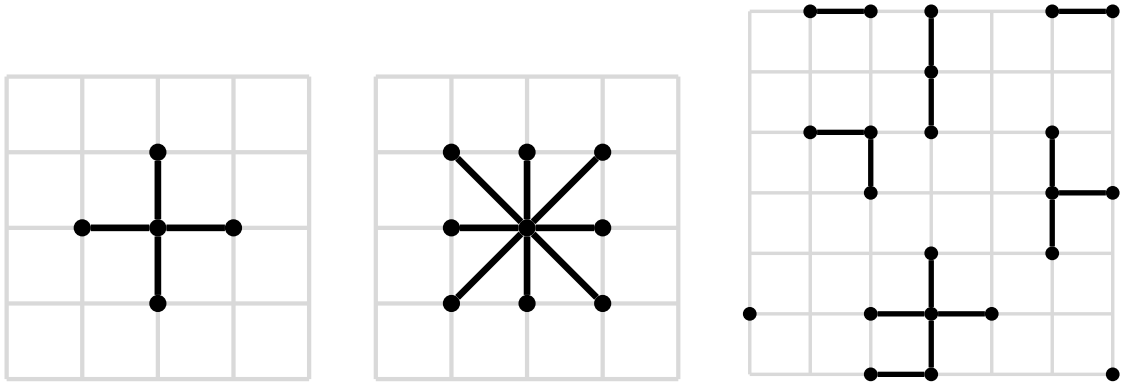
That is, $\mathrm{dir}(v_i, v_j)$ takes the endpoints of an undirected edge $v_i v_j$ and returns a directed version of this edge. The result is either the directed edge $v_i v_j$ or the directed edge $v_j v_i$. Use the above procedure to convert the graph of Figure 2.16 to a digraph, and run breadth- and depth-first searches over the resulting digraph.

(c) Table 2.6 lists distances in kilometers between the major capital cities shown in Figure 2.16. Let those distances be the weights of the edges shown in Figure 2.16. Repeatedly run Dijkstra's algorithm over this undirected, weighted graph with each city as the start vertex. Use the procedure from the previous exercise to obtain a directed version of the graph in Figure 2.16 and repeat the current exercise with the resulting weighted digraph.

(d) Repeat the previous exercise, but use the Bellman-Ford algorithm instead of Dijkstra's algorithm. Compare the results you obtain using these two different algorithms.

(e) Consider a weighted digraph version of the graph in Figure 2.16. Run the Floyd-Roy-Warshall and Johnson's algorithms over the weighted digraph. Compare the results output by these two algorithms to the results of the previous exercise.

2.15. Consider a finite square lattice Λ where points on Λ can connect to other points in their von Neumann or Moore neighborhoods, or other points in the lattice. See Figures 2.15(a) and 2.15(b) for illustrations of the von Neumann and Moore neighborhoods of a central node, respectively. We treat each point as a node and forbid each node from connecting to itself. The whole lattice can be considered as a simple undirected graph Γ. Let n be the maximum number of nodes in any connected components. For each $i \in \{1, 2, \ldots, n\}$, how many connected components have exactly n nodes? In other words, what is the density of connected components in Γ? For example, consider the lattice in Figure 2.15(c). Denote by $\kappa(i)$ the number of connected components with i nodes. Then the maximum number of nodes of any connected components is $n = 6$, with $\kappa(1) = 2$, $\kappa(2) = 2$, $\kappa(3) = 2$, $\kappa(4) = 1$, $\kappa(5) = 0$, and $\kappa(6) = 1$.



(a) Von Neumann neighborhood.

(b) Moore neighborhood.

(c) Square lattice with components.

Figure 2.15: Component density in a square lattice.

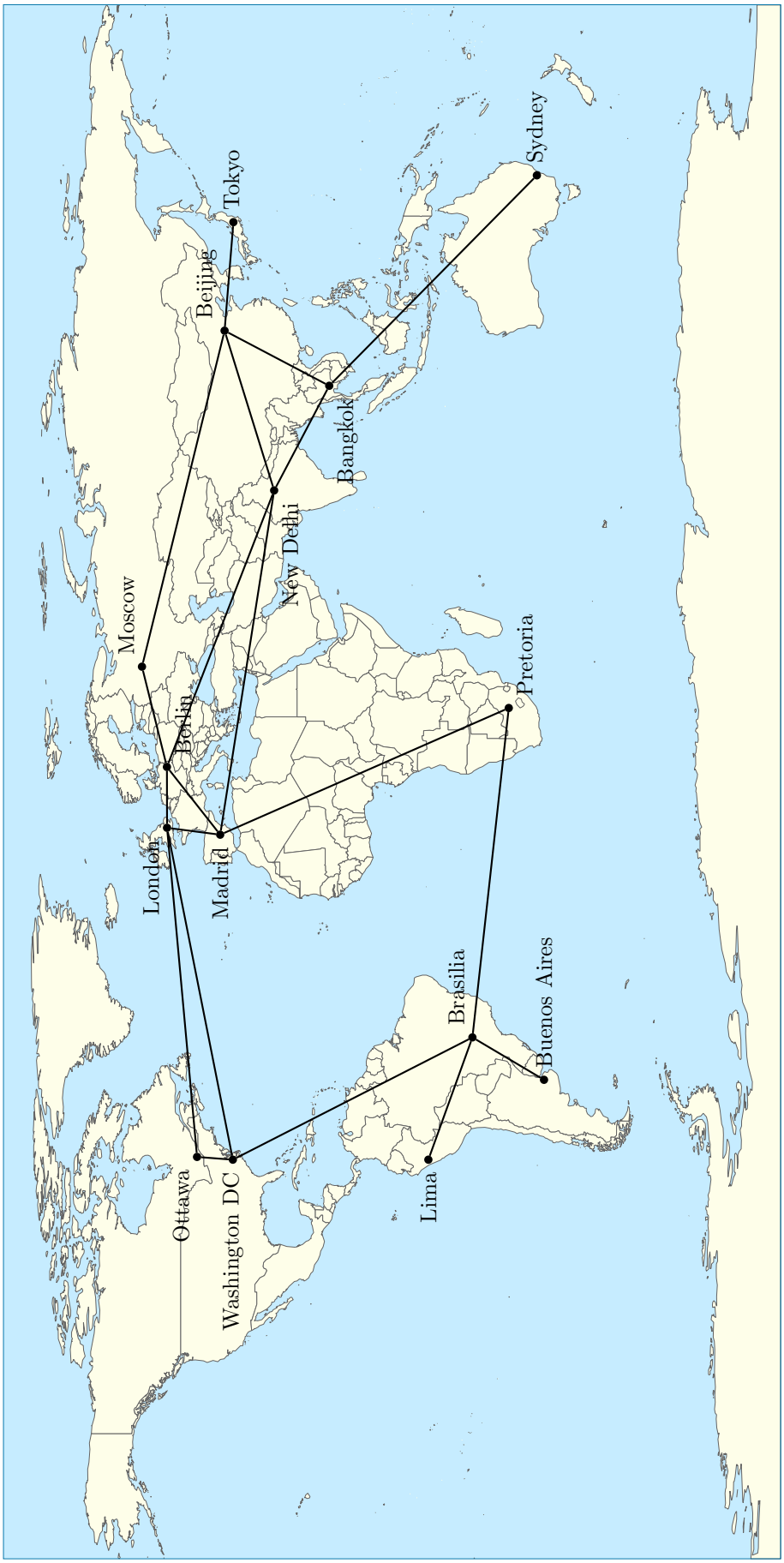


Figure 2.16: Major capital cities of the world.

	Bangkok	Beijing	Berlin	Brasilia	Buenos Aires	Lima	London	Madrid	Moscow	New Delhi	Ottawa	Pretoria	Sydney	Tokyo	Washington DC
Bangkok		3282								2908			7540		
Beijing	3282								5807	3784				2103	
Berlin							929	1866	1619	5796					
Brasilia					2314	3154						7906			6764
Buenos Aires					2314										
Lima				3154											
London			929					1261			5379				5915
Madrid			1866				1261			7288		8033			
Moscow		5807	1619												
New Delhi	2908	3784	5796					7288							
Ottawa							5379								734
Pretoria				7906				8033							
Sydney	7540														
Tokyo		2103													
Washington DC				6764			5915				734				

Table 2.6: Distances in kilometers between major world capital cities.

2.16. Various efficient search techniques exist that cater for special situations. Some of these are covered in chapter 6 in [**?**] and chapters 14–18 in [**?**]. Investigate an algorithm for and time complexity of trie search. Hashing techniques can result in searches that run in $O(1)$ time. Furthermore, hashing has important applications outside of the searching problem, a case in point being cryptology. Investigate how hashing can be used to speed up searches. For further information on hashing and its application to cryptology, see [**?**, **?**, **?**].

2.17. In addition to searching, there is the related problem of sorting a list according to an ordering relation. If the given list $L = [e_1, e_2, \ldots, e_n]$ consists of real numbers, we want to order elements in nondecreasing order. *Bubble sort* is a basic sorting algorithm that can be used to sort a list of real numbers, indeed any collection of objects that can be ordered according to an ordering relation. During each pass through the list L from left to right, we consider e_i and its right neighbor e_{i+1}. If $e_i \leq e_{i+1}$, then we move on to consider e_{i+1} and its right neighbor e_{i+2}. If $e_i > e_{i+1}$, then we swap these two values around in the list and then move on to consider e_{i+1} and its right neighbor e_{i+2}. Each successive pass pushes to the right end an element that is the next largest in comparison to the previous largest element pushed to the right end. Hence the name bubble sort for the algorithm. Algorithm 2.15 summarizes our discussion.

Algorithm 2.15: Bubble sort.

Input: A list L of $n > 1$ elements that can be ordered using the "less than or equal to" relation "$\leq$".

Output: The same list as L, but sorted in nondecreasing order.

```
1 for i ← n, n − 1, ..., 2 do
2     for j ← 2, 3, ..., i do
3         if L[j − 1] > L[j] then
4             swap the values of L[j − 1] and L[j]
5 return L
```

(a) Analyze the worst-case runtime of Algorithm 2.15.

(b) Modify Algorithm 2.15 so that it sorts elements in nonincreasing order.

(c) Line 4 of Algorithm 2.15 is where elements are actually sorted. Swapping the values of two elements is such a common task that sometimes we want to perform the swap as efficiently as possible, i.e. using as few operations as possible. A common way to swap the values of two elements a and b is to create a temporary placeholder t and realize the swap as presented in Algorithm 2.16. Some programming languages allow for swapping the values of two elements without creating a temporary placeholder for an intermediary value. Investigate how to realize this swapping method using a programming language of your choice.

2.18. *Selection sort* is another simple sorting algorithm that works as follows. Let $L = [e_1, e_2, \ldots, e_n]$ be a list of elements that can be ordered according to the relation "$\leq$", e.g. the e_i can all be real numbers or integers. On the first scan of L from left to right, among the elements $L[2], \ldots, L[n]$ we find the smallest element

Algorithm 2.16: Swapping values using a temporary placeholder.

Input: Two objects a and b.
Output: The values of a and b swapped with each other.

```
1 t ← a
2 a ← b
3 b ← t
```

and exchange it with $L[1]$. On the second scan, we find the smallest element among $L[3], \ldots, L[n]$ and exchange that smallest element with $L[2]$. In general, during the i-th scan we find the smallest element among $L[i+1], \ldots, L[n]$ and exchange that with $L[i]$. At the end of the i-th scan, the element $L[i]$ is in its final position and would not be processed again. When the index reaches $i = n$, the list would have been sorted in nondecreasing order. The procedure is summarized in Algorithm 2.17.

(a) Analyze the worst-case runtime of Algorithm 2.17 and compare your result to the worst-case runtime of the bubble sort Algorithm 2.15.

(b) Modify Algorithm 2.17 to sort elements in nonincreasing order.

(c) Line 6 of Algorithm 2.17 assumes that among $L[i+1], L[i+2], \ldots, L[n]$ there is a smallest element $L[k]$ such that $L[i] > L[k]$, hence we perform the swap. It is possible that $L[i] < L[k]$, obviating the need to carry out the value swapping. Modify Algorithm 2.17 to take account of our discussion.

Algorithm 2.17: Selection sort.

Input: A list L of $n > 1$ elements that can be ordered using the relation "$\leq$".
Output: The same list as L, but sorted in nondecreasing order.

```
1 for i ← 1, 2, ..., n − 1 do
2     min ← i
3     for j ← i + 1, i + 2, ..., n do
4         if L[j] < L[min] then
5             min ← j
6     swap the values of L[min] and L[i]
7 return L
```

2.19. In addition to bubble and selection sort, other algorithms exist whose runtime is more efficient than these two basic sorting algorithms. Chapter 5 in [**?**] describes various efficient sorting techniques. See also chapters 8–13 in [**?**].

(a) Investigate and provide pseudocode for insertion sort and compare its runtime efficiency with that of selection sort. Compare the similarities and differences between insertion and selection sort.

(b) Shellsort is a variation on insertion sort that can speed up the runtime of insertion sort. Describe and provide pseudocode for shellsort. Compare the time complexity of shellsort with that of insertion sort. In what ways is shellsort different from or similar to insertion sort?

(c) The quicksort algorithm due to C. A. R. Hoare [**?**] was published in 1962. Describe and provide pseudocode for quicksort and compare its runtime complexity with the other sorting algorithms covered in this chapter.

2.20. Algorithm 2.3 uses breadth-first search to determine the connectivity of an undirected graph. Modify this algorithm to use depth-first search. How can Algorithm 2.3 be used or modified to test the connectivity of a digraph?

2.21. The following problem is known as the *river crossing problem*. A man, a goat, a wolf, and a basket of cabbage are all on one side of a river. They have a boat that could be used to cross to the other side of the river. The boat can only hold at most two passengers, one of whom must be able to row the boat. One of the two passengers must be the man and the other passenger can be either the goat, the wolf, or the basket of cabbage. When crossing the river, if the man leaves the wolf with the goat, the wolf would prey on the goat. If he leaves the goat with the basket of cabbage, the goat would eat the cabbage. The objective is to cross the river in such a way that the wolf has no chance of preying on the goat, nor that the goat eat the cabbage.

(a) Let M, G, W, and C denote the man, the goat, the wolf, and the basket of cabbage, respectively. Initially all four are on the left side of the river and none of them are on the right side. Denote this by the ordered pair ($MGWC$, _), which is called the initial state of the problem. When they have all crossed to the right side of the river, the final state of the problem is (_, $MGWC$). The underscore "_" means that neither M, G, W, nor C are on the corresponding side of the river. List a finite sequence of moves to get from ($MGWC$, _) to (_, $MGWC$). Draw your result as a digraph.

(b) In the digraph Γ obtained from the previous exercise, let each edge of Γ be of unit weight. Find a shortest path from ($MGWC$, _) to (_, $MGWC$).

(c) Rowing from one side of the river to the other side is called a crossing. What is the minimum number of crossings needed to get from ($MGWC$, _) to (_, $MGWC$)?

2.22. Symbolic computation systems such as Magma, Maple, Mathematica, Maxima, and Sage are able to read in a symbolic expression such as

```
(a + b)^2 - (a - b)^2
```

and determine whether or not the brackets match. A bracket is any of the following characters:

```
( ) { } [ ]
```

A string S of characters is said to be *balanced* if any left bracket in S has a corresponding right bracket that is also in S. Furthermore, if there are k occurrences of one type of left bracket, then there must be k occurrences of the corresponding right bracket. The *balanced bracket problem* is concerned with determining whether or not the brackets in S are balanced. Algorithm 2.18 contains a procedure to determine if the brackets in S are balanced, and if so return a list of positive integers to indicate how the brackets match.

(a) Implement Algorithm 2.18 in Sage and test your implementation on various strings containing brackets. Test your implementation on nonempty strings without any brackets.

(b) Modify Algorithm 2.18 so that it returns `True` if the brackets of an input string are balanced, and returns `False` otherwise.

(c) What is the worst-case runtime of Algorithm 2.18?

Algorithm 2.18: A brackets parser.

Input: A nonempty string S of characters.
Output: A list L of positive integers indicating how the brackets match. If the brackets are not balanced, return the empty string ε.

```
1  L ← [ ]
2  T ← empty stack
3  c ← 1
4  n ← |S|
5  for i ← 0, 1, ..., n do
6      if S[i + 1] is a left bracket then
7          append(L, c)
8          push (S[i + 1], c) onto T
9          c ← c + 1
10     if S[i + 1] is a right bracket then
11         if T is empty then
12             return ε
13         (left, d) ← pop(T)
14         if left matches S[i + 1] then
15             append(L, d)
16         else
17             return ε
18 if T is empty then
19     return L
20 return ε
```

2.23. An arithmetic expression written in the form $a + b$ is said to be in *infix notation* because the operator is in between the operands. The same expression can also be written in *reverse Polish notation* (or *postfix notation*) as

$$a\ b\ +$$

with the operator following its two operands. Given an arithmetic expression $A = e_0 e_1 \cdots e_n$ written in reverse Polish notation, we can use the stack data structure to evaluate the expression. Let $P = [e_0, e_1, \ldots, e_n]$ be the stack representation of A, where traversing P from left to right we are moving from the top of the stack to the bottom of the stack. We call P the Polish stack and the stack E containing intermediate results the evaluation stack. While P is not empty, pop the Polish stack and assign the extracted result to x. If x is an operator, we pop the evaluation stack twice: the result of the first pop is assigned to b and the result of the second

pop is assigned to a. Compute the infix expression $a\ x\ b$ and push the result onto E. However, if x is an operand, we push x onto E. Iterate the above process until P is empty, at which point the top of E contains the evaluation of A. Refer to Algorithm 2.19 for pseudocode of the above discussion.

(a) Prove the correctness of Algorithm 2.19.

(b) What is the worst-case runtime of Algorithm 2.19?

(c) Modify Algorithm 2.19 to support the exponentiation operator.

Algorithm 2.19: Evaluate arithmetic expressions in reverse Polish notation.

Input: A Polish stack P containing an arithmetic expression in reverse Polish notation.

Output: An evaluation of the arithmetic expression represented by P.

```
1  E ← empty stack
2  v ← NULL
3  while P is not empty do
4      x ← pop(P)
5      if x is an operator then
6          b ← pop(E)
7          a ← pop(E)
8          if x is addition operator then
9              v ← a + b
10         else if x is subtraction operator then
11             v ← a − b
12         else if x is multiplication operator then
13             v ← a × b
14         else if x is division operator then
15             v ← a/b
16         else
17             exit algorithm with error
18         push(E, v)
19     else
20         push(E, x)
21 v ← pop(E)
22 return v
```

2.24. Figure 2.5 provides a knight's tour for the knight piece with initial position as in Figure 2.5(a). By rotating the chessboard in Figure 2.5(b) by $90n$ degrees for positive integer values of n, we obtain another knight's tour that, when represented as a graph, is isomorphic to the graph in Figure 2.5(c).

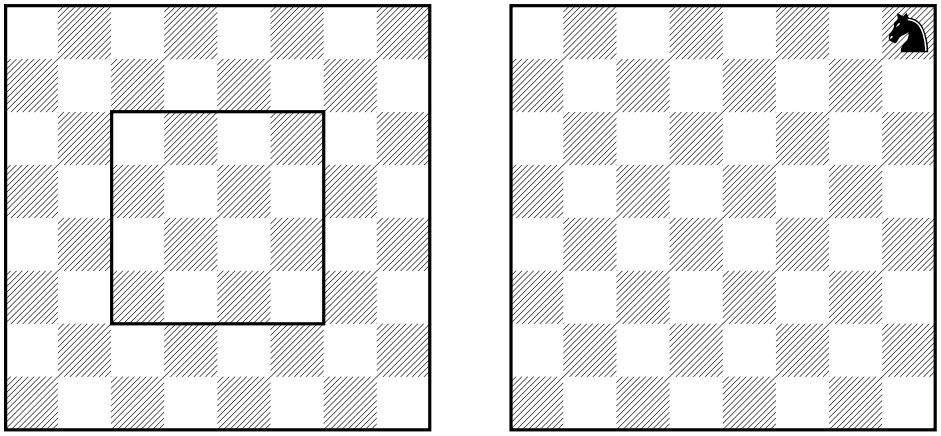
(a) At the beginning of the 18th century, de Montmort and de Moivre provided the following strategy [**?**, p.176] to solve the knight's tour problem on an 8×8 chessboard. Divide the board into an inner 4×4 square and an outer shell of two squares deep, as shown in Figure 2.17(a). Place a knight on a square in the outer shell and move the knight piece around that shell, always in the

same direction, so as to visit each square in the outer shell. After that, move into the inner square and solve the knight's tour problem for the 4×4 case. Apply this strategy to solve the knight's tour problem with the initial position as in Figure 2.17(b).

(b) Use the Montmort-Moivre strategy to obtain a knight's tour, starting at the position of the black-filled node in the outer shell in Figure 2.5(b).

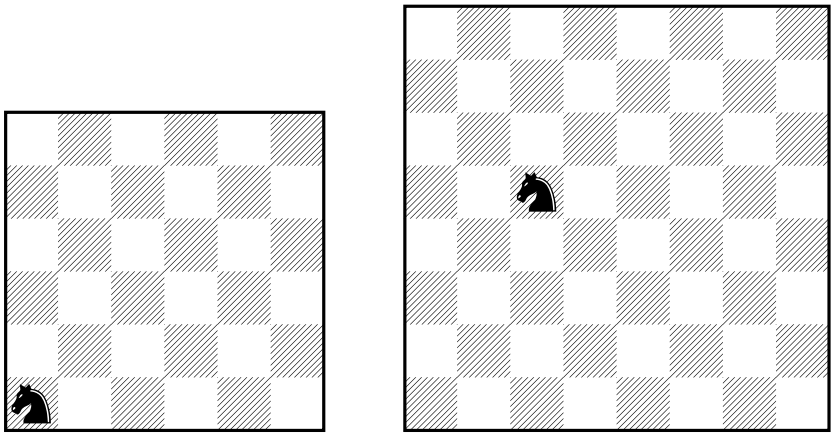
(c) A *re-entrant* or *closed* knight's tour is a knight's tour that starts and ends at the same square. Find re-entrant knight's tours with initial positions as in Figure 2.18.

(d) Devise a backtracking algorithm to solve the knight's tour problem on an $n \times n$ chessboard for $n > 3$.



(a) A 4×4 inner square. (b) Initial position in the outer shell.

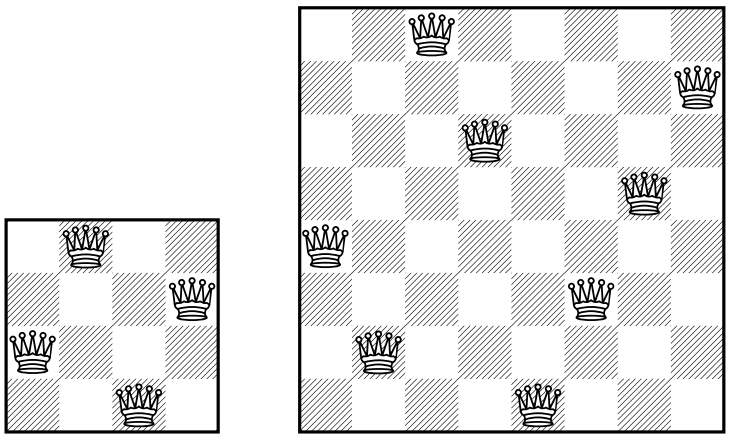
Figure 2.17: De Montmort and de Moivre's solution strategy for the 8×8 knight's tour problem.



(a) A 6×6 chessboard. (b) An 8×8 chessboard.

Figure 2.18: Initial positions of re-entrant knight's tours.

2.25. The n-queens problem is concerned with the placement of n queens on an $n \times n$ chessboard such that no two queens can attack each other. Two queens can attack



(a) $n = 4$ (b) $n = 8$

Figure 2.19: Solutions of the n-queens problem for $n = 4, 8$.

each other if they are in the same row, column, diagonal, or antidiagonal of the chessboard. The trivial case $n = 1$ is easily solved by placing the one queen in the only given position. There are no solutions for the cases $n = 2, 3$. Solutions for the cases $n = 4, 8$ are shown in Figure 2.19. Devise a backtracking algorithm to solve the n-queens problem for the case where $n > 3$. See [**?**] for a survey of the n-queens problem and its solutions.

2.26. Hampton Court Palace in England is well-known for its maze of hedges. Figure 2.20 shows a maze and its graph representation; the figure is adapted from page 434 in [**?**]. To obtain the graph representation, we use a vertex to represent an intersection in the maze. An edge joining two vertices represents a path from one intersection to another.

(a) Suppose the entrance to the maze is represented by the lower-left black-filled vertex in Figure 2.20(b) and the exit is the upper-right black-filled vertex. Solve the maze by providing a path from the entrance to the exit.

(b) Repeat the previous exercise for each pair of distinct vertices, letting one vertex of the pair be the entrance and the other vertex the exit.

(c) What is the diameter of the graph in Figure 2.20(b)?

(d) Investigate algorithms for generating and solving mazes.

2.27. For each of the algorithms below: (i) justify whether or not it can be applied to multigraphs or multidigraphs; (ii) if not, modify the algorithm so that it is applicable to multigraphs or multidigraphs.

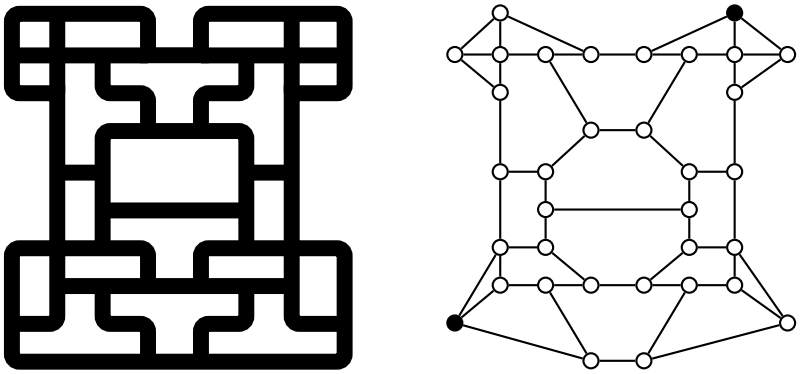
(a) Breadth-first search Algorithm 2.1.

(b) Depth-first search Algorithm 2.2.

(c) Graph connectivity test Algorithm 2.3.

(d) General shortest path Algorithm 2.4.

(e) Dijkstra's Algorithm 2.5.



(a) (b)

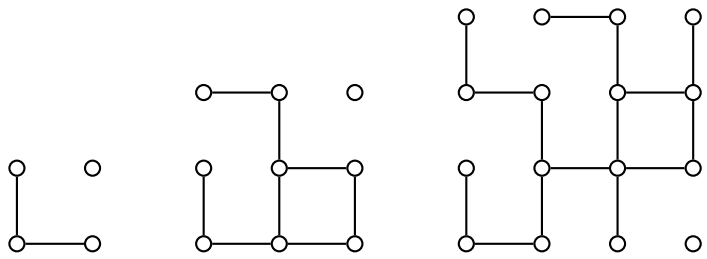
Figure 2.20: A maze and its graph representation.

(f) The Bellman-Ford Algorithms 2.6 and 2.7.

(g) The Floyd-Roy-Warshall Algorithm 2.8.

(h) The transitive closure Algorithm 2.9.

(i) Johnson's Algorithm 2.10.



(a) 2×2 (b) 3×3 (c) 4×4

Figure 2.21: Grid graphs for $n = 2, 3, 4$.

2.28. Let n be a positive integer. An $n \times n$ *grid graph* is a graph on the Euclidean plane, where each vertex is an ordered pair from $\mathbf{Z} \times \mathbf{Z}$. In particular, the vertices are ordered pairs $(i, j) \in \mathbf{Z} \times \mathbf{Z}$ such that

$$0 \leq i, j < n. \tag{2.12}$$

Each vertex (i, j) is adjacent to any of the following vertices provided that expression (2.12) is satisfied: the vertex $(i - 1,\ j)$ immediately to its left, the vertex $(i + 1,\ j)$ immediately to its right, the vertex $(i,\ j + 1)$ immediately above it, or the vertex $(i,\ j - 1)$ immediately below it. Figure 2.21 illustrates some examples of grid graphs. The 1×1 grid graph is the trivial graph K_1.

(a) Fix a positive integer $n > 1$. Describe and provide pseudocode of an algorithm to generate all nonisomorphic $n \times n$ grid graphs. What is the worst-case runtime of your algorithm?

(b) How many $n \times n$ grid graphs are there? How many of those graphs are nonisomorphic to each other?

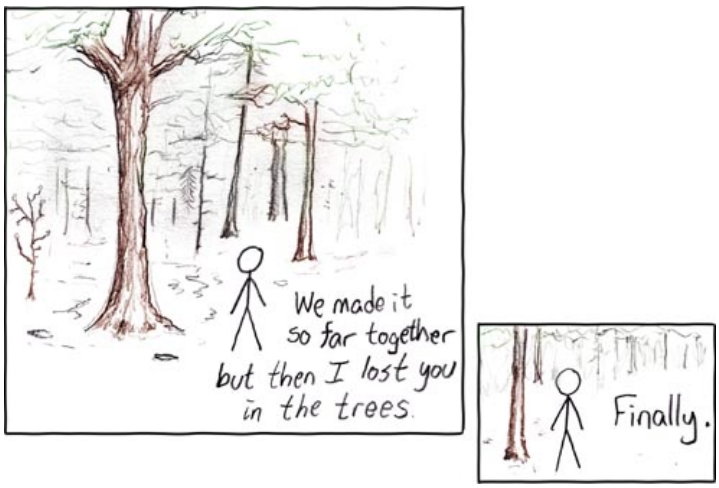
(c) Describe and provide pseudocode of an algorithm to generate a random $n \times n$ grid graph. Analyze the worst-case runtime of your algorithm.

(d) Extend the grid graph by allowing edges to be diagonals. That is, a vertex (i, j) can also be adjacent to any of the following vertices so long as expression (2.12) holds: $(i-1,\, j-1)$, $(i-1,\, j+1)$, $(i+1,\, j+1)$, $(i+1,\, j-1)$. With this extension, repeat the previous exercises.

2.29. Let $G = (V, E)$ be a digraph with integer weight function $w : E \to \mathbf{Z}\backslash\{0\}$, where either $w(e) > 0$ or $w(e) < 0$ for each $e \in E$. Yamada and Kinoshita [**?**] provide a divide-and-conquer algorithm to enumerate all the negative cycles in G. Investigate the divide and conquer technique for algorithm design. Describe and provide pseudocode of the Yamada-Kinoshita algorithm. Analyze its runtime complexity and prove the correctness of the algorithm.

Chapter 3

Trees and forests



— Randall Munroe, xkcd, http://xkcd.com/71/

In section 1.2.1, we briefly touched upon trees and provided examples of how trees could be used to model hierarchical structures. This chapter provides an in-depth study of trees, their properties, and various applications. After defining trees and related concepts in section 3.1, we then present various basic properties of trees in section 3.2. Each connected graph G has an underlying subgraph called a spanning tree that contains all the vertices of G. Spanning trees are discussed in section 3.3 together with various common algorithms for finding spanning trees. We then discuss binary trees in section 3.4, followed by an application of binary trees to coding theory in section 3.5. Whereas breadth- and depth-first searches are general methods for traversing a graph, trees require specialized techniques in order to visit their vertices, a topic that is taken up in section 3.6.

3.1 Definitions and examples

> I think that I shall never see
> A poem lovely as a tree.
> — Joyce Kilmer, *Trees and Other Poems*, 1914, "Trees"

Recall that a path in a graph $G = (V, E)$ whose start and end vertices are the same is called a cycle. We say G is *acyclic*, or a *forest*, if it has no cycles. In a forest, a vertex of degree one is called an *endpoint* or a *leaf*. Any vertex that is not a leaf is called an

internal vertex. A connected forest is a *tree*. In other words, a tree is a graph without cycles and each edge is a bridge. A forest can also be considered as a collection of trees.

A *rooted tree* T is a tree with a specified *root* vertex v_0, i.e. exactly one vertex has been specially designated as the root of T. However, if G is a rooted tree with root vertex v_0 having degree one, then by convention we do not call v_0 an endpoint or a leaf. The *depth* $\text{depth}(v)$ of a vertex v in T is its distance from the root. The *height* $\text{height}(T)$ of T is the length of a longest path starting from the root vertex, i.e. the height is the maximum depth among all vertices of T. It follows by definition that $\text{depth}(v) = 0$ if and only if v is the root of T, $\text{height}(T) = 0$ if and only if T is the trivial graph, $\text{depth}(v) \geq 0$ for all $v \in V(T)$, and $\text{height}(T) \leq \text{diam}(T)$.

The Unix, in particular Linux, filesystem hierarchy can be viewed as a tree (see Figure 3.1). As shown in Figure 3.1, the root vertex is designated with the forward slash, which is also referred to as the root directory. Other examples of trees include the organism classification tree in Figure 3.2, the family tree in Figure 3.3, and the expression tree in Figure 3.4.

A *directed tree* is a digraph which would be a tree if the directions on the edges were ignored. A rooted tree can be regarded as a directed tree since we can imagine an edge uv for $u, v \in V$ being directed from u to v if and only if v is further away from v_0 than u is. If uv is an edge in a rooted tree, then we call v a *child* vertex with *parent* u. Directed trees are pervasive in theoretical computer science, as they are useful structures for describing algorithms and relationships between objects in certain datasets.

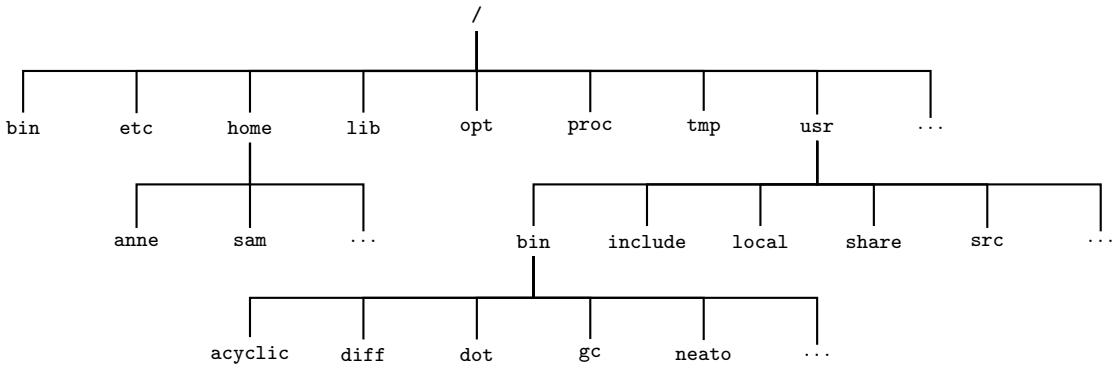


Figure 3.1: The Linux filesystem hierarchy.

An *ordered tree* is a rooted tree for which an ordering is specified for the children of each vertex. An *n-ary tree* is a rooted tree for which each vertex that is not a leaf has at most n children. The case $n = 2$ are called *binary trees*. An n-ary tree is said to be *complete* if each of its internal vertices has exactly n children and all leaves have the same depth. A *spanning tree* of a connected, undirected graph G is a subgraph that is a tree and containing all vertices of G.

Example 3.1. Consider the 4×4 grid graph with 16 vertices and 24 edges. Two examples of a spanning tree are given in Figure 3.5 by using a darker line shading for its edges. ■

Example 3.2. *For $n = 1, \ldots, 6$, how many distinct (nonisomorphic) trees are there of order n? Construct all such trees for each n.*

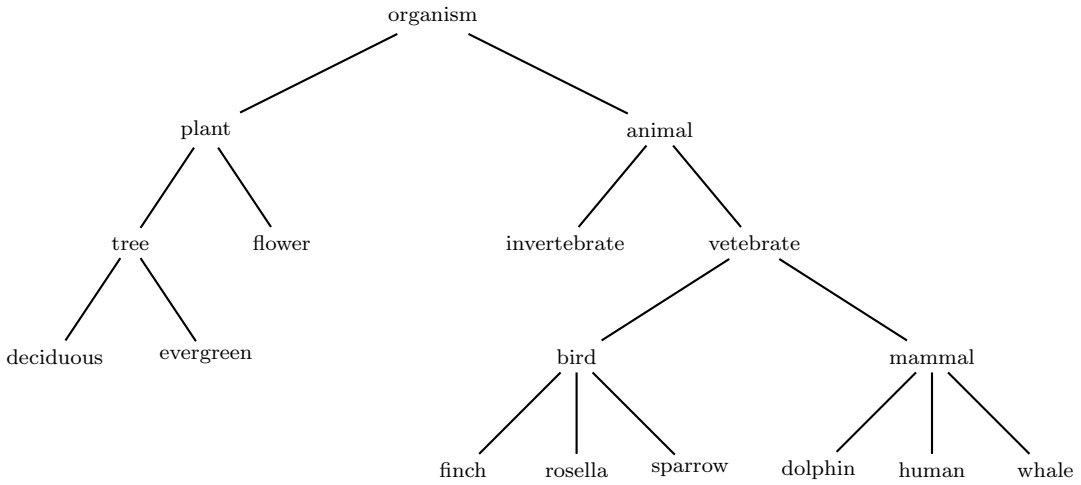


Figure 3.2: Classification tree of organisms.

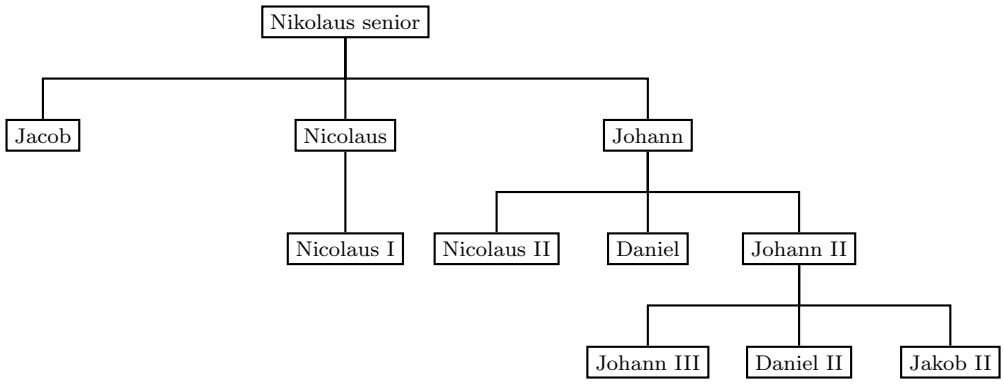


Figure 3.3: Bernoulli family tree of mathematicians.

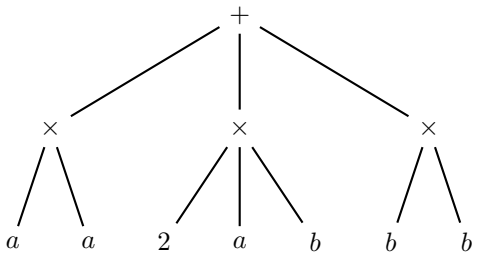


Figure 3.4: Expression tree for the perfect square $a^2 + 2ab + b^2$.

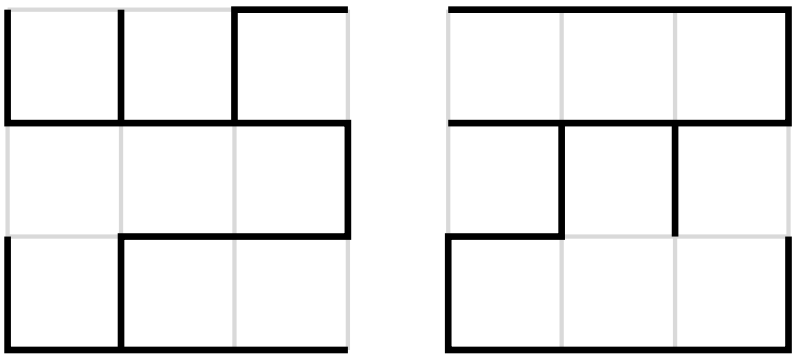


Figure 3.5: Two spanning trees for the 4×4 grid graph.

Solution. For $n = 1$, there is only one tree of order 1, i.e. K_1. The same is true for $n = 2$ and $n = 3$, where the required trees are P_2 and P_3, respectively (see Figure 3.6). We have two trees of order $n = 4$ (see Figure 3.7), three of order $n = 5$ (see Figure 3.8), and six of order $n = 6$ (see Figure 3.9). ■

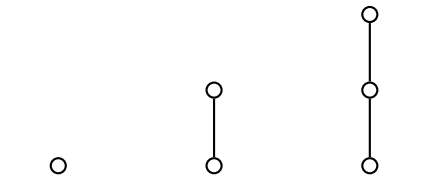


Figure 3.6: All distinct trees of order $n = 1, 2, 3$.

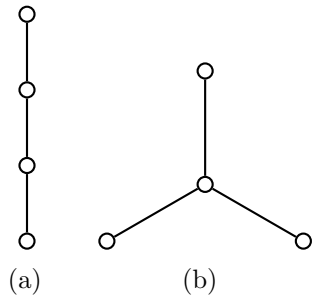


Figure 3.7: All distinct trees of order $n = 4$.

Example 3.3. *Let $T = (V, E)$ be a tree with vertex set*

$$V = \{a, b, c, d, e, f, v, w, x, y, z\}$$

edge set

$$E = \{va,\ vw,\ wx,\ wy,\ xb,\ xc,\ yd,\ yz,\ ze,\ zf\}$$

and root vertex v. Verify that T is a binary tree. Suppose that x is the root of the branch we want to remove from T. Find all children of x and cut off the branch rooted at x from T. Is the resulting graph also a binary tree?

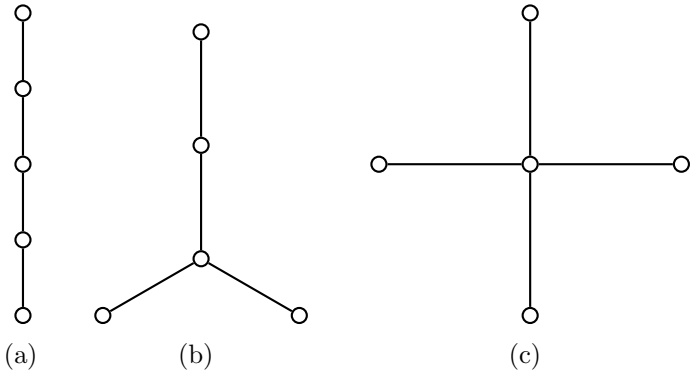


Figure 3.8: All distinct trees of order $n = 5$.

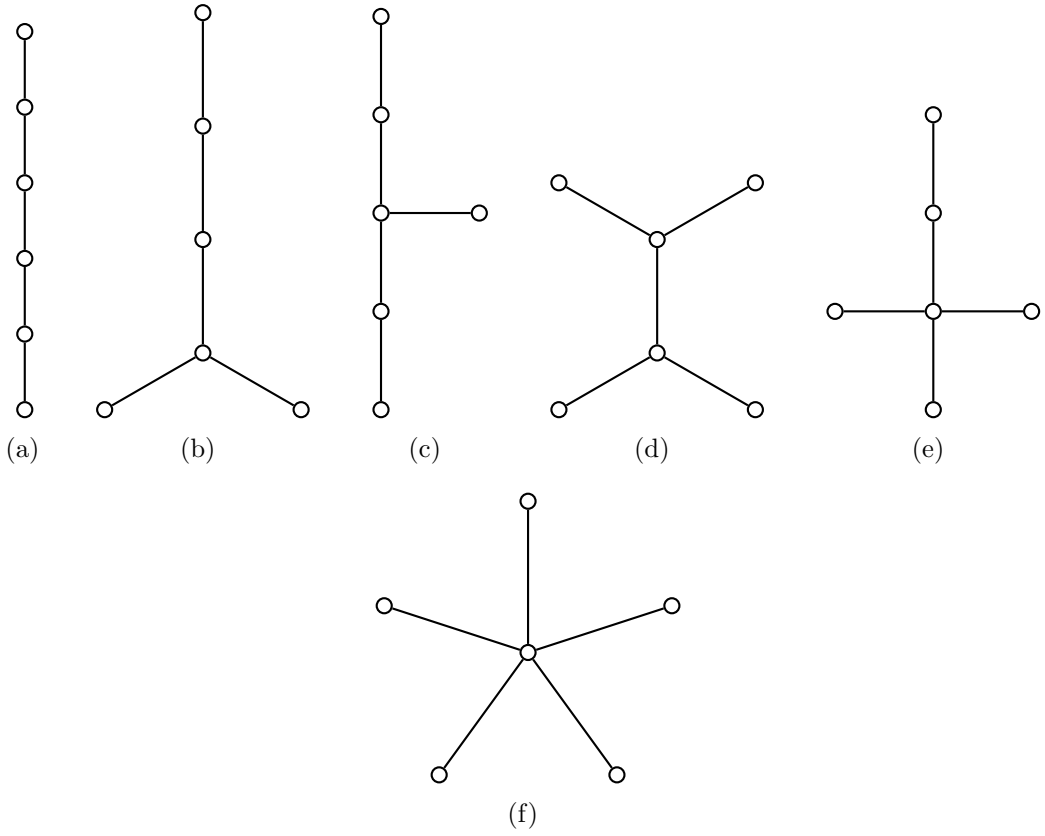


Figure 3.9: All distinct trees of order $n = 6$.

Solution. We construct the tree T in Sage as follows:

```
sage: T = DiGraph({
...    "v":["a","w"], "w":["x","y"],
...    "x":["c","b"], "y":["z","d"],
...    "z":["f","e"]})
sage: for v in T.vertex_iterator():
...       print(v),
a c b e d f w v y x z
sage: for e in T.edge_iterator():
...       print("%s%s" % (e[0], e[1])),
wy wx va vw yd yz xc xb ze zf
```

Each vertex in a binary tree has at most 2 children. Use this definition to test whether or not a graph is a binary tree.

```
sage: T.is_tree()
True
sage: def is_bintree1(G):
...       for v in G.vertex_iterator():
...           if len(G.neighbors_out(v)) > 2:
...               return False
...       return True
sage: is_bintree1(T)
True
```

Here's another way to test for binary trees. Let T be an undirected rooted tree. Each vertex in a binary tree has a maximum degree of 3. If the root vertex is the only vertex with degree 2, then T is a binary tree. (Problem 3.5 asks you to prove this result.) We can use this test because the root vertex v of T is the only vertex with two children.

```
sage: def is_bintree2(G):
...       if G.is_tree() and max(G.degree()) == 3 and G.degree().count(2) == 1:
...           return True
...       return False
sage: is_bintree2(T.to_undirected())
True
```

As x is the root vertex of the branch we want to cut off from T, we could use breadth- or depth-first search to determine all the children of x. We then delete x and its children from T.

```
sage: T2 = copy(T)
sage: # using breadth-first search
sage: V = list(T.breadth_first_search("x")); V
['x', 'c', 'b']
sage: T.delete_vertices(V)
sage: for v in T.vertex_iterator():
...       print(v),
a e d f w v y z
sage: for e in T.edge_iterator():
...       print("%s%s" % (e[0], e[1])),
wy va vw yd yz ze zf
sage: # using depth-first search
sage: V = list(T2.depth_first_search("x")); V
['x', 'b', 'c']
sage: T2.delete_vertices(V)
sage: for v in T2.vertex_iterator():
...       print(v),
a e d f w v y z
sage: for e in T2.edge_iterator():
...       print("%s%s" % (e[0], e[1])),
wy va vw yd yz ze zf
```

The resulting graph T is a binary tree because each vertex has at most two children.

```
sage: T
Digraph on 8 vertices
sage: is_bintree1(T)
True
```

Notice that the test defined in the function `is_bintree2` can no longer be used to test whether or not T is a binary tree, because T now has two vertices, i.e. v and w, each of

which has degree 2. ■

Consider again the organism classification tree in Figure 3.2. We can view the vertex "organism" as the root of the tree and having two children. The first branch of "organism" is the subtree rooted at "plant" and its second branch is the subtree rooted at "animal". We form the complete tree by joining an edge between "organism" and "plant", and an edge between "organism" and "animal". The subtree rooted at "plant" can be constructed in the same manner. The first branch of this subtree is the subtree rooted at "tree" and the second branch is the subtree rooted at "flower". To construct the subtree rooted at "plant", we join an edge between "plant" and "tree", and an edge between "plant" and "flower". The other subtrees of the tree in Figure 3.2 can be constructed using the above recursive procedure.

In general, the recursive construction in Theorem 3.4 provides an alternative way to define trees. We say *construction* because it provides an algorithm to construct a tree, as opposed to the nonconstructive definition presented earlier in this section, where we defined the conditions under which a graph qualifies as a tree without presenting a procedure to construct a tree. Furthermore, we say *recursive* since a larger tree can be viewed as being constructed from smaller trees, i.e. join up existing trees to obtain a new tree. The recursive construction of trees as presented in Theorem 3.4 is illustrated in Figure 3.10.

Theorem 3.4. ***Recursive construction of trees.*** *An isolated vertex is a tree. That single vertex is the root of the tree. Given a collection $T_1, T_2, \ldots, T_n$ of $n > 0$ trees, construct a new tree as follows:*

1. *Let T be a tree having exactly the one vertex v, which is the root of T.*
2. *Let v_i be the root of the tree T_i.*
3. *For $i = 1, 2, \ldots, n$, add the edge vv_i to T and add T_i to T. That is, each v_i is now a child of v.*

The result is the tree T rooted at v with vertex set

$$V(T) = \{v\} \cup \left(\bigcup_i V(T_i) \right)$$

and edge set

$$E(T) = \bigcup_i \big(\{vv_i\} \cup E(T_i) \big).$$

The following game is a variant of the Shannon switching game, due to Edmonds and Lehman. We follow the description in Oxley's survey [**?**]. Recall that a minimal edge cut of a graph is also called a bond of the graph. The following two-person game is played on a connected graph $G = (V, E)$. Two players Alice and Bob alternately tag elements of E. Alice's goal is to tag the edges of a spanning tree, while Bob's goal is to tag the edges of a bond. If we think of this game in terms of a communication network, then Bob's goal is to separate the network into pieces that are no longer connected to each other, while Alice is aiming to reinforce edges of the network to prevent their destruction. Each move for Bob consists of destroying one edge, while each move for Alice involves securing an edge against destruction. The next result characterizes winning strategies on G. The full proof can be found in Oxley [**?**]. See Rasmussen [**?**] for optimization algorithms for solving similar games.

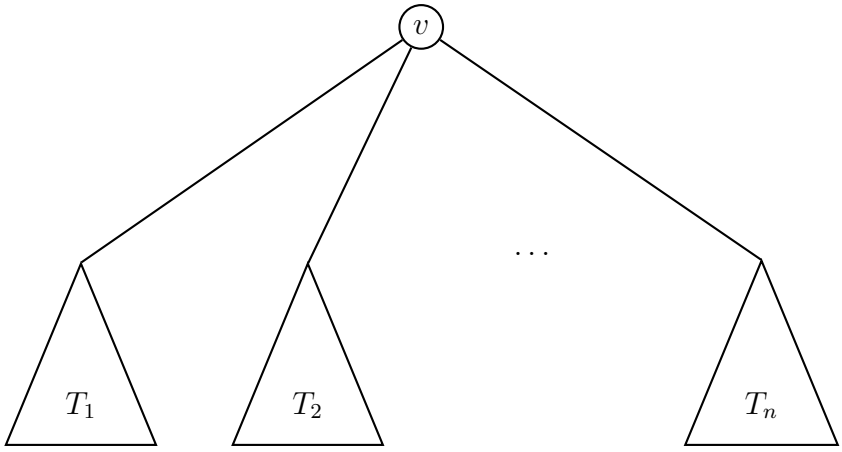


Figure 3.10: Recursive construction of a tree.

Theorem 3.5. *The following statements are equivalent for a connected graph* $G = (V, E)$*.*

1. *Bob plays first and Alice can win against all possible strategies of Bob.*
2. *The graph* G *has* 2 *edge-disjoint spanning trees.*
3. *For all partitions* P *of the vertex set* V*, the number of edges of* G *that join vertices in different classes of the partition is at least* $2(|P| - 1)$*.*

3.2 Properties of trees

> All theory, dear friend, is grey, but the golden tree of actual life springs ever green.
> — Johann Wolfgang von Goethe, *Faust*, part 1, 1808

By Theorem 1.33, each edge of a tree is a bridge. Removing any edge of a tree partitions the tree into two components, each of which is a subtree of the original tree. The following results provide further basic characterizations of trees.

Theorem 3.6. *Any tree* $T = (V, E)$ *has size* $|E| = |V| - 1$*.*

Proof. This follows by induction on the number of vertices. By definition, a tree has no cycles. We need to show that any tree $T = (V, E)$ has size $|E| = |V| - 1$. For the base case $|V| = 1$, there are no edges. Assume for induction that the result holds for all integers less than or equal to $k \geq 2$. Let $T = (V, E)$ be a tree having $k + 1$ vertices. Remove an edge from T, but not the vertices it is incident to. This disconnects T into two components $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, where $|E| = |E_1| + |E_2| + 1$ and $|V| = |V_1| + |V_2|$ (and possibly one of the E_i is empty). Each T_i is a tree satisfying the conditions of the induction hypothesis. Therefore,

$$\begin{aligned} |E| &= |E_1| + |E_2| + 1 \\ &= |V_1| - 1 + |V_2| - 1 + 1 \\ &= |V| - 1. \end{aligned}$$

as required. ∎

Corollary 3.7. *If $T = (V, E)$ is a graph of order $|V| = n$, then the following are equivalent:*

1. *T is a tree.*
2. *T contains no cycles and has $n - 1$ edges.*
3. *T is connected and has $n - 1$ edges.*
4. *Every edge of T is a cut set.*

Proof. (1) $\implies$ (2): This holds by definition of trees and Theorem 3.6.

(2) $\implies$ (3): If $T = (V, E)$ has k connected components then it is a disjoint union of trees $T_i = (V_i, E_i)$, $i = 1, 2, \ldots, k$, for some k. By part (2), each of these satisfy

$$|E_i| = |V_i| - 1$$

so

$$\begin{aligned} |E| &= \sum_{i=1}^{k} |E_i| \\ &= \sum_{i=1}^{k} |V_i| - k \\ &= |V| - k. \end{aligned}$$

This contradicts part (2) unless $k = 1$. Therefore, T is connected.

(3) $\implies$ (4): If removing an edge $e \in E$ leaves $T = (V, E)$ connected then $T' = (V, E')$ is a tree, where $E' = E - e$. However, this means that $|E'| = |E| - 1 = |V| - 1 - 1 = |V| - 2$, which contradicts part (3). Therefore e is a cut set.

(4) $\implies$ (1): From part (2) we know that T has no cycles and from part (3) we know that T is connected. Conclude by the definition of trees that T is a tree. ∎

Theorem 3.8. *Let $T = (V, E)$ be a tree and let $u, v \in V$ be distinct vertices. Then T has exactly one u-v path.*

Proof. Suppose for contradiction that

$$P : v_0 = u,\ v_1,\ v_2, \ldots, v_k = v$$

and

$$Q : w_0 = u,\ w_1,\ w_2, \ldots, w_\ell = v$$

are two distinct u-v paths. Then P and Q has a common vertex x, which is possibly $x = u$. For some $i \geq 0$ and some $j \geq 0$ we have $v_i = x = w_j$, but $v_{i+1} \neq w_{j+1}$. Let y be the first vertex after x such that y belongs to both P and Q. (It is possible that $y = v$.) We now have two distinct x-y paths that have only x and y in common. Taken together, these two x-y paths result in a cycle, contradicting our hypothesis that T is a tree. Therefore T has only one u-v path. ∎

Theorem 3.9. *If $T = (V, E)$ is a graph then the following are equivalent:*

1. *T is a tree.*

2. *For any new edge e, the join $T + e$ has exactly one cycle.*

Proof. (1) $\implies$ (2): Let $e = uv$ be a new edge connecting $u, v \in V$. Suppose that

$$P : v_0 = w,\ v_1,\ v_2, \ldots, v_k = w$$

and

$$P' : v'_0 = w,\ v'_1,\ v'_2, \ldots, v'_\ell = w$$

are two cycles in $T + e$. If either P or P' does not contain e, say P does not contain e, then P is a cycle in T. Let $u = v_0$ and let $v = v_1$. The edge $(v_0 = w,\ v_1)$ is a u-v path and the sequence $v = v_1,\ v_2, \ldots, v_k = w = u$ taken in reverse order is another u-v path. This contradicts Theorem 3.8.

We may now suppose that P and P' both contain e. Then P contains a subpath $P_0 = P - e$ (which is not closed) that is the same as P except it lacks the edge from u to v. Likewise, P' contains a subpath $P'_0 = P' - e$ (which is not closed) that is the same as P' except it lacks the edge from u to v. By Theorem 3.8, these u-v paths P_0 and P'_0 must be the same. This forces P and P' to be the same, which proves part (2).

(2) $\implies$ (1): Part (2) implies that T is acyclic. (Otherwise, it is trivial to make two cycles by adding an extra edge.) We must show T is connected. Suppose T is disconnected. Let u be a vertex in one component, T_1 say, of T and v a vertex in another component, T_2 say, of T. Adding the edge $e = uv$ does not create a cycle (if it did then T_1 and T_2 would not be disjoint), which contradicts part (2). ■

Taking together the results in this section, we have the following characterizations of trees.

Theorem 3.10. ***Basic characterizations of trees.*** *If $T = (V, E)$ is a graph with n vertices, then the following statements are equivalent:*

1. *T is a tree.*

2. *T contains no cycles and has $n - 1$ edges.*

3. *T is connected and has $n - 1$ edges.*

4. *Every edge of T is a cut set.*

5. *For any pair of distinct vertices $u, v \in V$, there is exactly one u-v path.*

6. *For any new edge e, the join $T + e$ has exactly one cycle.*

Let $G = (V_1, E_1)$ be a graph and $T = (V_2, E_2)$ a subgraph of G that is a tree. As in part (6) of Theorem 3.10, we see that adding just one edge in $E_1 - E_2$ to T will create a unique cycle in G. Such a cycle is called a *fundamental cycle* of G. The set of such fundamental cycles of G depends on T.

The following result essentially says that if a tree has at least one edge, then the tree has at least two vertices each of which has degree one. In other words, each tree of order ≥ 2 has at least two pendants.

Theorem 3.11. *Every nontrivial tree has at least two leaves.*

Proof. Let T be a nontrivial tree of order m and size n. Consider the degree sequence $d_1, d_2, \ldots, d_m$ of T where $d_1 \leq d_2 \leq \cdots \leq d_m$. As T is nontrivial and connected, then $m \geq 2$ and $d_i \geq 1$ for $i = 1, 2, \ldots, m$. If T has less than two leaves, then $d_1 \geq 1$ and $d_i \geq 2$ for $2 \leq i \leq m$, hence

$$\sum_{i=1}^{m} d_i \geq 1 + 2(m-1) = 2m - 1. \tag{3.1}$$

But by Theorems 1.9 and 3.6, we have

$$\sum_{i=1}^{m} d_i = 2n = 2(m-1) = 2m - 2$$

which contradicts inequality (3.1). Conclude that T has at least two leaves. ■

Theorem 3.12. *If T is a tree of order m and G is a graph with minimum degree $\delta(G) \geq m - 1$, then T is isomorphic to a subgraph of G.*

Proof. Use an inductive argument on the number of vertices. The result holds for $m = 1$ because K_1 is a subgraph of every nontrivial graph. The result also holds for $m = 2$ since K_2 is a subgraph of any graph with at least one edge.

Let $m \geq 3$, let T_1 be a tree of order $m-1$, and let H be a graph with $\delta(H) \geq m-2$. Assume for induction that T_1 is isomorphic to a subgraph of H. We need to show that if T is a tree of order m and G is a graph with $\delta(G) \geq m-1$, then T is isomorphic to a subgraph of G. Towards that end, consider a leaf v of T and let u be a vertex of T such that u is adjacent to v. Then $T - v$ is a tree of order $m-1$ and $\delta(G) \geq m-1 > m-2$. Apply the inductive hypothesis to see that $T - v$ is isomorphic to a subgraph T' of G. Let u' be the vertex of T' that corresponds to the vertex u of T under an isomorphism. Since $\deg(u') \geq m-1$ and T' has $m-2$ vertices distinct from u', it follows that u' is adjacent to some $w \in V(G)$ such that $w \notin V(T')$. Therefore T is isomorphic to the graph obtained by adding the edge $u'w$ to T'. ■

Example 3.13. *Consider a positive integer n. The Euler phi function $\varphi(n)$ counts the number of integers a, with $1 \leq a \leq n$, such that $\gcd(a, n) = 1$. The Euler phi sequence of n is obtained by repeatedly iterating $\varphi(n)$ with initial iteration value n. Continue on iterating and stop when the output of $\varphi(\alpha_k)$ is 1, for some positive integer α_k. The number of terms generated by the iteration, including the initial iteration value n and the final value of 1, is the length of $\varphi(n)$.*

(a) Let $s_0 = n, s_1, s_2, \ldots, s_k = 1$ be the Euler phi sequence of n and produce a digraph G of this sequence as follows. The vertex set of G is $V = \{s_0 = n, s_1, s_2, \ldots, s_k = 1\}$ and the edge set of G is $E = \{s_i s_{i+1} \mid 0 \leq i < k\}$. Produce the digraphs of the Euler phi sequences of 15, 22, 33, 35, 69, and 72. Construct the union of all such digraphs and describe the resulting graph structure.

(b) For each $n = 1, 2, \ldots, 1000$, compute the length of $\varphi(n)$ and plot the pairs $(n, \varphi(n))$ on one set of axes.

Solution. The Euler phi sequence of 15 is

$$15, \quad \varphi(15) = 8, \quad \varphi(8) = 4, \quad \varphi(4) = 2, \quad \varphi(2) = 1.$$

The Euler phi sequences of 22, 33, 35, 69, and 72 can be similarly computed to obtain their respective digraph representations. The union of all such digraphs is a directed tree rooted at 1, as shown in Figure 3.11(a). Figure 3.11(b) shows a scatterplot of n versus the length of $\varphi(n)$. ■

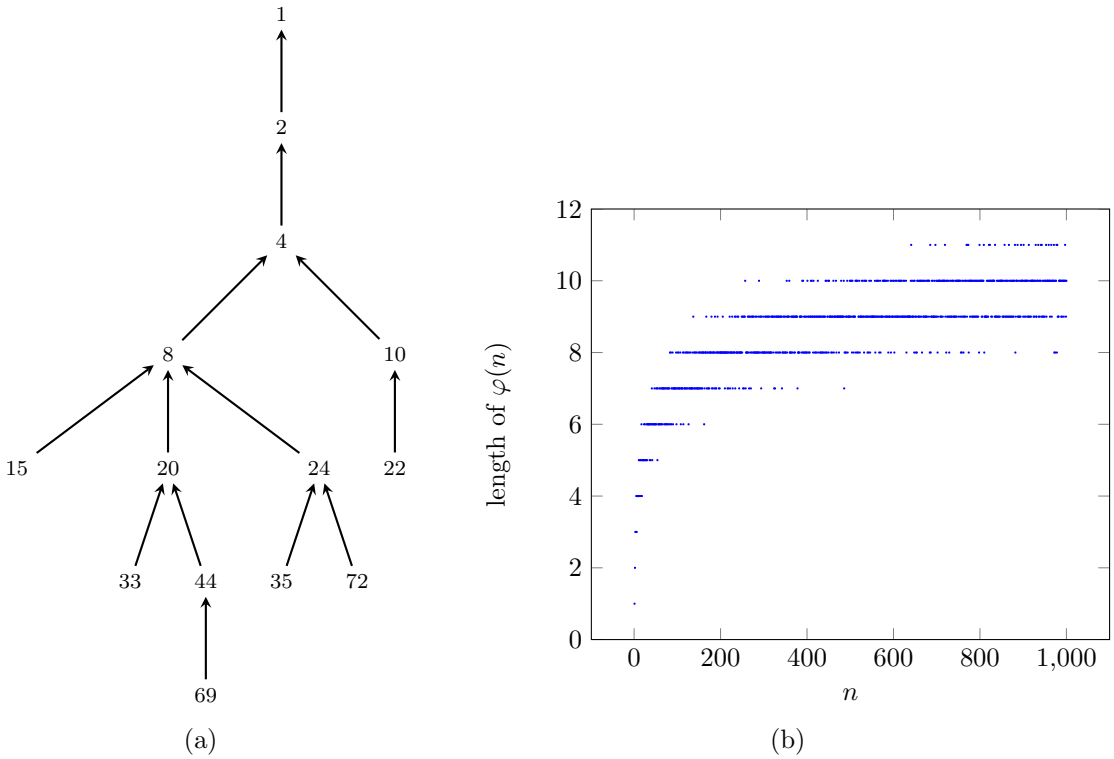


Figure 3.11: Union of digraphs of Euler phi sequences and scatterplot.

3.3 Minimum spanning trees

Suppose we want to design an electronic circuit connecting several components. If these components represent the vertices of a graph and a wire connecting two components represents an edge of the graph, then for economical reasons we will want to connect the components together using the least amount of wire. The problem essentially amounts to finding a minimum spanning tree in the graph containing these vertices.

But what is a spanning tree? We can characterize a spanning tree in several ways, each leading to an algorithm for constructing a spanning tree. Let G be a connected graph and let T be a subgraph of G. If T is a tree that contains all the vertices of G, then T is called a *spanning tree* of G. We can think of T as a tree that is also an edge-deletion subgraph of G. That is, we start with a connected graph G and delete an edge from G such that the resulting edge-deletion subgraph T_1 is still connected. If T_1 is a tree, then we have obtained a spanning tree of G. Otherwise, we delete an edge from T_1 to obtain an edge-deletion subgraph T_2 that is still connected. If T_2 is a tree, then we are done. Otherwise, we repeat the above procedure until we obtain an edge-deletion subgraph T_k of G such that T_k is connected, T_k is a tree, and it contains all vertices of G. Each edge removal does not decrease the number of vertices and must also leave

the resulting edge-deletion subgraph connected. Thus eventually the above procedure results in a spanning tree of G. Our discussion is summarized in Algorithm 3.1.

Algorithm 3.1: Randomized spanning tree construction.

Input: A connected graph G.
Output: A spanning tree of G.

```
1 T ← G
2 while T is not a tree do
3     e ← random edge of T
4     if T − e is connected then
5         T ← T − e
6 return T
```

Another characterization of a spanning tree T of a connected graph G is that T is a maximal set of edges of G that contains no cycle. Kruskal's algorithm (see section 3.3.1) exploits this condition to construct a minimum spanning tree (MST). A *minimum spanning tree* is a spanning tree of a weighted graph having lowest total weight among all possible spanning trees of the graph. A third characterization of a spanning tree is that it is a minimal set of edges that connect all vertices, a characterization that results in yet another algorithm called Prim's algorithm (see section 3.3.2) for constructing minimum spanning trees. The task of determining a minimum spanning tree in a connected weighted graph is called the *minimum spanning tree problem*. As early as 1926, Otakar Borůvka stated [**?**, **?**] this problem and offered a solution now known as Borůvka's algorithm (see section 3.3.3). See [**?**, **?**] for a history of the minimum spanning tree problem.

3.3.1 Kruskal's algorithm

In 1956, Joseph B. Kruskal published [**?**] a procedure for constructing a minimum spanning tree of a connected weighted graph $G = (V, E)$. Now known as Kruskal's algorithm, with a suitable implementation the procedure runs in $O\big(|E| \cdot \log |E|\big)$ time. Variants of Kruskal's algorithm include the algorithm by Prim [**?**] and that by Loberman and Weinberger [**?**].

Kruskal's algorithm belongs to the class of greedy algorithms. As will be explained below, when constructing a minimum spanning tree Kruskal's algorithm considers only the edge having minimum weight among all available edges. Given a weighted nontrivial graph $G = (V, E)$ that is connected, let $w : E \to \mathbf{R}$ be the weight function of G. The first stage is creating a "skeleton" of the tree T that is initially set to be a graph without edges, i.e. $T = (V, \emptyset)$. The next stage involves sorting the edges of G by weights in nondecreasing order. In other words, we label the edges of G as follows:

$$E = \{e_1, e_2, \ldots, e_n\}$$

where $n = |E|$ and $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_n)$. Now consider each edge e_i for $i = 1, 2, \ldots, n$. We add e_i to the edge set of T provided that e_i does not result in T having a cycle. The only way adding $e_i = u_i v_i$ to T would create a cycle is if both u_i and v_i were endpoints of edges (not necessarily distinct) in the same connected component of T. As long as the acyclic condition holds with the addition of a new edge to T, we add that new edge. Following the acyclic test, we also test that the (updated) graph

T is a tree of G. As G is a graph of order $|V|$, apply Theorem 3.10 to see that if T has size $|V| - 1$, then it is a spanning tree of G. Algorithm 3.2 provides pseudocode of our discussion of Kruskal's algorithm. When the algorithm halts, it returns a minimum spanning tree of G. The correctness of Algorithm 3.2 is proven in Theorem 3.14.

Algorithm 3.2: Kruskal's algorithm.

Input: A connected weighted graph $G = (V, E)$ with weight function w.
Output: A minimum spanning tree of G.

```
1 m ← |V|
2 T ← ∅
3 sort E = {e_1, e_2, ..., e_n} by weights so that w(e_1) ≤ w(w_2) ≤ ··· ≤ w(e_n)
4 for i ← 1, 2, ..., n do
5     if e_i ∉ E(T) and T ∪ {e_i} is acyclic then
6         T ← T ∪ {e_i}
7     if |T| = m − 1 then
8         return T
```

Theorem 3.14. ***Correctness of Algorithm 3.2.*** *If G is a nontrivial connected weighted graph, then Algorithm 3.2 outputs a minimum spanning tree of G.*

Proof. Let G be a nontrivial connected graph of order m and having weight function w. Let T be a subgraph of G produced by Kruskal's algorithm 3.2. By construction, T is a spanning tree of G with

$$E(T) = \{e_1, e_2, \ldots, e_{m-1}\}$$

where $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_{m-1})$ so that the total weight of T is

$$w(T) = \sum_{i=1}^{m-1} w(e_i).$$

Suppose for contradiction that T is not a minimum spanning tree of G. Among all the minimum spanning trees of G, let H be a minimum spanning tree of G such that H has the most number of edges in common with T. As T and H are distinct subgraphs of G, then T has at least an edge not belonging to H. Let $e_i \in E(T)$ be the first edge not in H. Construct the graph $G_0 = H + e_i$ obtained by adding the edge e_i to H. Note that G_0 has exactly one cycle C. Since T is acyclic, there exists an edge $e_0 \in E(C)$ such that e_0 is not in T. Construct the graph $T_0 = G_0 - e_0$ obtained by deleting the edge e_0 from G_0. Then T_0 is a spanning tree of G with

$$w(T_0) = w(H) + w(e_i) - w(e_0)$$

and $w(H) \leq w(T_0)$ and hence $w(e_0) \leq w(e_i)$. By Kruskal's algorithm 3.2, e_i is an edge of minimum weight such that $\{e_1, e_2, \ldots, e_{i-1}\} \cup \{e_i\}$ is acyclic. Furthermore, the subgraph $\{e_1, e_2, \ldots, e_{i-1}, e_0\}$ of H is acyclic. Thus we have $w(e_i) = w(e_0)$ and $w(T_0) = w(H)$ and so T is a minimum spanning tree of G. By construction, T_0 has more edges in common with T than H has with T, in contradiction of our hypothesis. ■

```
def kruskal(G):
    """
    Implements Kruskal's algorithm to compute a MST of a graph.

    INPUT:
        G - a connected edge-weighted graph or digraph
                whose vertices are assumed to be 0, 1, ...., n-1.
    OUTPUT:
        T - a minimum weight spanning tree.

    If G is not explicitly edge-weighted then the algorithm
    assumes all edge weights are 1. The tree T returned is
    a weighted graph, even if G is not.

    EXAMPLES:
        sage: A = matrix([[0,1,2,3],[0,0,2,1],[0,0,0,3],[0,0,0,0]])
        sage: G = DiGraph(A, format = "adjacency_matrix", weighted = True)
        sage: TE = kruskal(G); TE.edges()
        [(0, 1, 1), (0, 2, 2), (1, 3, 1)]
        sage: G.edges()
        [(0, 1, 1), (0, 2, 2), (0, 3, 3), (1, 2, 2), (1, 3, 1), (2, 3, 3)]
        sage: G = graphs.PetersenGraph()
        sage: TE = kruskal(G); TE.edges()
        [(0, 1, 1), (0, 4, 1), (0, 5, 1), (1, 2, 1), (1, 6, 1), (2, 3, 1),
         (2, 7, 1), (3, 8, 1), (4, 9, 1)]

    TODO:
        Add ''verbose'' option to make steps more transparent.
       (Useful for teachers and students.)
    """
    T_vertices = G.vertices() # a list of the form range(n)
    T_edges = []
    E = G.edges() # a list of triples
    # start ugly hack
    Er = [list(x) for x in E]
    E0 = []
    for x in Er:
        x.reverse()
        E0.append(x)
    E0.sort()
    E = []
    for x in E0:
        x.reverse()
        E.append(tuple(x))
    # end ugly hack to get E is sorted by weight
    for x in E:  # find edges of T
        TV = flatten(T_edges)
        u = x[0]
        v = x[1]
        if not(u in TV and v in TV):
            T_edges.append([u,v])
    # find adj mat of T
    if G.weighted():
        AG = G.weighted_adjacency_matrix()
    else:
        AG = G.adjacency_matrix()
    GV = G.vertices()
    n = len(GV)
    AT = []
    for i in GV:
        rw = [0]*n
        for j in GV:
            if [i,j] in T_edges:
                rw[j] = AG[i][j]
        AT.append(rw)
    AT = matrix(AT)
    return Graph(AT, format = "adjacency_matrix", weighted = True)
```

Here is an example. We start with the grid graph. This is implemented in Sage such that the vertices are given by the coordinates of the grid the graph lies on, as opposed to $0, 1, \ldots, n-1$. Since the above implementation of Kruskal's algorithm assumes that the vertices are $V = \{0, 1, \ldots, n-1\}$, we first redefine the graph suitable for running Kruskal's algorithm on it.

```
sage: G = graphs.GridGraph([4,4])
```

```
sage: A = G.adjacency_matrix()
sage: G = Graph(A, format="adjacency_matrix", weighted=True)
sage: T = kruskal(G); T.edges()
[(0, 1, 1), (0, 4, 1), (1, 2, 1), (1, 5, 1), (2, 3, 1), (2, 6, 1), (3,7, 1),
 (4, 8, 1), (5, 9, 1), (6, 10, 1), (7, 11, 1), (8, 12, 1), (9, 13, 1),
 (10, 14, 1), (11, 15, 1)]
```

An illustration of this graph is given in Figure 3.12.

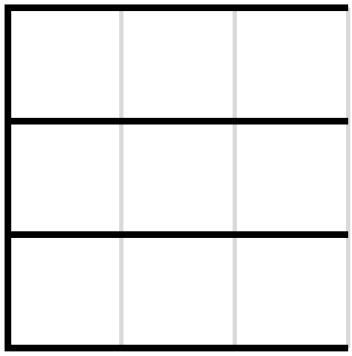


Figure 3.12: Kruskal's algorithm for the 4×4 grid graph.

3.3.2 Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm uses a greedy approach to computing a minimum spanning tree of a connected weighted graph $G = (V, E)$, where $n = |V|$ and $m = |E|$. The algorithm was developed in 1930 by Czech mathematician V. Jarník [**?**] and later independently by R. C. Prim [**?**] and E. W. Dijkstra [**?**]. However, Prim was the first to present an implementation that runs in time $O(n^2)$. Using 2-heaps, the runtime can be reduced [**?**] to $O(m \log n)$. With a Fibonacci heap implementation [**?**], the runtime can be reduced even further to $O(m + n \log n)$.

Pseudocode of Prim's algorithm is given in Algorithm 3.3. For each $v \in V$, cost[v] denotes the minimum weight among all edges connecting v to a vertex in the tree T, and parent[v] denotes the parent of v in T. During the algorithm's execution, vertices v that are not in T are organized in the minimum-priority queue Q, prioritized according to cost[v]. Lines 1 to 3 set each cost[v] to a number that is larger than any weight in the graph G, usually written ∞. The parent of each vertex is set to `NULL` because we have not yet started constructing the MST T. In lines 4 to 6, we choose an arbitrary vertex r from V and mark that vertex as the root of T. The minimum-priority queue is set to be all vertices from V. We set cost[r] to zero, making r the only vertex so far with a cost that is $< \infty$. During the first execution of the while loop from lines 7 to 12, r is the first vertex to be extracted from Q and processed. Line 8 extracts a vertex u from Q based on the key cost, thus moving u to the vertex set of T. Line 9 considers all vertices adjacent to u. In an undirected graph, these are simply the neighbors of u. (In a digraph, one could try to replace adj(u) with the out-neighbors oadj(u). Unfortunately, in the digraph case the Prim algorithm in general fails to find a minimum spanning tree with the same orientation as the original digraph.) The while loop updates the cost and parent fields of each vertex v adjacent to u that is not in T. If parent[v] $\neq$ `NULL`, then cost[v] $< \infty$ and cost[v] is the weight of an edge connecting v to some vertex already in T. Lines 13 to 14 construct the edge set of the minimum spanning tree and return this edge set. The proof of correctness of Algorithm 3.3 is similar to the proof of Theorem 3.14.

Figure 3.13 shows the minimum spanning tree rooted at vertex 1 as a result of running Prim's algorithm over a digraph; Figure 3.14 shows the corresponding tree rooted at vertex 5 of an undirected graph.

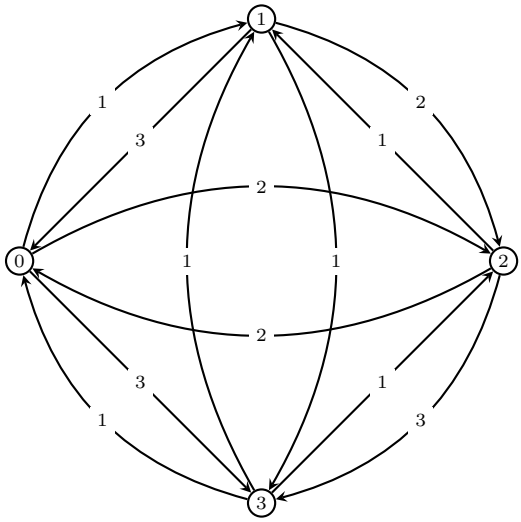
Algorithm 3.3: Prim's algorithm.

Input: A weighted connected graph $G = (V, E)$ with weight function w.
Output: A minimum spanning tree T of G.

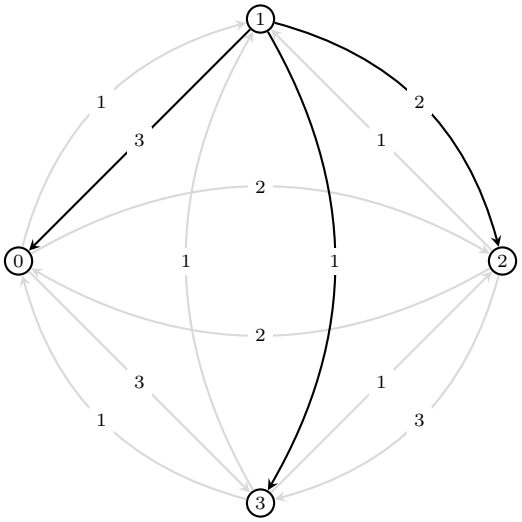
```
1  for each v ∈ V do
2      cost[v] ← ∞
3      parent[v] ← NULL
4  r ← arbitrary vertex of V
5  cost[r] ← 0
6  Q ← V
7  while Q ≠ ∅ do
8      u ← extractMin(Q)
9      for each v ∈ adj(u) do
10         if v ∈ Q and w(u, v) < cost[v] then
11             parent[v] ← u
12             cost[v] ← w(u, v)
13 T ← {(v, parent[v]) | v ∈ V − {r}}
14 return T
```

```
def prim(G):
    """
    Implements Prim's algorithm to compute a MST of a graph.

    INPUT:
        G - a connected graph.
    OUTPUT:
        T - a minimum weight spanning tree.

    REFERENCES:
        http://en.wikipedia.org/wiki/Prim's_algorithm
    """
    T_vertices = [0] # assumes G.vertices = range(n)
    T_edges = []
    E = G.edges() # a list of triples
    V = G.vertices()
    # start ugly hack to sort E
    Er = [list(x) for x in E]
    E0 = []
    for x in Er:
        x.reverse()
        E0.append(x)
    E0.sort()
    E = []
    for x in E0:
        x.reverse()
        E.append(tuple(x))
    # end ugly hack to get E is sorted by weight
    for x in E:
        u = x[0]
        v = x[1]
        if u in T_vertices and not(v in T_vertices):
            T_edges.append([u,v])
            T_vertices.append(v)
    # found T_vertices, T_edges
    # find adj mat of T
    if G.weighted():
        AG = G.weighted_adjacency_matrix()
    else:
        AG = G.adjacency_matrix()
```

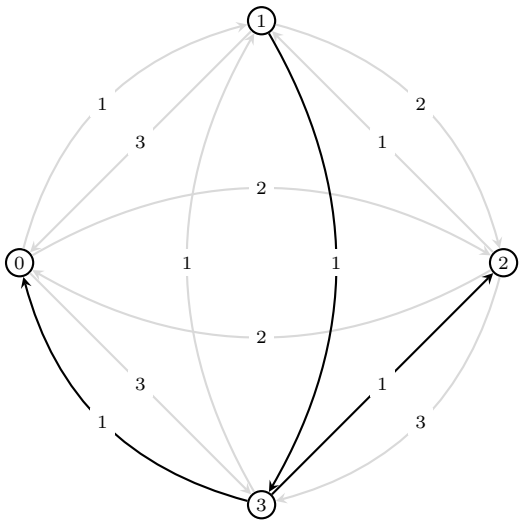
```
    GV = G.vertices()
    n = len(GV)
    AT = []
    for i in GV:
        rw = [0]*n
        for j in GV:
            if [i,j] in T_edges:
                rw[j] = AG[i][j]
        AT.append(rw)
    AT = matrix(AT)
    return Graph(AT, format = "adjacency_matrix", weighted = True)

sage: A = matrix([[0,1,2,3], [3,0,2,1], [2,1,0,3], [1,1,1,0]])
sage: G = DiGraph(A, format="adjacency_matrix", weighted=True)
sage: E = G.edges(); E
[(0, 1, 1), (0, 2, 2), (0, 3, 3), (1, 0, 3), (1, 2, 2), (1, 3, 1), (2, 0, 2),
(2, 1, 1), (2, 3, 3), (3, 0, 1), (3, 1, 1), (3, 2, 1)]
sage: prim(G)
Multi-graph on 4 vertices
sage: prim(G).edges()
[(0, 1, 1), (0, 2, 2), (1, 3, 1)]
```



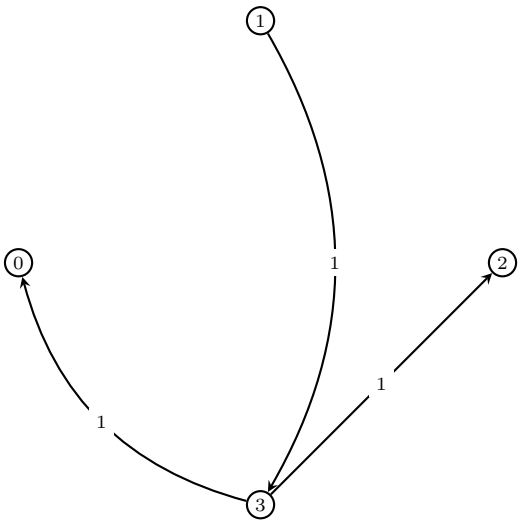
(a) Original digraph.



(b) 1st iteration of while loop.



(c) 2nd iteration of while loop.



(d) Final MST.

Figure 3.13: Running Prim's algorithm over a digraph.

```
sage: A = matrix([[0,7,0,5,0,0,0], [0,0,8,9,7,0,0], [0,0,0,0,5,0,0], \
```

```
...   [0,0,0,0,15,6,0], [0,0,0,0,0,8,9], [0,0,0,0,0,0,11], [0,0,0,0,0,0,0]])
sage: G = Graph(A, format="adjacency_matrix", weighted=True)
sage: E = G.edges(); E
[(0, 1, 7), (0, 3, 5), (1, 2, 8), (1, 3, 9), (1, 4, 7), (2, 4, 5),
(3, 4, 15), (3, 5, 6), (4, 5, 8), (4, 6, 9), (5, 6, 11)]
sage: prim(G).edges()
[(0, 1, 7), (0, 3, 5), (1, 2, 8), (1, 4, 7), (3, 5, 6), (4, 6, 9)]
```

3.3.3 Borůvka's algorithm

Borůvka's algorithm [**?**, **?**] is a procedure for finding a minimum spanning tree in a weighted connected graph $G = (V, E)$ for which all edge weights are distinct. It was first published in 1926 by Otakar Borůvka but subsequently rediscovered by many others, including Choquet [**?**] and Florek et al. [**?**]. If G has order $n = |V|$ and size $m = |E|$, it can be shown that Borůvka's algorithm runs in time $O(m \log n)$.

Algorithm 3.4: Borůvka's algorithm.

Input: A weighted connected graph $G = (V, E)$ with weight function w. All the edge weights of G are distinct.

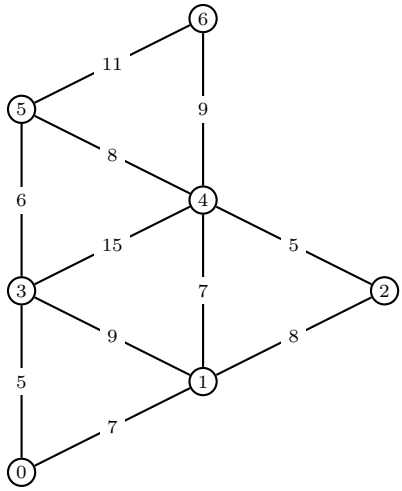
Output: A minimum spanning tree T of G.

```
1 n ← |V|
2 T ← K̄_n
3 while |E(T)| < n − 1 do
4     for each component T′ of T do
5         e′ ← edge of minimum weight that leaves T′
6         E(T) ← E(T) ∪ e′
7 return T
```

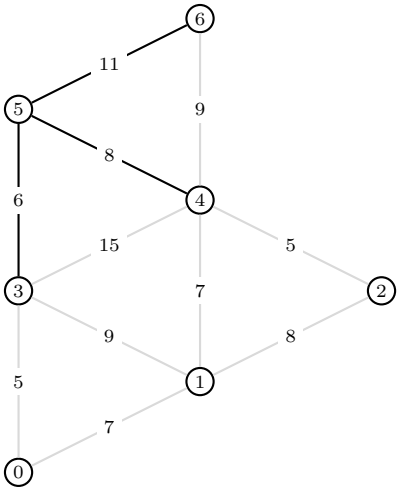
Algorithm 3.4 provides pseudocode of Borůvka's algorithm. Given a weighted connected graph $G = (V, E)$ all of whose edge weights are distinct, the initialization steps in lines 1 and 2 construct a spanning forest T of G, i.e. the subgraph of G containing all of the latter's vertices and no edges. The initial forest has n components, each being the trivial graph K_1. The while loop from lines 3 to 6 constructs a spanning tree of G via a recursive procedure similar to Theorem 3.4. For each component T' of T, we consider all the out-going edges of T' and choose an edge e' that has minimum weight among all such edges. This edge is then added to the edge set of T. In this way, two distinct components, each of which is a tree, are joined together by a bridge. At the end of the while loop, our final graph is a minimum spanning tree of G. Note that the forest-merging steps in the for loop from lines 4 to 6 are amenable to parallelization, hence the alternative name to Borůvka's algorithm: the parallel forest-merging method.

Example 3.15. Figure 3.15 illustrates the gradual construction of a minimum spanning tree for the undirected graph given in Figure 3.15(a). In this case, we require two iterations of the while loop in Borůvka's algorithm in order to obtain the final minimum spanning tree in Figure 3.15(d). ■

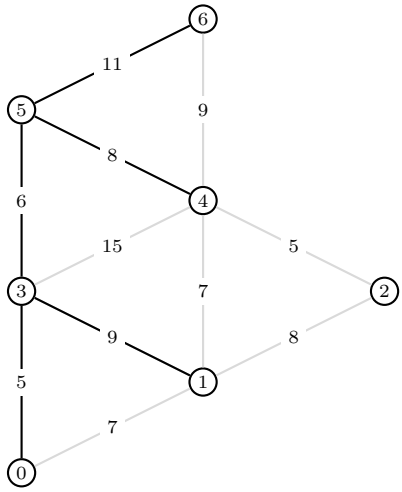
```
def which_index(x,L):
    """
    L is a list of sublists (or tuple of sets or list
    of tuples, etc).
```



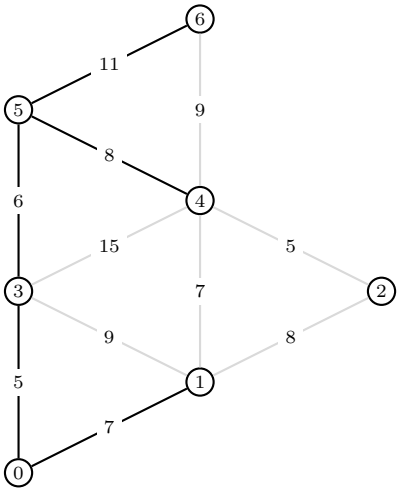
(a) Original undirected graph.



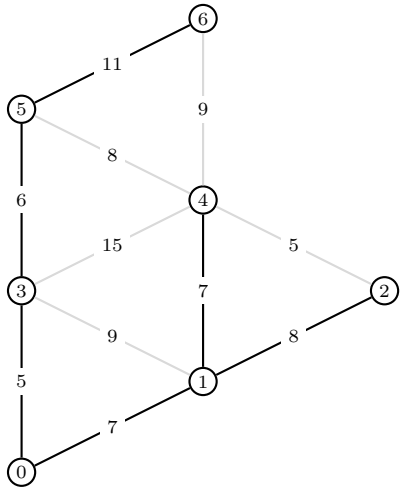
(b) 1st iteration of while loop.



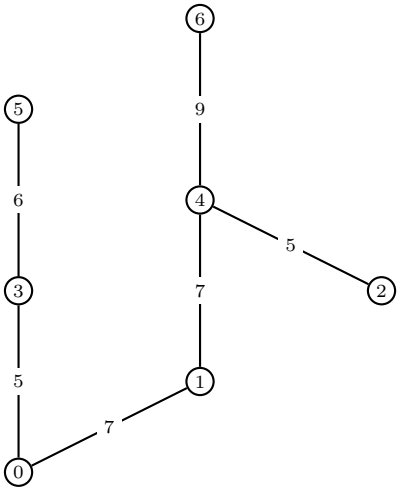
(c) 2nd iteration of while loop.



(d) 3rd iteration of while loop.

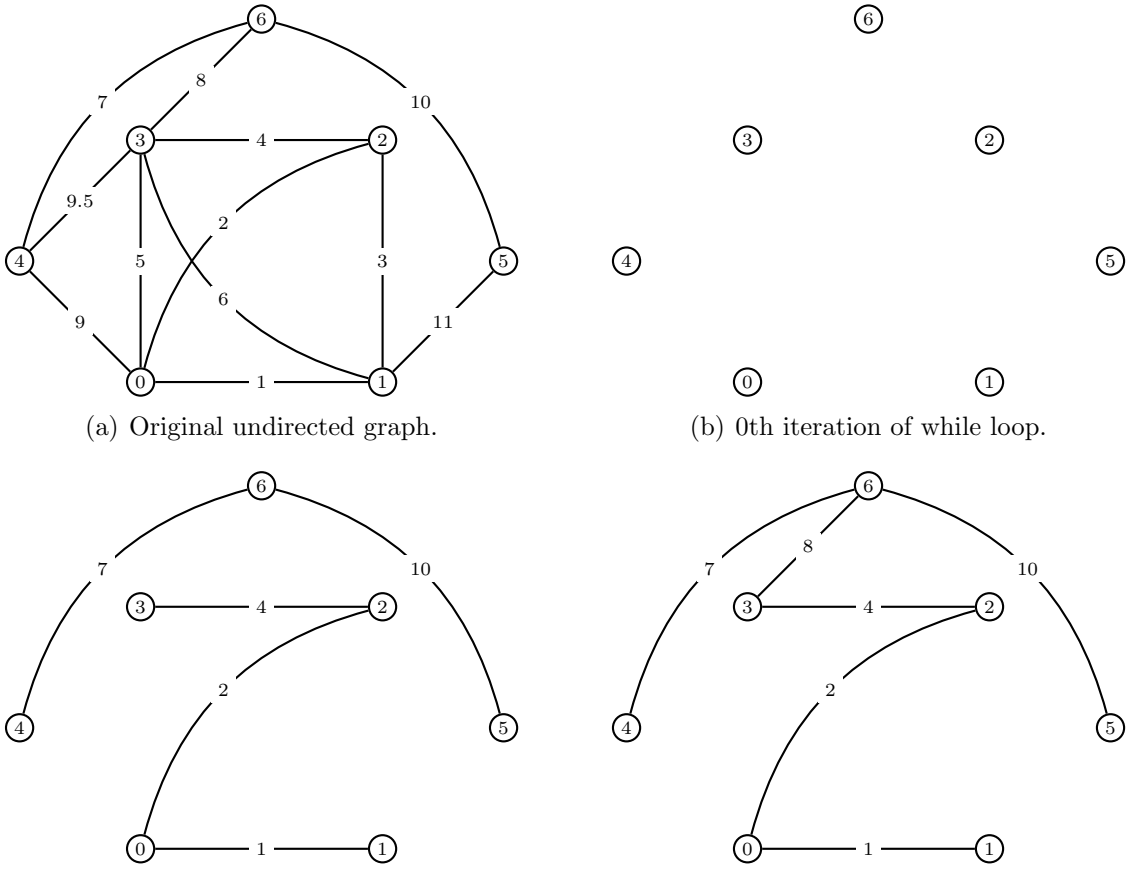


(e) 4th iteration of while loop.



(f) Final MST.

Figure 3.14: Running Prim's algorithm over an undirected graph.



(a) Original undirected graph.

(b) 0th iteration of while loop.

(c) 1st iteration of while loop.

(d) 2nd iteration of while loop.

Figure 3.15: Recursive construction of MST via Borůvka's algorithm.

```
    Returns the index of the first sublist which x belongs
    to, or None if x is not in flatten(L).

    The 0-th element in
    Lx = [L.index(S) for S in L if x in S]
    almost works, but if the list is empty then Lx[0]
    throws an exception.

    EXAMPLES:
        sage: L = [[1,2,3],[4,5],[6,7,8]]
        sage: which_index(3,L)
        0
        sage: which_index(4,L)
        1
        sage: which_index(7,L)
        2
        sage: which_index(9,L)
        sage: which_index(9,L) == None
        True
    """
    for S in L:
        if x in S:
            return L.index(S)
    return None

def boruvka(G):
    """
    Implements Boruvka's algorithm to compute a MST of a graph.

    INPUT:
        G - a connected edge-weighted graph with distinct weights.
    OUTPUT:
        T - a minimum weight spanning tree.

    REFERENCES:
        http://en.wikipedia.org/wiki/Boruvka's_algorithm
    """
    T_vertices = [] # assumes G.vertices = range(n)
    T_edges = []
    T = Graph()
    E = G.edges() # a list of triples
    V = G.vertices()
    # start ugly hack to sort E
    Er = [list(x) for x in E]
    E0 = []
    for x in Er:
        x.reverse()
        E0.append(x)
    E0.sort()
    E = []
    for x in E0:
        x.reverse()
        E.append(tuple(x))
    # end ugly hack to get E is sorted by weight
    for e in E:
        # create about |V|/2 edges of T "cheaply"
        TV = T.vertices()
        if not(e[0] in TV) or not(e[1] in TV):
           T.add_edge(e)
    for e in E:
        # connect the "cheapest" components to get T
        C = T.connected_components_subgraphs()
        VC = [S.vertices() for S in C]
        if not(e in T.edges()) and (which_index(e[0],VC) != which_index(e[1],VC)):
           if T.is_connected():
                break
            T.add_edge(e)
    return T
```

Some examples using Sage:

```
sage: A = matrix([[0,1,2,3], [4,0,5,6], [7,8,0,9], [10,11,12,0]])
sage: G = DiGraph(A, format="adjacency_matrix", weighted=True)
sage: boruvka(G)
Multi-graph on 4 vertices
sage: boruvka(G).edges()
[(0, 1, 1), (0, 2, 2), (0, 3, 3)]
```

```
sage: A = matrix([[0,2,0,5,0,0,0], [0,0,8,9,7,0,0], [0,0,0,0,1,0,0],\
...    [0,0,0,0,15,6,0], [0,0,0,0,0,3,4], [0,0,0,0,0,0,11], [0,0,0,0,0,0,0]])
sage: G = Graph(A, format="adjacency_matrix", weighted=True)
sage: E = G.edges(); E
[(0, 1, 2), (0, 3, 5), (1, 2, 8), (1, 3, 9), (1, 4, 7),
(2, 4, 1), (3, 4, 15), (3, 5, 6), (4, 5, 3), (4,6, 4), (5, 6, 11)]
sage: boruvka(G)
Multi-graph on 7 vertices
sage: boruvka(G).edges()
[(0, 1, 2), (0, 3, 5), (2, 4, 1), (3, 5, 6), (4, 5, 3), (4, 6, 4)]
sage: A = matrix([[0,1,2,5], [0,0,3,6], [0,0,0,4], [0,0,0,0]])
sage: G = Graph(A, format="adjacency_matrix", weighted=True)
sage: boruvka(G).edges()
[(0, 1, 1), (0, 2, 2), (2, 3, 4)]
sage: A = matrix([[0,1,5,0,4], [0,0,0,0,3], [0,0,0,2,0], [0,0,0,0,0], [0,0,0,0,0]])
sage: G = Graph(A, format="adjacency_matrix", weighted=True)
sage: boruvka(G).edges()
[(0, 1, 1), (0, 2, 5), (1, 4, 3), (2, 3, 2)]
```

3.3.4 Circuit matrix

Recall, the term *cycle* refers to a closed path. If G is a digraph then a cycle refers to a sequence of edges which form a path in the associated undirected graph. If repeated vertices are allowed, it is more often called a *closed walk*. If the path is a simple path, with no repeated vertices or edges other than the starting and ending vertices, it may also be called a *simple cycle* or *circuit*. A cycle in a directed graph is called a *directed cycle*.

Let $Z_1, \ldots, Z_M$ denote an enumeration of the cycles (simple closed paths) of G. The *cycle matrix* or *circuit matrix* is a $M \times m$ matrix $C = (c_{ij})$ whose rows are parameterized by the cycles and whose columns are parameterized by the edges $E = \{e_1, \ldots, e_m\}$, where

$$c_{ij} = \begin{cases} 1, & e_i \in Z_i, \\ 0, & \text{otherwise.} \end{cases}$$

Recall the incidence matrix was defined in §1.3.2.

Theorem 3.16. *If G is a directed graph then the rows of the incidence matrix $D(G)$ are orthogonal to the rows of $C(G)$.*

Proof. We first show that $C \cdot D^t = 0$. Consider the ith row of C and jth column of D^t. There are non-zero entries in the corresponding entries of these vectors if and only if the jth vertex of G is incident to an edge which occurs in the ith cycle of G. Assume this entry is non-zero. Since G is a directed graph, there is another vertex (the other vertex defining this edge) for which the associated entry is the same but with opposite sign. Therefore the dot product of the ith row of Q and jth column of D^t is zero. ■

The theorem above implies that the column space of the matrix C^t (namely, the image of the associated linear transformation) is contained in the kernel of the incidence matrix D.

Let F be a field. There is a general result called *Sylvester's Law of Nullity* which says that if K is an $r \times s$ matrix over F and L is an $s \times t$ matrix over F (so KL is defined), and if

$$KL = 0,$$

then

$$\text{rank}_F(K) + \text{rank}_F(L) \leq s.$$

(This is an immediate corollary of the rank plus nullity theorem from linear algebra.) It follows from this fact that

$$\text{rank}_{\mathbb{Q}}(D) + \text{rank}_{\mathbb{Q}}(C) \leq m.$$

As a corollary to Theorem 1.28, we see that

$$\text{rank}_{\mathbb{Q}}(D) \leq m - n + 1.$$

Theorem 3.17. *If G is a directed graph then*

$$\text{rank}_{\mathbb{Q}}(C) = m - n + 1.$$

Proof. To show that equality is attained, we exhibit $m-n+1$ linearly independent rows of C. Let $T = (V(T), E(T))$ be a spanning tree for G. This tree has $n-1$ edges and so G has $m-n+1$ edges not in T. Let $e \notin E(T)$ be such an edge. The graph $T+e$ is a cycle. Recall such a cycle is called a fundamental cycle (associated to T). There are exactly $m-n+1$ such cycles. Note that no other fundamental cycle "supports" edge e. Suppose that e is the ith edge in $E(G)$. The row of C associated to the cycle $T+e$ has a 1 in the ith coordinate. Every other row of C associated to a fundamental cycle has a 0 in the ith coordinate. Therefore, the rows of C associated to these $m-n+1$ fundamental cycles are linearly independent. ■

3.3.5 Cutset matrix

The term *cutset* (or *edge cutset*) refers to a sequence of edges in a connected graph $G = (V, E)$ which, when removed from G, creates a disconnected graph and, furthermore, that no proper subset of these edges has this property. When a graph can be disconnected by removing only one edge, that edge is called a *bridge*.

A cutset can be associated to each vertex $v \in V$ as follows.

Lemma 3.18. *The subset $S_v \subset E$ of all the edges incident to v forms forms a cutset.*

This uses only the definitions and is left to the interested reader as an exercise.

Let $S_1, \ldots, S_N$ denote the cutsets of G. The *cutset matrix* $N \times m$ matrix $Q = (q_{ij})$ whose rows are parameterized by the cutsets and whose columns are parameterized by the edges $E = \{e_1, \ldots, e_m\}$, where

$$q_{ij} = \begin{cases} 1, & e_i \in S_i, \\ 0, & \text{otherwise.} \end{cases}$$

Theorem 3.19. *If G is a connected digraph then*

$$\text{rank}_{\mathbb{Q}}(Q) = n - 1.$$

The analog of this theorem for undirected graphs is false (as the examples below show). However, the undirected analog does work if you use the rank over $GF(2)$.

Proof. Let G be any connected graph (directed or not). By Lemma 3.18 above, for each $v \in V$ there is a cut-set S_v consisting of all edges incident to v. The characteristic vector $q_v = (q_1, \ldots, q_m)$ of this set ($q_i = 1$ if $e_i \in S_v$ and $q_i = 0$ otherwise) gives us a row vector in the incidence matrix and in the cut-set matrix. Moreover, all such rows in the incidence matrix are of this form, so

$$\text{rank}_{\mathbb{Q}}(Q) \geq \text{rank}_{\mathbb{Q}}(C) = n - 1.$$

Suppose now G is a directed graph. We show equality must hold. Let S be any edge cutset in G and, for each $v \in V$, let S_v denote the associated cutset as above. If S defined a partitioning

$$V = V_1 \cup V_2$$

then

$$S = \sum_{v \in V_1} S_v = \sum_{v \in V_2} S_v,$$

where the sum of two cutsets is simply the formal sum as oriented edges in the free abelian group $\mathbb{Z}[E]$. From this it is clear that the dimension of the row-span of Q is equal to the dimension of the row-span of the submatrix of Q given by the S_v's. ■

Example 3.20. To construct the example, we will make use of the following Python function.

```
def edge_cutset_matrix(G):
    """
    Returns the edge cutset matrix of the connected graph $G$.
    """
    V = G.vertices()
    E = G.edges()
    rows = []
    for v1 in V:
        for v2 in V:
            if v1 != v2:
                S = G.edge_cut(v1, v2, value_only=False)[1]
                char_S = lambda e: int(bool(e in S))
                rows.append([char_S(e) for e in E])
    Q = matrix(rows)
    return Q
```

We use the function above in the examples below.

For the cube graph in three dimensions (see Figure 3.16) and for the Desargues graph (see Figure 3.17), the undirected analog of Theorem 3.19 does hold.

```
sage: G = graphs.CubeGraph(3)
sage: G
3-Cube: Graph on 8 vertices
sage: Q = edge_cutset_matrix(G)
sage: rank(Q)
7
sage: G = graphs.DesarguesGraph()
sage: G
Desargues Graph: Graph on 20 vertices
sage: edge_cutset_matrix(G).rank()
19
```

On the other hand, for the Frucht graph (see Figure 3.18):

```
sage: G = graphs.FruchtGraph()
sage: G
Frucht graph: Graph on 12 vertices
sage: edge_cutset_matrix(G).rank()
12
```

■

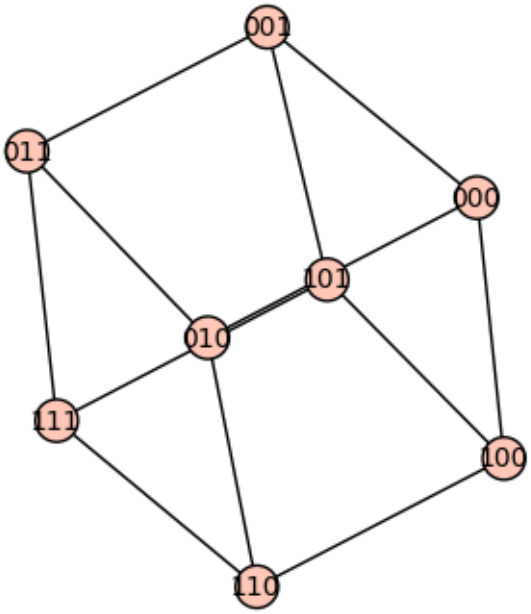


Figure 3.16: The 3-dimensional cube graph.

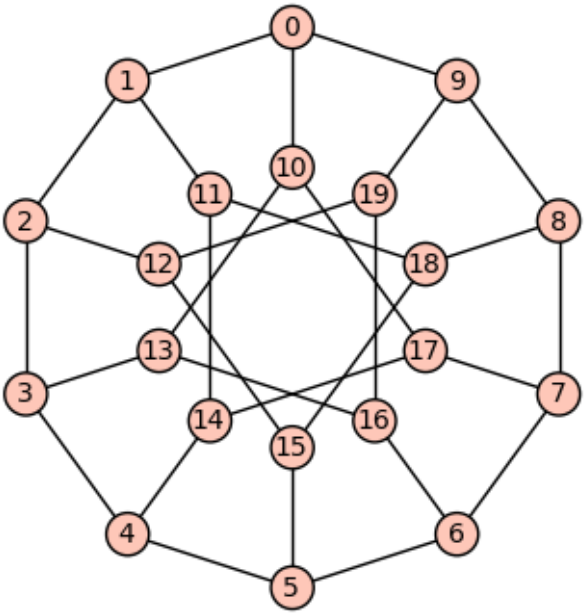


Figure 3.17: The Desargues graph.

These shall be discussed further in §9.1.

3.4 Binary trees

A *binary tree* is a rooted tree with at most two children per parent. Each child is designated as either a *left-child* or a *right-child*. Thus binary trees are also 2-ary trees.

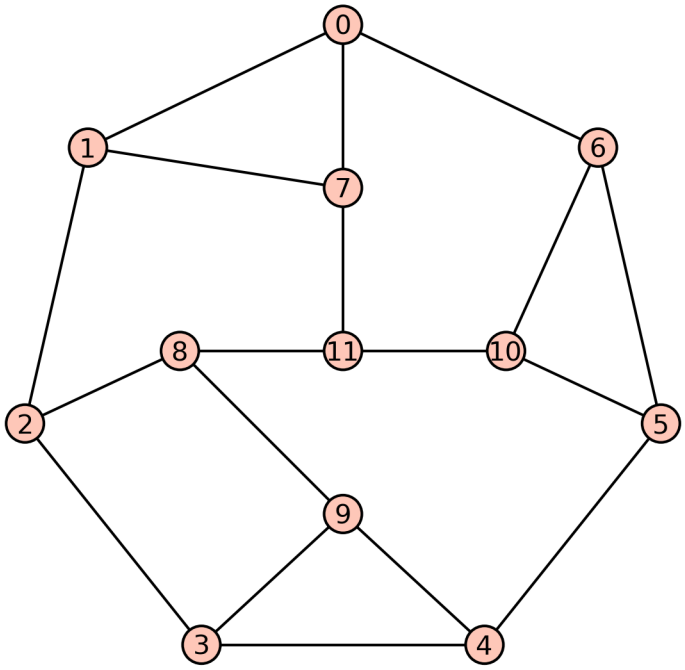


Figure 3.18: The Frucht graph.

Some examples of binary trees are illustrated in Figure 3.19. Given a vertex v in a binary tree T of height h, the *left subtree* of v is comprised of the subtree that spans the left-child of v and all of this child's descendants. The notion of a *right-subtree* of a binary tree is similarly defined. Each of the left and right subtrees of v is itself a binary tree with height $\leq h-1$. If v is the root vertex, then each of its left and right subtrees has height $\leq h-1$, and at least one of these subtrees has height equal to $h-1$.

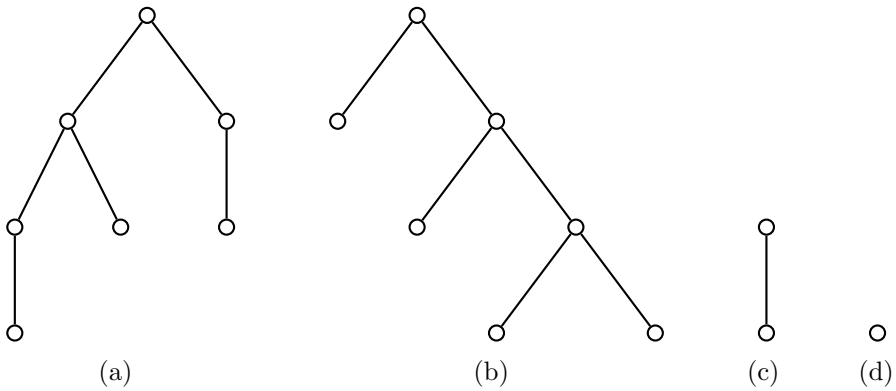


Figure 3.19: Examples of binary trees.

Theorem 3.21. *If T is a complete binary tree of height h, then T has $2^{h+1}-1$ vertices.*

Proof. Argue by induction on h. The assertion of the theorem is trivially true in the base case $h=0$. Let $k \geq 0$ and assume for induction that any complete binary tree of height k has order $2^{k+1}-1$. Suppose T is a complete binary tree of height $k+1$ and denote the left and right subtrees of T by T_1 and T_2, respectively. Each T_i (for $i=1,2$)

is a complete binary tree of height k and by our induction hypothesis T_i has $2^{k+1} - 1$ vertices. Thus T has order

$$1 + (2^{k+1} - 1) + (2^{k+1} - 1) = 2^{k+2} - 1$$

as required. ■

Theorem 3.21 provides a useful upper bound on the order of a binary tree of a given height. This upper bound is stated in the following corollary.

Corollary 3.22. *A binary tree of height h has at most $2^{h+1} - 1$ vertices.*

We now count the number of possible binary trees on n vertices. Let b_n be the number of binary trees of order n. For $n = 0$, we set $b_0 = 1$. The trivial graph is the only binary tree with one vertex, hence $b_1 = 1$. Suppose $n > 1$ and let T be a binary tree on n vertices. Then the left subtree of T has order $0 \leq i \leq n - 1$ and the right subtree has $n-1-i$ vertices. As there are b_i possible left subtrees and b_{n-1-i} possible right subtrees, T has a total of $b_i b_{n-1-i}$ different combinations of left and right subtrees. Summing from $i = 0$ to $i = n - 1$ and we have

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-1-i}. \tag{3.2}$$

Expression (3.2) is known as the *Catalan recursion* and the number b_n is the n-th Catalan number, which we know from problem 1.15 can be expressed in the closed form

$$b_n = \frac{1}{n+1}\binom{2n}{n}. \tag{3.3}$$

Figures 3.20 to 3.22 enumerate all the different binary trees on 2, 3, and 4 vertices, respectively.

Figure 3.20: The $b_2 = 2$ binary trees on 2 vertices.

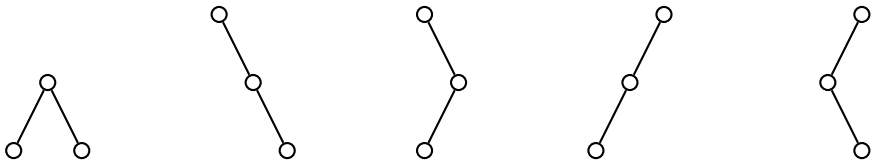


Figure 3.21: The $b_3 = 5$ binary trees on 3 vertices.

The first few values of (3.3) are

$$b_0 = 1, \quad b_1 = 1, \quad b_2 = 2, \quad b_3 = 5, \quad b_4 = 14$$

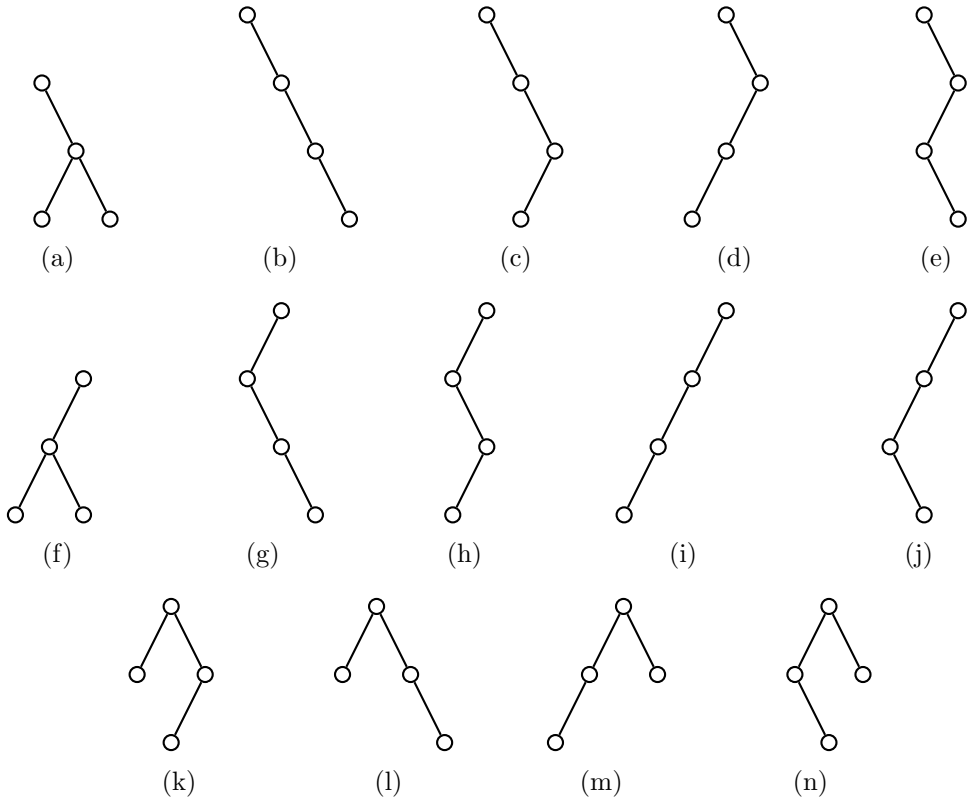


Figure 3.22: The $b_4 = 14$ binary trees on 4 vertices.

which are rather small and of manageable size if we want to explicitly enumerate all different binary trees with the above orders. However, from $n = 4$ onwards the value of b_n increases very fast. Instead of enumerating all the b_n different binary trees of a specified order n, a related problem is generating a random binary tree of order n. That is, we consider the set B as a sample space of b_n different binary trees on n vertices, and choose a random element from B. Such a random element can be generated using Algorithm 3.5. The list `parent` holds all vertices with less than two children, each vertex can be considered as a candidate parent to which we can add a child. An element of `parent` is a two-tuple (v, k) where the vertex v currently has k children.

3.4.1 Binary codes

What is a code?

A *code* is a rule for converting data in one format, or well-defined tangible representation, into sequences of symbols in another format. The finite set of symbols used is called the *alphabet*. We shall identify a code as a finite set of symbols which are the image of the alphabet under this conversion rule. The elements of this set are referred to as *codewords*. For example, using the ASCII code, the letters in the English alphabet get converted into numbers in the set $\{0, 1, \ldots, 255\}$. If these numbers are written in binary, then each codeword of a letter has length 8, i.e. eight bits. In this way, we can reformat or encode a "string" into a sequence of binary symbols, i.e. 0's and 1's. *Encoding* is the conversion process one way. *Decoding* is the reverse process, converting these sequences of code-symbols back into information in the original format.

Algorithm 3.5: Random binary tree.

Input: Positive integer n.
Output: A random binary tree on n vertices.

```
1  if n = 1 then
2      return K_1
3  v ← 0
4  T ← null graph
5  add v to T
6  parent ← [(v, 0)]
7  for i ← 1, 2, ..., n − 1 do
8      (v, k) ← remove random element from parent
9      if k < 1 then
10         add (v, k + 1) to parent
11     add edge (v, i) to T
12     add (i, 0) to parent
13 return T
```

Codes are used for:

- *Economy*. Sometimes this is called *entropy encoding* since there is an entropy function which describes how much information a channel (with a given error rate) can carry and such codes are designed to maximize entropy as best as possible. In this case, in addition to simply being given an alphabet $\mathcal{A}$, one might be given a *weighted alphabet*, i.e. an alphabet for which each symbol $a \in \mathcal{A}$ is associated with a nonnegative number $w_a \geq 0$ (in practice, this number represents the probability that the symbol a occurs in a typical word).

- *Reliability*. Such codes are called *error-correcting codes*, since such codes are designed to communicate information over a noisy channel in such a way that the errors in transmission are likely to be correctable.

- *Security*. Such codes are called *cryptosystems*. In this case, the inverse of the coding function $c : \mathcal{A} \to B^*$ is designed to be computationally infeasible. In other words, the coding function c is designed to be a *trapdoor function*.

Other codes are merely simpler ways to communicate information (e.g. flag semaphores, color codes, genetic codes, braille codes, musical scores, chess notation, football diagrams, and so on) and have little or no mathematical structure. We shall not study them.

Basic definitions

If every word in the code has the same length, the code is called a *block code*. If a code is not a block code, then it is called a *variable-length* code. A *prefix-free* code is a code (typically one of variable-length) with the property that there is no valid codeword in the code that is a prefix or start of any other codeword.[1] This is the *prefix-free condition*.

[1] In other words, a codeword $s = s_1 \cdots s_m$ is a *prefix* of a codeword $t = t_1 \cdots t_n$ if and only if $m \leq n$ and $s_1 = t_1, \ldots, s_m = t_m$. Codes that are prefix-free are easier to decode than codes that are not prefix-free.

One example of a prefix-free code is the ASCII code. Another example is

$$00,\ 01,\ 100.$$

On the other hand, a non-example is the code

$$00,\ 01,\ 010,\ 100$$

since the second codeword is a prefix of the third one. Another non-example is Morse code recalled in Table 3.1, where we use 0 for "·" ("dit") and 1 for "−" ("dah"). For example, consider the Morse code for `a`and the Morse code for `w`. These codewords violate the prefix-free condition.

A	01	N	10
B	1000	O	111
C	1010	P	0110
D	100	Q	1101
E	0	R	010
F	0010	S	000
G	110	T	1
H	0000	U	001
I	00	V	0001
J	0111	W	011
K	101	X	1001
L	0100	Y	1011
M	11	Z	1100

Table 3.1: Morse code

Gray codes

We begin with some history.[2] Frank Gray (1887–1969) wrote about the so-called Gray codes in a 1951 paper published in the Bell System Technical Journal and then in 1953 patented a device (used for television sets) based on his paper. However, the idea of a binary Gray code appeared earlier. In fact, it appeared in an earlier patent (one by Stibitz in 1943). It was also used in the French engineer E. Baudot's telegraph machine of 1878 and in a French booklet by L. Gros on the solution published in 1872 to the Chinese ring puzzle.

The term "Gray code" is ambiguous. It is actually a large family of sequences of n-tuples. Let $\mathbf{Z}_m = \{0, 1, \ldots, m-1\}$. More precisely, an *$m$-ary Gray code of length n* (called a *binary Gray code* when $m = 2$) is a sequence of all possible (i.e. $N = m^n$) n-tuples

$$g_1, g_2, \ldots, g_N$$

where

- each $g_i \in \mathbf{Z}_m^n$,

[2] This history comes from an unpublished section 7.2.1.1 ("Generating all n-tuples") in volume 4 of Donald Knuth's *The Art of Computer Programming.*

- g_i and g_{i+1} differ by 1 in exactly one coordinate.

In other words, an m-ary Gray code of length n is a particular way to order the set of all m^n n-tuples whose coordinates are taken from $\mathbf{Z}_m$. From the transmission/communication perspective, this sequence has two advantages:

- It is easy and fast to produce the sequence, since successive entries differ in only one coordinate.
- An error is relatively easy to detect, since we can compare an n-tuple with the previous one. If they differ in more than one coordinate, we conclude that an error was made.

Example 3.23. Here is a 3-ary Gray code of length 2:

$$[0,0],\ [1,0],\ [2,0],\ [2,1],\ [1,1],\ [0,1],\ [0,2],\ [1,2],\ [2,2]$$

and the sequence

$$[0,0,0],\ [1,0,0],\ [1,1,0],\ [0,1,0],\ [0,1,1],\ [1,1,1],\ [1,0,1],\ [0,0,1]$$

is a binary Gray code of length 3. ■

Gray codes have applications to engineering, recreational mathematics (solving the Tower of Hanoi puzzle, The Brain puzzle, the Chinese ring puzzle, etc.), and to mathematics (e.g. aspects of combinatorics, computational group theory, and the computational aspects of linear codes).

Binary Gray codes

Consider the so-called n-hypercube graph Q_n, whose first few instances are illustrated in Figure 1.33. This can be envisioned as the graph whose vertices are the vertices of a cube in n-space

$$\{(x_1, \ldots, x_n) \mid 0 \leq x_i \leq 1\}$$

and whose edges are those line segments in $\mathbf{R}^n$ connecting two *neighboring* vertices, i.e. two vertices that differ in exactly one coordinate. A binary Gray code of length n can be regarded as a path on the hypercube graph Q_n that visits each vertex of the cube exactly once. In other words, a binary Gray code of length n may be identified with a Hamiltonian path on the graph Q_n. For example, Figure 3.23 illustrates a Hamiltonian path on Q_3.

How do we efficiently compute a Gray code? Perhaps the simplest way to state the idea of quickly constructing the *reflected binary Gray code* Γ_n of length n is as follows:

$$\begin{aligned}\Gamma_0 &= [\,],\\ \Gamma_n &= \big[[0, \Gamma_{n-1}],\ [1, \Gamma_{n-1}^{\text{rev}}]\big]\end{aligned}$$

where Γ_m^{rev} means the Gray code in reverse order. For instance, we have

$$\begin{aligned}\Gamma_0 &= [\,],\\ \Gamma_1 &= \big[[0],\ [1]\big],\\ \Gamma_2 &= \big[[0,0],\ [0,1],\ [1,1],\ [1,0]\big]\end{aligned}$$

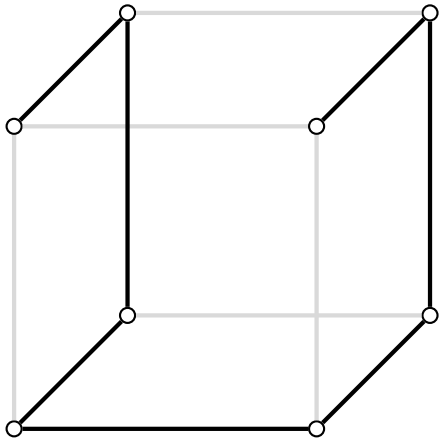


Figure 3.23: Viewing Γ_3 as a Hamiltonian path on Q_3.

and so on. This is a nice procedure for creating the entire list at once, which gets very long very fast. An implementation of the reflected Gray code using Python is given below.

```
def graycode(length,modulus):
    """
    Returns the n-tuple reflected Gray code mod m.


    EXAMPLES:
        sage: graycode(2,4)

        [[0, 0],
         [1, 0],
         [2, 0],
         [3, 0],
         [3, 1],
         [2, 1],
         [1, 1],
         [0, 1],
         [0, 2],
         [1, 2],
         [2, 2],
         [3, 2],
         [3, 3],
         [2, 3],
         [1, 3],
         [0, 3]]
    """
    n,m = length,modulus
    F = range(m)
    if n == 1:
        return [[i] for i in F]
    L = graycode(n-1, m)
    M = []
    for j in F:
        M = M+[ll+[j] for ll in L]
    k = len(M)
    Mr = [0]*m
    for i in range(m-1):
        i1 = i*int(k/m)        # this requires Python 3.0 or Sage
        i2 = (i+1)*int(k/m)
        Mr[i] = M[i1:i2]
    Mr[m-1] = M[(m-1)*int(k/m):]
    for i in range(m):
        if is_odd(i):
            Mr[i].reverse()
    M0 = []
    for i in range(m):
        M0 = M0+Mr[i]
    return M0
```

Consider the reflected binary code of length 8, i.e. Γ_8. This has $2^8 = 256$ codewords. Sage can easily create the list plot of the coordinates (x, y), where x is an integer $j \in \mathbf{Z}_{256}$ that indexes the codewords in Γ_8 and the corresponding y is the j-th codeword in Γ_8 converted to decimal. This will give us some idea of how the Gray code "looks" in some sense. The plot is given in Figure 3.24.

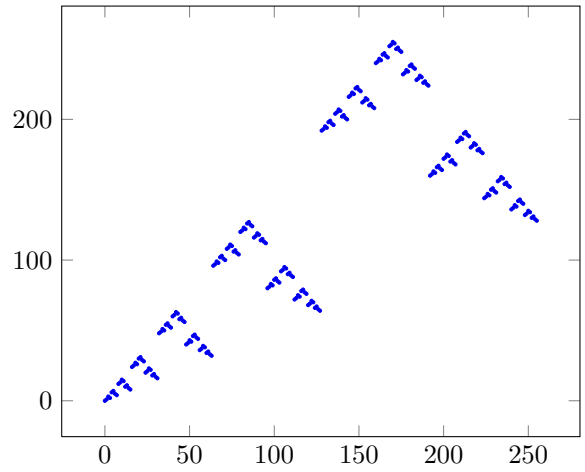


Figure 3.24: Scatterplot of Γ_8.

What if we only want to compute the i-th Gray codeword in the Gray code of length n? Can it be computed quickly without computing the entire list? At least in the case of the reflected binary Gray code, there is a very simple way to do this. The k-th element in the above-described reflected binary Gray code of length n is obtained by simply adding the binary representation of k to the binary representation of the integer part of $k/2$. An example using Sage is given below.

```
def int2binary(m, n):
    '''
    returns GF(2) vector of length n obtained
    from the binary repr of m, padded by 0's
    (on the left) to length n.

    EXAMPLES:
        sage: for j in range(8):
        ....:     print int2binary(j,3)+int2binary(int(j/2),3)
        ....:
        (0, 0, 0)
        (0, 0, 1)
        (0, 1, 1)
        (0, 1, 0)
        (1, 1, 0)
        (1, 1, 1)
        (1, 0, 1)
        (1, 0, 0)
    '''
    s = bin(m)
    k = len(s)
    F = GF(2)
    b = [F(0)]*n
    for i in range(2,k):
        b[n-k+i] = F(int(s[i]))
    return vector(b)

def graycodeword(m, n):
    '''
    returns the k-th codeword in the reflected binary Gray code
    of length n.

    EXAMPLES:
        sage: graycodeword(3,3)
```

```
    (0, 1, 0)
'''
return map(int, int2binary(m,n)+int2binary(int(m/2),n))
```

3.5 Huffman codes

An *alphabet* $\mathcal{A}$ is a finite set whose elements are referred to as *symbols*. A *word* (or *string* or *message*) over $\mathcal{A}$ is a finite sequence of symbols in $\mathcal{A}$ and the *length* of the word is the number of symbols it contains. A word is usually written by concatenating symbols together, e.g. $a_1a_2\cdots a_k$ ($a_i \in \mathcal{A}$) is a word of length k.

A commonly occurring alphabet in practice is the *binary alphabet* $\mathbb{B} = \{0, 1\}$. A word over the binary alphabet is a finite sequence of 0's and 1's. If $\mathcal{A}$ is an alphabet, let $\mathcal{A}^*$ denote the set of all words in $\mathcal{A}$. The length of a word is denoted by vertical bars. That is, if $w = a_1 \cdots a_k$ is a word over $\mathcal{A}$, then define $|w| : \mathcal{A}^* \to \mathbf{Z}$ by

$$|w| = |a_1 \cdots a_k| = k.$$

Let $\mathcal{A}$ and $\mathcal{B}$ be two alphabets. A *code* for $\mathcal{A}$ using $\mathcal{B}$ is an injection $c : \mathcal{A} \to \mathcal{B}^*$. By abuse of notation, we often denote the code simply by the set

$$C = c(A) = \{c(a) \mid a \in \mathcal{A}\}.$$

The elements of C are called *codewords*. If $\mathcal{B}$ is the binary alphabet, then C is called a *binary code*.

3.5.1 Tree representation

Any binary code can be represented by a tree, as Example 3.24 shows.

Example 3.24. *Let $\mathbb{B}_\ell$ be the binary code of length $\leq \ell$. Represent codewords of $\mathbb{B}_\ell$ using trees.*

Solution. Here is how to represent the code $\mathbb{B}_\ell$ consisting of all binary strings of length $\leq \ell$. Start with the root node ε being the empty string. The two children of this node, v_0 and v_1, correspond to the two strings of length 1. Label v_0 with a "0" and v_1 with a "1". The two children of v_0, i.e. v_{00} and v_{01}, correspond to the strings of length 2 which start with a 0. Similarly, the two children of v_1, i.e. v_{10} and v_{11}, correspond to the strings of length 2 that each starts with a 1. Continue creating child nodes until we reach length ℓ, at which point we stop. There are a total of $2^{\ell+1} - 1$ nodes in this tree and 2^ℓ of them are leaves (vertices of a tree with degree 1, i.e. childless nodes). Note that the parent of any node is a prefix to that node. Label each node v_s with the string "s", where s is a binary sequence of length $\leq \ell$. See Figure 3.25 for an example when $\ell = 2$. ■

In general, if C is a code contained in $\mathbb{B}_\ell$, then to create the tree for C, start with the tree for $\mathbb{B}_\ell$. First, remove all nodes associated to a binary string for which it and all of its descendants are not in C. Next, remove all labels which do not correspond to codewords in C. The resulting labeled graph is the tree associated to the binary code C.

For visualizing the construction of Huffman codes later, it is important to see that we can *reverse* this construction to start from such a binary tree and recover a binary code from it. The codewords are determined by the following rules:

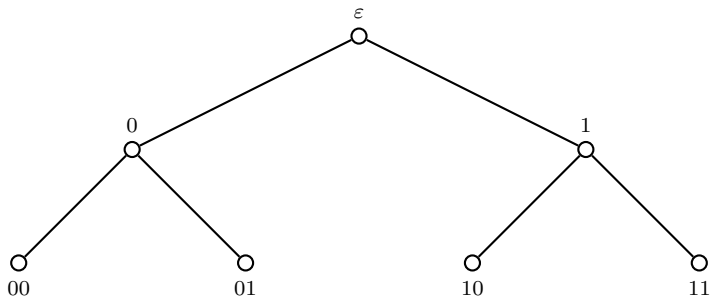


Figure 3.25: Tree representation of the binary code $\mathbb{B}_2$.

- The root node gets the empty codeword.
- Each left-ward branch gets a 0 appended to the end of its parent. Each right-ward branch gets a 1 appended to the end.

3.5.2 Uniquely decodable codes

If $c : \mathcal{A} \to \mathcal{B}^*$ is a code, then we can extend c to $\mathcal{A}^*$ by concatenation:

$$c(a_1 a_2 \cdots a_k) = c(a_1)c(a_2)\cdots c(a_k).$$

If the extension $c : \mathcal{A}^* \to \mathcal{T}^*$ is also an injection, then c is called *uniquely decodable*. The property of unique decodability or decipherability informally means that any given sequence of symbols has at most one interpretation as a sequence of codewords.

Example 3.25. *Is the Morse code in Table 3.1 uniquely decodable? Why or why not?*

Solution. Note that these Morse codewords all have lengths less than or equal to 4. Other commonly occurring symbols used (the digits 0 through 9, punctuation symbols, and some others) are also encodable in Morse code, but they use longer codewords.

Let $\mathcal{A}$ denote the English alphabet, $\mathbb{B} = \{0, 1\}$ the binary alphabet, and $c : \mathcal{A} \to \mathbb{B}^*$ the Morse code. Since $c(ET) = 01 = c(A)$, it is clear that the Morse code is *not* uniquely decodable. ■

In fact, prefix-free implies uniquely decodable.

Theorem 3.26. *If a code $c : \mathcal{A} \to \mathcal{B}^*$ is prefix-free, then it is uniquely decodable.*

Proof. We use induction on the length of a message. We want to show that if $x_1 \cdots x_k$ and $y_1 \cdots y_\ell$ are messages with $c(x_1)\cdots c(x_k) = c(y_1)\cdots c(y_\ell)$, then $x_1 \cdots x_k = y_1 \cdots y_\ell$. This in turn implies $k = \ell$ and $x_i = y_i$ for all i.

The case of length 1 follows from the fact that $c : \mathcal{A} \to \mathcal{B}^*$ is injective (by the definition of code).

Suppose that the statement of the theorem holds for all codes of length $< m$. We must show that the length m case is true. Suppose $c(x_1)\cdots c(x_k) = c(y_1)\cdots c(y_\ell)$, where $m = \max(k, \ell)$. These strings are equal, so the substring $c(x_1)$ of the left-hand side and the substring $c(y_1)$ of the right-hand side are either equal or one is contained in the other. If, say, $c(x_1)$ is properly contained in $c(y_1)$, then c is not prefix-free. Likewise if $c(y_1)$ is properly contained in $c(x_1)$. Therefore, $c(x_1) = c(y_1)$, which implies $x_1 = y_1$. Now remove this codeword from both sides, so $c(x_2)\cdots c(x_k) = c(y_2)\cdots c(y_\ell)$. By the induction hypothesis, $x_2 \cdots x_k = y_2 \cdots y_\ell$. These facts together imply $k = \ell$ and $x_i = y_i$ for all i. ■

Consider now a weighted alphabet $(\mathcal{A}, p)$, where $p : \mathcal{A} \to [0,1]$ satisfies $\sum_{a\in\mathcal{A}} p(a) = 1$, and a code $c : \mathcal{A} \to \mathcal{B}^*$. In other words, p is a probability distribution on $\mathcal{A}$. Think of $p(a)$ as the probability that the symbol a arises in a typical message. The *average word length* $L(c)$ is[3]

$$L(c) = \sum_{a\in\mathcal{A}} p(a) \cdot |c(a)|$$

where $|\cdot|$ is the length of a codeword. Given a weighted alphabet $(\mathcal{A}, p)$ as above, a code $c : \mathcal{A} \to \mathcal{B}^*$ is called *optimal* if there is no such code with a smaller average word length. Optimal codes satisfy the following amazing property. For a proof, which is very easy and highly recommended for anyone who is curious to see more, refer to section 3.6 of Biggs [**?**].

Lemma 3.27. *Suppose $c : \mathcal{A} \to \mathbb{B}^*$ is a binary optimal prefix-free code and let $\ell = \max_{a\in\mathcal{A}} \big(|c(a)|\big)$ denote the maximum length of a codeword. The following statements hold.*

1. *If $|c(a')| > |c(a)|$, then $p(a') \le p(a)$.*
2. *The subset of codewords of length ℓ, i.e.*

$$C_\ell = \{c \in c(\mathcal{A}) \mid \ell = |c(a)|\}$$

contains two codewords of the form $b0$ and $b1$ for some $b \in \mathbb{B}^*$.

3.5.3 Huffman coding

The Huffman code construction is based on the second property in Lemma 3.27. Using this property, in 1952 David Huffman [**?**] presented an optimal prefix-free binary code, which has since been named Huffman code.

Here is the recursive/inductive construction of a Huffman code. We shall regard the binary Huffman code as a tree, as described above. Suppose that the weighted alphabet $(\mathcal{A}, p)$ has n symbols. We assume inductively that there is an optimal prefix-free binary code for any weighted alphabet $(\mathcal{A}', p')$ having $< n$ symbols.

Huffman's rule 1 Let $a, a' \in \mathcal{A}$ be symbols with the smallest weights. Construct a new weighted alphabet with a, a' replaced by the single symbol $a^* = aa'$ and having weight $p(a^*) = p(a) + p(a')$. All other symbols and weights remain unchanged.

Huffman's rule 2 For the code $(\mathcal{A}', p')$ above, if a^* is encoded as the binary string s, then the encoded binary string for a is $s0$ and the encoded binary string for a' is $s1$.

The above two rules tell us how to inductively build the tree representation for the Huffman code of $(\mathcal{A}, p)$ up from its leaves (associated to the low weight symbols).

- Find two different symbols of lowest weight, a and a'. If two such symbols do not exist, stop. Replace the weighted alphabet with the new weighted alphabet as in Huffman's rule 1.

[3] In probability terminology, this is the expected value $\mathrm{E}(X)$ of the random variable X, which assigns to a randomly selected symbol in $\mathcal{A}$ the length of the associated codeword in c.

- Add two nodes (labeled with a and a', respectively) to the tree, with parent a^* (see Huffman's rule 1).
- If there are no remaining symbols in $\mathcal{A}$, label the parent a^* with the empty set and stop. Otherwise, go to the first step.

These ideas are captured in Algorithm 3.6, which outlines steps to construct a binary tree corresponding to the Huffman code of an alphabet. Line 2 initializes a minimum-priority queue Q with the symbols in the alphabet A. Line 3 creates an empty binary tree that will be used to represent the Huffman code corresponding to A. The for loop from lines 4 to 10 repeatedly extracts from Q two elements a and b of minimum weights. We then create a new vertex z for the tree T and also let a and b be vertices of T. The weight $W[z]$ of z is the sum of the weights of a and b. We let z be the parent of a and b, and insert the new edges za and zb into T. The newly created vertex z is now inserted into Q with priority $W[z]$. After $n-1$ rounds of the for loop, the priority queue has only one element in it, namely the root r of the binary tree T. We extract r from Q (line 11) and return it together with T (line 12).

Algorithm 3.6: Binary tree representation of Huffman codes.

Input: An alphabet A of n symbols. A weight list W of size n such that $W[i]$ is the weight of $a_i \in A$.

Output: A binary tree T representing the Huffman code of A and the root r of T.

```
1  n ← |A|
2  Q ← A                          /* minimum priority queue */
3  T ← empty tree
4  for i ← 1, 2, ..., n − 1 do
5      a ← extractMin(Q)
6      b ← extractMin(Q)
7      z ← node with left child a and right child b
8      add the edges za and zb to T
9      W[z] ← W[a] + W[b]
10     insert z into priority queue Q
11 r ← extractMin(Q)
12 return (T, r)
```

The runtime analysis of Algorithm 3.6 depends on the implementation of the priority queue Q. Suppose Q is a simple unsorted list. The initialization on line 2 requires $O(n)$ time. The for loop from line 4 to 10 is executed exactly $n-1$ times. Searching Q to determine the element of minimum weight requires time at most $O(n)$. Determining two elements of minimum weights requires time $O(2n)$. The for loop requires time $O(2n^2)$, which is also the time requirement for the algorithm. An efficient implementation of the priority queue Q, e.g. as a binary minimum heap, can lower the running time of Algorithm 3.6 down to $O(n \log_2(n))$.

Algorithm 3.6 represents the Huffman code of an alphabet as a binary tree T rooted at r. For an illustration of the process of constructing a Huffman tree, see Figure 3.26. To determine the actual encoding of each symbol in the alphabet, we feed T and r to Algorithm 3.7 to obtain the encoding of each symbol. Starting from the root r whose designated label is the empty string ε, the algorithm traverses the vertices of T in a

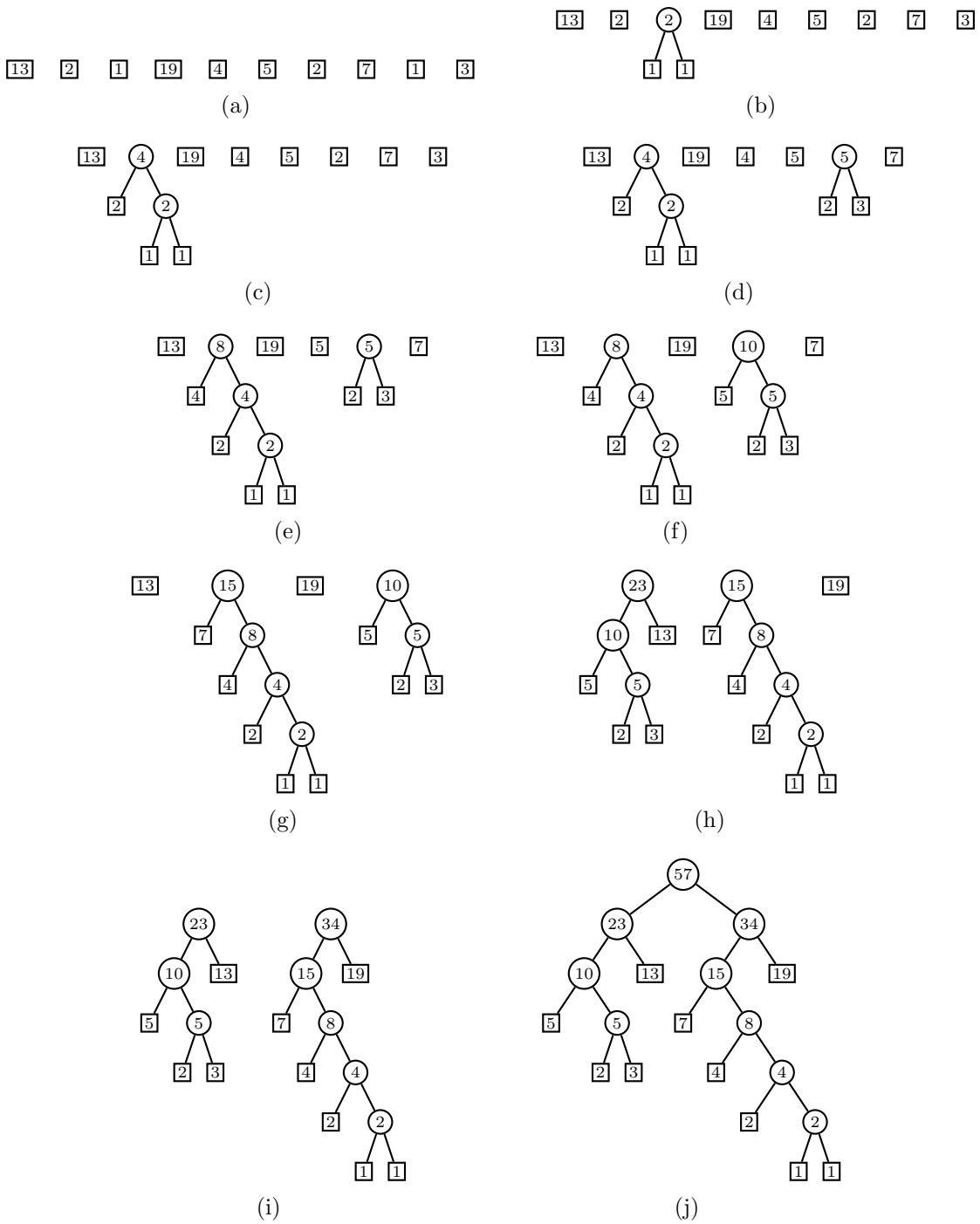


Figure 3.26: Constructing a Huffman tree.

breadth-first search fashion. If v is an internal vertex with label `e`, the label of its left-child is the concatenation `e0` and for the right-child of v we assign the label `e1`. If v happens to be a leaf vertex, we take its label to be its Huffman encoding. Any Huffman encoding assigned to a symbol of an alphabet is not unique. Either of the two children of an internal vertex can be designated as the left- (respectively, right-) child. The runtime of Algorithm 3.7 is $O(|V|)$, where V is the vertex set of T.

Algorithm 3.7: Huffman encoding of an alphabet.

Input: A binary tree T representing the Huffman code of an alphabet A. The root r of T.

Output: A list H representing a Huffman code of A, where $H[a_i]$ corresponds to a Huffman encoding of $a_i \in A$.

```
1  H ← [ ]                          /* list of Huffman encodings */
2  Q ← [r]                              /* queue of vertices */
3  while length(Q) > 0 do
4      root ← dequeue(Q)
5      if root is a leaf then
6          H[root] ← label of root
7      else
8          a ← left child of root
9          b ← right child of root
10         enqueue(Q, a)
11         enqueue(Q, b)
12         label of a ← label of root + 0
13         label of b ← label of root + 1
14 return H
```

Example 3.28. *Consider the alphabet $\mathcal{A} = \{a, b, c, d, e, f\}$ with corresponding weights $w(a) = 19$, $w(b) = 2$, $w(c) = 40$, $w(d) = 25$, $w(e) = 31$, and $w(f) = 3$. Construct a binary tree representation of the Huffman code of $\mathcal{A}$ and determine the encoding of each symbol of $\mathcal{A}$.*

Solution. Use Algorithm 3.6 to construct a binary tree representation of the weighted alphabet $\mathcal{A}$. The resulting binary tree T is shown in Figure 3.27(a), where $a_i : w_i$ is an abbreviation for "vertex a_i has weight w_i". The binary tree is rooted at k. To encode each alphabetic symbol, input T and k into Algorithm 3.7 to get the encodings shown in Figure 3.27(b). ■

3.6 Tree traversals

In computer science, *tree traversal* refers to the process of examining each vertex in a tree data structure. Starting at the root of an ordered tree T, we can traverse the vertices of T in one of various ways.

A *level-order traversal* of an ordered tree T examines the vertices in increasing order of depth, with vertices of equal depth being examined according to their prescribed order. One way to think about level-order traversal is to consider vertices of T having

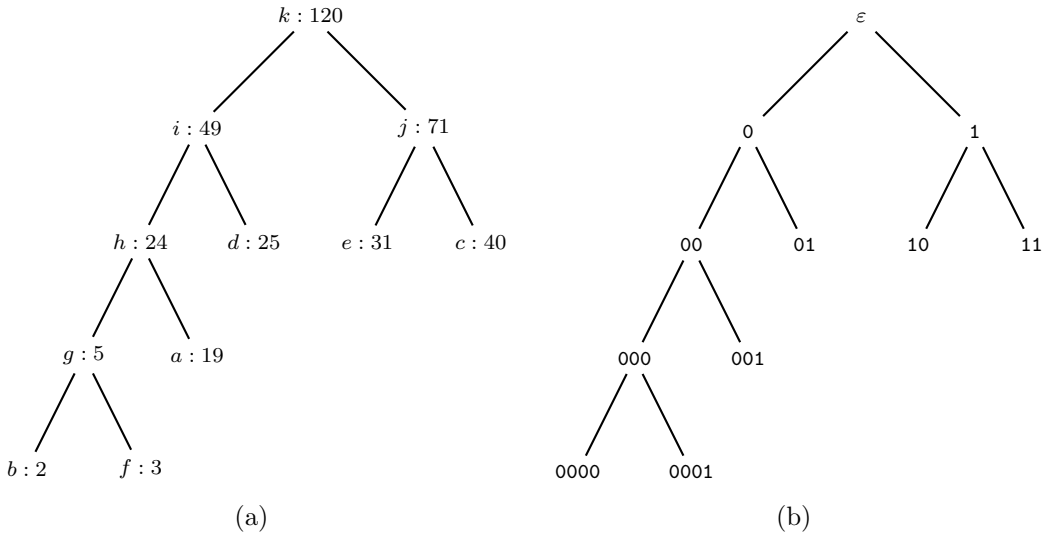


Figure 3.27: Binary tree representation of an alphabet and its Huffman encodings.

the same depth as being ordered from left to right in decreasing order of importance. If $[v_1, v_2, \ldots, v_n]$ lists the vertices from left to right at depth k, a decreasing order of importance can be realized by assigning each vertex a numeric label using a labelling function $L : V(T) \to \mathbf{R}$ such that $L(v_1) < L(v_2) < \cdots < L(v_n)$. In this way, a vertex with a lower numeric label is examined prior to a vertex with a higher numeric label. A level-order traversal of T, whose vertices of equal depth are prioritized according to L, is an examination of the vertices of T from top to bottom, left to right. As an example, the level-order traversal of the tree in Figure 3.28 is

$$42,\ 4,\ 15,\ 2,\ 3,\ 5,\ 7,\ 10,\ 11,\ 12,\ 13,\ 14.$$

Our discussion is formalized in Algorithm 3.8, whose general structure mimics that of breadth-first search. For this reason, level-order traversal is also known as *breadth-first traversal*. Each vertex is enqueued and dequeued exactly once. The while loop is executed n times, hence we have a runtime of $O(n)$. Another name for level-order traversal is *top-down traversal* because we first visit the root node and then work our way down the tree, increasing the depth as we move downward.

Pre-order traversal is a traversal of an ordered tree using a general strategy similar to depth-first search. For this reason, pre-order traversal is also referred to as *depth-first traversal*. Parents are visited prior to their respective children and siblings are visited according to their prescribed order. The pseudocode for pre-order traversal is presented in Algorithm 3.9. Note the close resemblance to Algorithm 3.8; the only significant change is to use a stack instead of a queue. Each vertex is pushed and popped exactly once, so the while loop is executed n times, resulting in a runtime of $O(n)$. Using Algorithm 3.9, a pre-order traversal of the tree in Figure 3.28 is

$$42,\ 4,\ 2,\ 3,\ 10,\ 11,\ 14,\ 5,\ 12,\ 13,\ 15,\ 7.$$

Whereas pre-order traversal lists a vertex v the first time we visit it, *post-order traversal* lists v the last time we visit it. In other words, children are visited prior to their respective parents, with siblings being visited in their prescribed order. The prefix "pre" in "pre-order traversal" means "before", i.e. visit parents before visiting children. On the

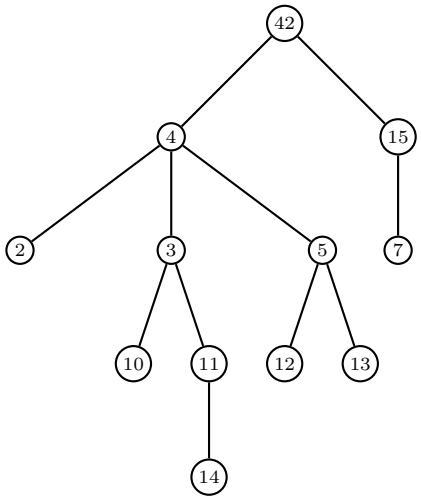


Figure 3.28: Traversing a tree.

Algorithm 3.8: Level-order traversal.

Input: An ordered tree T on $n > 0$ vertices.
Output: A list of the vertices of T in level-order.

```
1  L ← [ ]
2  Q ← empty queue
3  r ← root of T
4  enqueue(Q, r)
5  while length(Q) > 0 do
6      v ← dequeue(Q)
7      append(L, v)
8      [u_1, u_2, ..., u_k] ← ordering of children of v
9      for i ← 1, 2, ..., k do
10         enqueue(Q, u_i)
11 return L
```

Algorithm 3.9: Pre-order traversal.

Input: An ordered tree T on $n > 0$ vertices.
Output: A list of the vertices of T in pre-order.

```
1  L ← [ ]
2  S ← empty stack
3  r ← root of T
4  push(S, r)
5  while length(S) > 0 do
6      v ← pop(S)
7      append(L, v)
8      [u_1, u_2, ..., u_k] ← ordering of children of v
9      for i ← k, k − 1, ..., 1 do
10         push(S, u_i)
11 return L
```

other hand, the prefix "post" in "post-order traversal" means "after", i.e. visit parents after having visited their children. The pseudocode for post-order traversal is presented in Algorithm 3.10, whose general structure bears close resemblance to Algorithm 3.9. The while loop of the former is executed n times because each vertex is pushed and popped exactly once, resulting in a runtime of $O(n)$. The post-order traversal of the tree in Figure 3.28 is

$$2, 10, 14, 11, 3, 12, 13, 5, 4, 7, 15, 42.$$

Algorithm 3.10: Post-order traversal.

Input: An ordered tree T on $n > 0$ vertices.
Output: A list of the vertices of T in post-order.

```
1  L ← [ ]
2  S ← empty stack
3  r ← root of T
4  push(S, r)
5  while length(S) > 0 do
6      if top(S) is unmarked then
7          mark top(S)
8          [u_1, u_2, ..., u_k] ← ordering of children of top(S)
9          for i ← k, k − 1, ..., 1 do
10             push(S, u_i)
11     else
12         v ← pop(S)
13         append(L, v)
14 return L
```

Instead of traversing a tree T from top to bottom as is the case with level-order traversal, we can reverse the direction of our traversal by traversing a tree from bottom to top. Called *bottom-up traversal*, we first visit all the leaves of T and consider the subtree T_1 obtained by vertex deletion of those leaves. We then recursively perform bottom-up traversal of T_1 by visiting all of its leaves and obtain the subtree T_2 resulting from vertex deletion of those leaves of T_1. Apply bottom-up traversal to T_2 and its vertex deletion subtrees until we have visited all vertices, including the root vertex. The result is a procedure for bottom-up traversal as presented in Algorithm 3.11. In lines 3 to 5, we initialize the list C to contain the number of children of vertex i. This takes $O(m)$ time, where $m = |E(T)|$. Lines 6 to 14 extract all the leaves of T and add them to the queue Q. From lines 15 to 23, we repeatedly apply bottom-up traversal to subtrees of T. As each vertex is enqueued and dequeued exactly once, the two loops together run in time $O(n)$ and therefore Algorithm 3.11 has a runtime of $O(n + m)$. As an example, a bottom-up traversal of the tree in Figure 3.28 is

$$2, 7, 10, 12, 13, 14, 15, 5, 11, 3, 4, 42.$$

Yet another common tree traversal technique is called *in-order traversal*. However, in-order traversal is only applicable to binary trees, whereas the other traversal techniques we considered above can be applied to any tree with at least one vertex. Given a binary tree T having at least one vertex, in-order traversal first visits the root of T and consider

Algorithm 3.11: Bottom-up traversal.

Input: An ordered tree T on $n > 0$ vertices.
Output: A list of the vertices of T in bottom-up order.

```
1  Q ← empty queue
2  r ← root of T
3  C ← [0, 0, ..., 0]                          /* n copies of 0 */
4  for each edge (u, v) ∈ E(T) do
5      C[u] ← C[u] + 1
6  R ← empty queue
7  enqueue(R, r)
8  while length(R) > 0 do
9      v ← dequeue(R)
10     for each w ∈ children(v) do
11         if C[w] = 0 then
12             enqueue(Q, w)
13         else
14             enqueue(R, w)
15 L ← [ ]
16 while length(Q) > 0 do
17     v ← dequeue(Q)
18     append(L, v)
19     if v ≠ r then
20         C[parent(v)] ← C[parent(v)] − 1
21         if C[parent(v)] = 0 then
22             u ← parent(v)
23             enqueue(Q, u)
24 return L
```

Algorithm 3.12: In-order traversal.

Input: A binary tree T on $n > 0$ vertices.
Output: A list of the vertices of T in in-order.

```
1  L ← [ ]
2  S ← empty stack
3  v ← root of T
4  while True do
5      if v ≠ NULL then
6          push(S, v)
7          v ← left-child of v
8      else
9          if length(S) = 0 then
10             exit the loop
11         v ← pop(S)
12         append(L, v)
13         v ← right-child of v
14 return L
```

its left- and right-children. We then recursively apply in-order traversal to the left and right subtrees of the root vertex. Notice the symmetry in our description of in-order traversal: start at the root, then traverse the left and right subtrees in in-order. For this reason, in-order traversal is sometimes referred to as *symmetric traversal*. Our discussion is summarized in Algorithm 3.12. In the latter algorithm, if a vertex does not have a left-child, then the operation of finding its left-child returns `NULL`. The same holds when the vertex does not have a right-child. Since each vertex is pushed and popped exactly once, it follows that in-order traversal runs in time $O(n)$. Using Algorithm 3.12, an in-order traversal of the tree in Figure 3.27(b) is

$$\texttt{0000}, \texttt{000}, \texttt{0001}, \texttt{00}, \texttt{001}, \texttt{0}, \texttt{01}, \varepsilon, \texttt{10}, \texttt{1}, \texttt{11}.$$

3.7 Problems

> When solving problems, dig at the roots instead of just hacking at the leaves.
> — Anthony J. D'Angelo, *The College Blue Book*

3.1. Construct all nonisomorphic trees of order 7.

3.2. Let G be a weighted connected graph and let T be a subgraph of G. Then T is a *maximum spanning tree* of G provided that the following conditions are satisfied:

(a) T is a spanning tree of G.

(b) The total weight of T is maximum among all spanning trees of G.

Modify Kruskal's, Prim's, and Borůvka's algorithms to return a maximum spanning tree of G.

3.3. Describe and present pseudocode of an algorithm to construct all spanning trees of a connected graph. What is the worst-case runtime of your algorithm? How many of the constructed spanning trees are nonisomorphic to each other? Repeat the exercise for minimum and maximum spanning trees.

3.4. Consider an undirected, connected simple graph $G = (V, E)$ of order n and size m and having an integer weight function $w : E \to \mathbf{Z}$ given by $w(e) > 0$ for all $e \in E$. Suppose that G has N minimum spanning trees. Yamada et al. [**?**] provide an $O(Nm \ln n)$ algorithm to construct all the N minimum spanning trees of G. Describe and provide pseudocode of the Yamada-Kataoka-Watanabe algorithm. Provide runtime analysis and prove the correctness of this algorithm.

3.5. The solution of Example 3.3 relied on the following result: Let $T = (V, E)$ be a tree rooted at v_0 and suppose v_0 has exactly two children. If $\max_{v \in V} \deg(v) = 3$ and v_0 is the only vertex with degree 2, then T is a binary tree. Prove this statement. Give examples of graphs that are binary trees but do not satisfy the conditions of the result. Under which conditions would the above test return an incorrect answer?

3.6. What is the worst-case runtime of Algorithm 3.1?

3.7. Figure 3.5 shows two nonisomorphic spanning trees of the 4×4 grid graph.

(a) For each $n = 1, 2, \ldots, 7$, construct all nonisomorphic spanning trees of the $n \times n$ grid graph.

(b) Explain and provide pseudocode of an algorithm for constructing all spanning trees of the $n \times n$ grid graph, where $n > 0$.

(c) In general, if n is a positive integer, how many nonisomorphic spanning trees are there in the $n \times n$ grid graph?

(d) Describe and provide pseudocode of an algorithm to generate a random spanning tree of the $n \times n$ grid graph. What is the worst-case runtime of your algorithm?

3.8. Theorem 3.4 shows how to recursively construct a new tree from a given collection of trees, hence it can be considered as a recursive definition of trees. To prove theorems based upon recursive definitions, we use a proof technique called *structural induction*. Let $S(C)$ be a statement about the collection of structures C, each of which is defined by a recursive definition. In the base case, prove $S(C)$ for the basis structure(s) C. For the inductive case, let X be a structure formed using the recursive definition from the structures $Y_1, Y_2, \ldots, Y_k$. Assume for induction that the statements $S(Y_1)$, $S(Y_2), \ldots, S(Y_k)$ hold and use the inductive hypotheses $S(Y_i)$ to prove $S(X)$. Hence conclude that $S(X)$ is true for all X. Apply structural induction to show that any graph constructed using Theorem 3.4 is indeed a tree.

3.9. In Kruskal's Algorithm 3.2, line 5 requires that the addition of a new edge to T does not result in T having a cycle. A tree by definition has no cycles. Suppose line 5 is changed to:

$$\textbf{if } e_i \notin E(T) \text{ and } T \cup \{e_i\} \text{ is a tree } \textbf{then}$$

With this change, explain why Algorithm 3.2 would return a minimum spanning tree or why the algorithm would fail to do so.

3.10. This problem is concerned with improving the runtime of Kruskal's Algorithm 3.2. Explain how to use a priority queue to obviate the need for sorting the edges by weight. Investigate the union-find data structure. Explain how to use union-find to ensure that the addition of each edge results in an acyclic graph.

3.11. Figure 3.29 shows a weighted version of the Chvátal graph, which has 12 vertices and 24 edges. Use this graph as input to Kruskal's, Prim's, and Borůvka's algorithms and compare the resulting minimum spanning trees.

3.12. Algorithm 3.1 presents a randomized procedure to construct a spanning tree of a given connected graph via repeated edge deletion.

(a) Describe and present pseudocode of a randomized algorithm to *grow* a spanning tree via edge addition.

(b) Would Algorithm 3.1 still work if the input graph G has self-loops or multiple edges? Explain why or why not. If not, modify Algorithm 3.1 to handle the case where G has self-loops and multiple edges.

(c) Repeat the previous exercise for Kruskal's, Prim's, and Borůvka's algorithms.

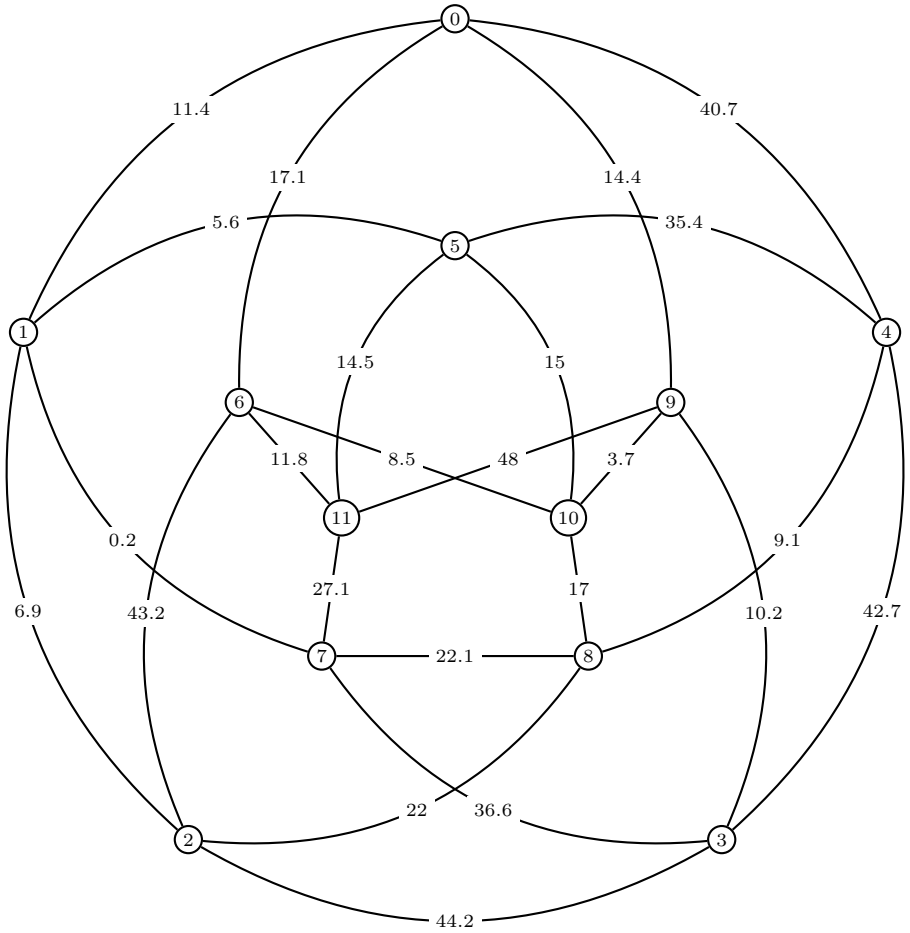


Figure 3.29: Weighted Chvátal graph.

Algorithm 3.13: Random spanning tree of K_n.

Input: A positive integer n representing the order of K_n, with vertex set $V = \{0, 1, \ldots, n-1\}$.

Output: A random spanning tree of K_n.

```
1 if n = 1 then
2     return K_1
3 P ← random permutation of V
4 T ← null tree
5 for i ← 1, 2, ..., n − 1 do
6     j ← random element from {0, 1, ..., i − 1}
7     add edge (P[j], P[i]) to T
8 return T
```

3.13. Algorithm 3.13 constructs a random spanning tree of the complete graph K_n on $n > 0$ vertices. Its runtime is dependent on efficient algorithms for obtaining a random permutation of a set of objects, and choosing a random element from a given set.

(a) Describe and analyze the runtime of a procedure to construct a random permutation of a set of nonnegative integers.

(b) Describe an algorithm for randomly choosing an element of a set of nonnegative integers. Analyze the runtime of this algorithm.

(c) Taking into consideration the previous two algorithms, what is the runtime of Algorithm 3.13?

3.14. We want to generate a random undirected, connected simple graph on n vertices and having m edges. Start by generating a random spanning tree T of K_n. Then add random edges to T until the requirements are satisfied.

(a) Present pseudocode to realize the above procedure. What is the worst-case runtime of your algorithm?

(b) Modify your algorithm to handle the case where $m < n - 1$. Why must $m \geq n - 1$?

(c) Modify your algorithm to handle the case where each edge has a weight within the closed interval $[\alpha, \beta]$.

3.15. Enumerate all the different binary trees on 5 vertices.

3.16. Algorithm 3.5 generates a random binary tree on $n > 0$ vertices. Modify this algorithm so that it generates a random k-ary tree of order $n > 0$, where $k \geq 3$.

3.17. Show by giving an example that the Morse code is not prefix-free.

3.18. Consider the alphabet $\mathcal{A} = \{a, b, c\}$ with corresponding probabilities (or weights) $p(a) = 0.5$, $p(b) = 0.3$, and $p(c) = 0.2$. Generate two different Huffman codes for $\mathcal{A}$ and illustrate the tree representations of those codes.

3.19. Find the Huffman code for the letters of the English alphabet weighted by the frequency of common American usage.[4]

3.20. Let $G = (V_1, E_2)$ be a graph and $T = (V_2, E_2)$ a spanning tree of G. Show that there is a one-to-one correspondence between fundamental cycles in G and edges not in T.

3.21. Let $G = (V, E)$ be the 3×3 grid graph and $T_1 = (V_1, E_1)$, $T_2 = (V_2, E_2)$ be spanning trees of G in Example 3.1. Find a fundamental cycle in G for T_1 that is not a fundamental cycle in G for T_2.

3.22. Usually there exist many spanning trees of a graph. Classify those graphs for which there is only one spanning tree. In other words, find necessary and sufficient conditions for a graph G such that if T is a spanning tree of G then T is unique.

[4] You can find this on the Internet or in the literature. Part of this exercise is finding this frequency distribution yourself.

3.23. Convert the function `graycodeword` into a pure Python function.

3.24. Example 3.13 verifies that for any positive integer $n > 1$, repeated iteration of the Euler phi function $\varphi(n)$ eventually produces 1. Show that this is the case or provide an explanation why it is in general false.

3.25. The Collatz conjecture [**?**] asserts that for any integer $n > 0$, repeated iteration of the function

$$T(n) = \begin{cases} \frac{3n+1}{2}, & \text{if } n \text{ is odd,} \\ \frac{n}{2}, & \text{if } n \text{ is even} \end{cases}$$

eventually produces the value 1. For example, repeated iteration of $T(n)$ starting from $n = 22$ results in the sequence

$$22, 11, 17, 26, 13, 20, 10, 5, 8, 4, 2, 1. \tag{3.4}$$

One way to think about the Collatz conjecture is to consider the digraph G produced by considering $(a_i, T(a_i))$ as a directed edge of G. Then the Collatz conjecture can be rephrased to say that there is some integer $k > 0$ such that $(a_k, T(a_k)) = (2, 1)$ is a directed edge of G. The graph obtained in this manner is called the *Collatz graph* of $T(n)$. Given a collection of positive integers $\alpha_1, \alpha_2, \ldots, \alpha_k$, let G_{α_i} be the Collatz graph of the function $T(\alpha_i)$ with initial iteration value α_i. Then the union of the G_{α_i} is the directed tree

$$\bigcup_i G_{\alpha_i}$$

rooted at 1, called the *Collatz tree* of $(\alpha_1, \alpha_2, \ldots, \alpha_k)$. Figure 3.30 shows such a tree for the collection of initial iteration values 1024, 336, 340, 320, 106, 104, and 96. See Lagarias [**?**, **?**] for a comprehensive survey of the Collatz conjecture.

(a) The *Collatz sequence* of a positive integer $n > 1$ is the integer sequence produced by repeated iteration of $T(n)$ with initial iteration value n. For example, the Collatz sequence of $n = 22$ is the sequence (3.4). Write a Sage function to produce the Collatz sequence of an integer $n > 1$.

(b) The *Collatz length* of $n > 1$ is the number of terms in the Collatz sequence of n, inclusive of the starting iteration value and the final integer 1. For instance, the Collatz length of 22 is 12, that of 106 is 11, and that of 51 is 18. Write a Sage function to compute the Collatz length of a positive integer $n > 1$. If $n > 1$ is a vertex in a Collatz tree, verify that the Collatz length of n is the distance $d(n, 1)$.

(c) Describe the Collatz graph produced by the function $T(n)$ with initial iteration value $n = 1$.

(d) Fix a positive integer $n > 1$ and let L_i be the Collatz length of the integer $1 \leq i \leq n$. Plot the pairs (i, L_i) on one set of axes.

3.26. The following result was first published in Wiener [**?**]. Let $T = (V, E)$ be a tree of order $n > 0$. For each edge $e \in E$, let $n_1(e)$ and $n_2(e) = n - n_1(e)$ be the orders of

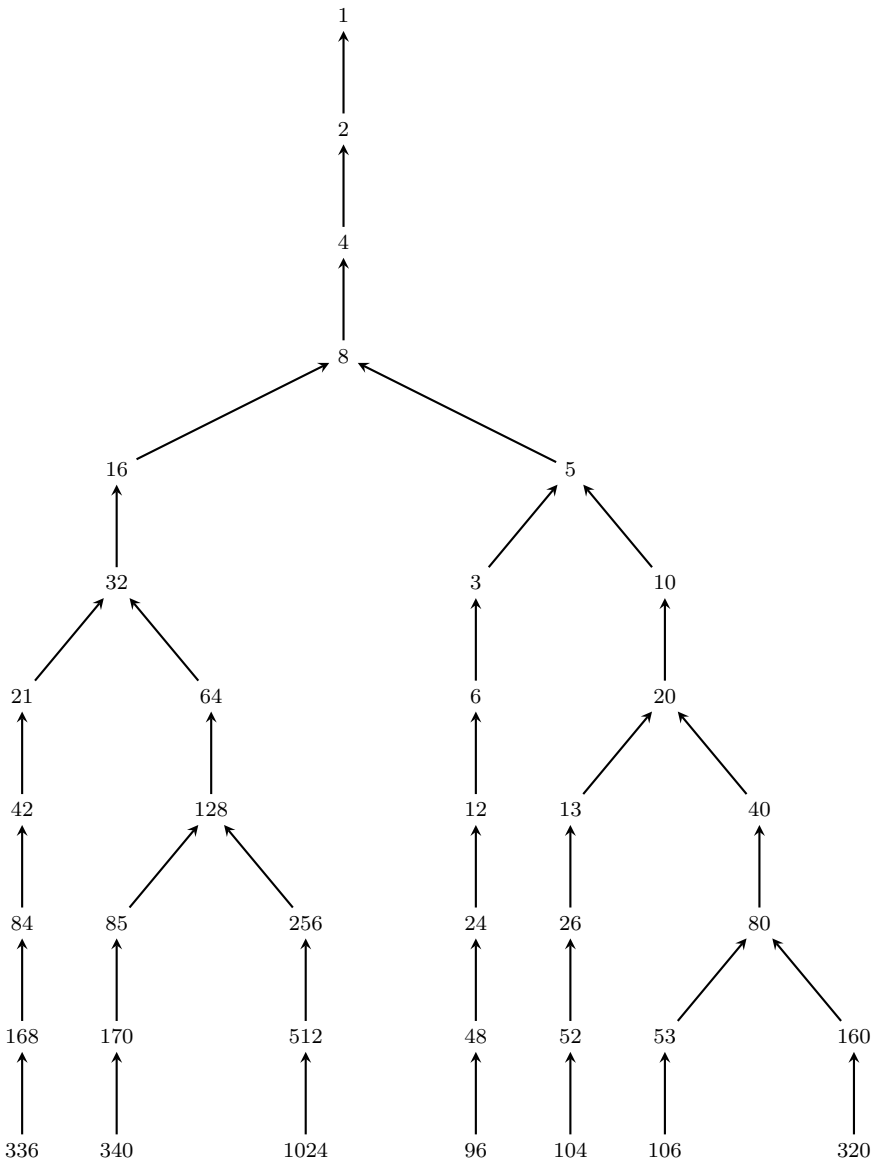


Figure 3.30: The union of Collatz graphs is a tree.

the two components of the edge-deletion subgraph $T - e$. Show that the Wiener number of T is

$$W(T) = \sum_{e \in E} n_1(e) \cdot n_2(e).$$

3.27. The following result [?] was independently discovered in the late 1980s by Merris and McKay, and is known as the Merris-McKay theorem. Let T be a tree of order n and let $\mathcal{L}$ be its Laplacian matrix having eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$. Show that the Wiener number of T is

$$W(T) = n \sum_{i=1}^{n-1} \frac{1}{\lambda_i}.$$

3.28. For each of the algorithms below: (i) justify whether or not it can be applied to multigraphs or multidigraphs; (ii) if not, modify the algorithm so that it is applicable to multigraphs or multidigraphs.

(a) Randomized spanning tree construction Algorithm 3.1.

(b) Kruskal's Algorithm 3.2.

(c) Prim's Algorithm 3.3.

(d) Borůvka's Algorithm 3.4.

3.29. Section 3.6 provides iterative algorithms for the following tree traversal techniques:

(a) Level-order traversal: Algorithm 3.8.

(b) Pre-order traversal: Algorithm 3.9.

(c) Post-order traversal: Algorithm 3.10.

(d) Bottom-up traversal: Algorithm 3.11.

(e) In-order traversal: Algorithm 3.12.

Rewrite each of the above as recursive algorithms.

3.30. In cryptography, the Merkle signature scheme [?] was introduced in 1987 as an alternative to traditional digital signature schemes such as the Digital Signature Algorithm or RSA. Buchmann et al. [?] and Szydlo [?] provide efficient algorithms for speeding up the Merkle signature scheme. Investigate this scheme and how it uses binary trees to generate digital signatures.

3.31. Consider the finite alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_r\}$. If C is a subset of $\mathcal{A}^*$, then we say that C is an r-ary code and call r the radix of the code. McMillan's theorem [?], first published in 1956, relates codeword lengths to unique decipherability. In particular, let $C = \{c_1, c_2, \ldots, c_n\}$ be an r-ary code where each c_i has length ℓ_i. If C is uniquely decipherable, McMillan's theorem states that the codeword lengths ℓ_i must satisfy Kraft's inequality

$$\sum_{i=1}^{n} \frac{1}{r^{\ell_i}} \leq 1.$$

Prove McMillan's theorem.

3.32. A code $C = \{c_1, c_2, \ldots, c_n\}$ is said to be *instantaneous* if each codeword c_i can be interpreted as soon as it is received. For example, given the the code {`01`, `010`} and the string `01010`, upon receiving the first `0` we are unable to decide whether that element belong to `01` or `010`. However, the code {`1`, `01`} is instantaneous because given the string `1101` and the first `1`, we can interpret the latter as the codeword `1`. Prove that a code is instantaneous if and only if it is prefix-free.

3.33. Kraft's inequality and the accompanying Kraft's theorem were first published [**?**] in 1949 in the Master's thesis of Leon Gordon Kraft. Kraft's theorem relates the inequality to instantaneous codes. Let $C = \{c_1, c_2, \ldots, c_n\}$ be an r-ary code where each codeword c_i has length ℓ_i. Kraft's theorem states that C is an instantaneous code if and only if the codeword lengths satisfy

$$\sum_{i=1}^{n} \frac{1}{r^{\ell_i}} \leq 1.$$

Prove Kraft's theorem.

3.34. Let T be a nontrivial tree and let n_i count the number of vertices of T that have degree i. Show that T has $2 + \sum_{i=3}^{\infty}(i-2)n_i$ leaves.

3.35. If a forest F has k trees totalling n vertices altogether, how many edges does F contain?

3.36. The Lucas number L_n, named after Édouard Lucas, has the following recursive definition:

$$L_n = \begin{cases} 2, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ L_{n-1} + L_{n-2}, & \text{if } n > 1. \end{cases}$$

(a) If $\varphi = (1 + \sqrt{5})/2$ is the golden ratio, show that

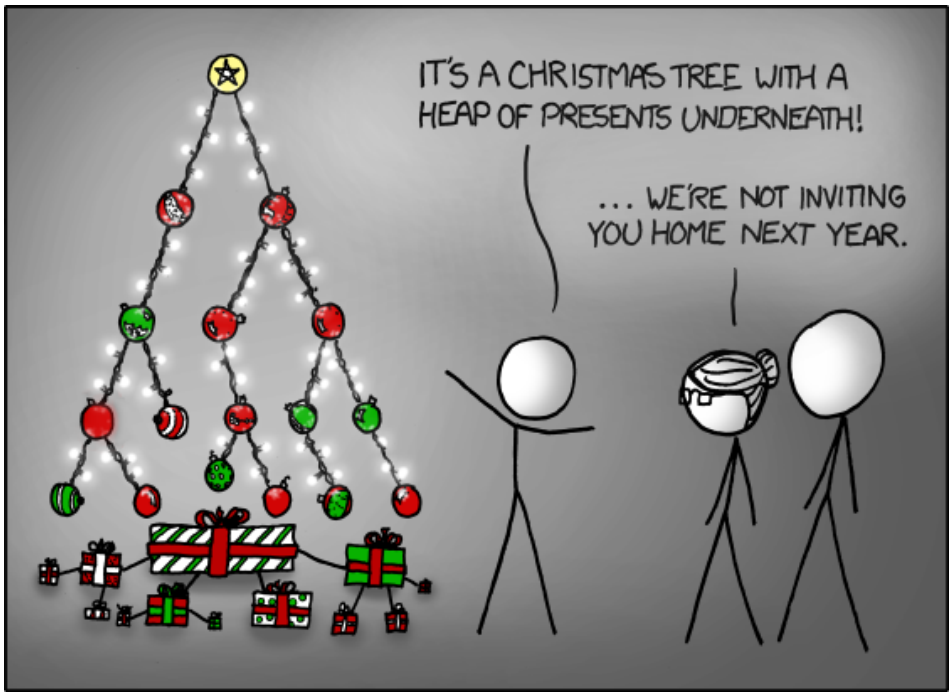
$$L_n = \varphi^n + (-\varphi)^{-n}.$$

(b) Let $\tau(W_n)$ be the number of spanning trees of the wheel graph. Benjamin and Yerger [**?**] provide a combinatorial proof that $\tau(W_n) = L_{2n} - 2$. Present the Benjamin-Yerger combinatorial proof.

(c) Let G be the Dodecahedral graph, implemented in Sage as `G = graphs.DodecahedralGraph(`. Does its cutset matrix satisfy the undirected analog of Theorem 3.19?

Chapter 4

Tree data structures



— Randall Munroe, xkcd, http://xkcd.com/835/

In Chapters 2 and 3, we discussed various algorithms that rely on priority queues as one of their fundamental data structures. Such algorithms include Dijkstra's algorithm, Prim's algorithm, and the algorithm for constructing Huffman trees. The runtime of any algorithm that uses priority queues crucially depends on an efficient implementation of the priority queue data structure. This chapter discusses the general priority queue data structure and various efficient implementations based on trees. Section 4.1 provides some theoretical underpinning of priority queues and considers a simple implementation of priority queues as sorted lists. Section 4.2 discusses how to use binary trees to realize an efficient implementation of priority queues called a binary heap. Although very useful in practice, binary heaps do not lend themselves to being merged in an efficient manner, a setback rectified in section 4.3 by a priority queue implementation called binomial heaps. As a further application of binary trees, section 4.4 discusses binary search trees as a general data structure for managing data in a sorted order.

4.1 Priority queues

A *priority queue* is essentially a queue data structure with various accompanying rules regarding how to access and manage elements of the queue. Recall from section 2.2.1 that an ordinary queue Q has the following basic accompanying functions for accessing and managing its elements:

- dequeue(Q) — Remove the front of Q.
- enqueue(Q, e) — Append the element e to the end of Q.

If Q is now a priority queue, each element is associated with a key or priority $p \in X$ from a totally ordered set X. A binary relation denoted by an infix operator, say "$\leq$", is defined on all elements of X such that the following properties hold for all $a, b, c \in X$:

- Totality: We have $a \leq b$ or $b \leq a$.
- Antisymmetry: If $a \leq b$ and $b \leq a$, then $a = b$.
- Transitivity: If $a \leq b$ and $b \leq c$, then $a \leq c$.

If the above three properties hold for the relation "$\leq$", then we say that "$\leq$" is a *total order* on X and that X is a *totally ordered set*. In all, if the key of each element of Q belongs to the same totally ordered set X, we use the total order defined on X to compare the keys of the queue elements. For example, the set $\mathbf{Z}$ of integers is totally ordered by the "less than or equal to" relation. If the key of each $e \in Q$ is an element of $\mathbf{Z}$, we use the latter relation to compare the keys of elements of Q. In the case of an ordinary queue, the key of each queue element is its position index.

To extract from a priority queue Q an element of lowest priority, we need to define the notion of smallest priority or key. Let p_i be the priority or key assigned to element e_i of Q. Then $p_{\min}$ is the lowest key if $p_{\min} \leq p$ for any element key p. The element with corresponding key $p_{\min}$ is the minimum priority element. Based upon the notion of key comparison, we define two operations on a priority queue:

- insert(Q, e, p) — Insert into Q the element e with key p.
- extractMin(Q) — Extract from Q an element having the smallest priority.

An immediate application of priority queues is sorting a finite sequence of items. Suppose L is a finite list of $n > 0$ items on which a total order is defined. Let Q be an empty priority queue. In the first phase of the priority queue sorting algorithm, we extract each element $e \in L$ from L and insert e into Q with key e itself. In other words, each element e is its own key. This first phase of the sorting algorithm requires n element extractions from L and n element insertions into Q. The second phase of the algorithm involves extracting elements from Q via the extractMin operation. Queue elements are extracted via extractMin and inserted back into L in the order in which they are extracted from Q. Algorithm 4.1 presents pseudocode of our discussion. The runtime of Algorithm 4.1 depends on how the priority queue Q is implemented.

Algorithm 4.1: Sorting a sequence via priority queue.

Input: A finite list L of $n > 0$ elements on which a total order is defined.
Output: The same list L sorted by the total order relation defined on its elements.

```
1 Q ← [ ]
2 for i ← 1, 2, ..., n do
3     e ← dequeue(L)
4     insert(Q, e, e)
5 for i ← 1, 2, ..., n do
6     e ← extractMin(Q)
7     enqueue(L, e)
```

4.1.1 Sequence implementation

A simple way to implement a priority queue is to maintain a sorted sequence. Let $e_0, e_1, \ldots, e_n$ be a sequence of $n+1$ elements with corresponding keys $\kappa_0, \kappa_1, \ldots, \kappa_n$ and suppose that the κ_i all belong to the same totally ordered set X having total order $\leq$. Using the total order, we assume that the κ_i are sorted as

$$\kappa_0 \leq \kappa_1 \leq \cdots \leq \kappa_n$$

and $e_i \leq e_j$ if and only if $\kappa_i \leq \kappa_j$. Then we consider the queue $Q = [e_0, e_1, \ldots, e_n]$ as a priority queue in which the head is always the minimum element and the tail is always the maximum element. Extracting the minimum element is simply a dequeue operation that can be accomplished in constant time $O(1)$. However, inserting a new element into Q takes linear time.

Let e be an element with corresponding key $\kappa \in X$. Inserting e into Q requires that we maintain elements of Q sorted according to the total order $\leq$. If Q is empty, we simply enqueue e into Q. Suppose now that Q is a nonempty priority queue. If $\kappa \leq \kappa_0$, then e becomes the new head of Q. If $\kappa_n \leq \kappa$, then e becomes the new tail of Q. Inserting a new head or tail into Q each requires constant time $O(1)$. However, if $\kappa_1 \leq \kappa \leq \kappa_{n-1}$ then we need to traverse Q starting from e_1, searching for a position at which to insert e. Let e_i be the queue element at position i within Q. If $\kappa \leq \kappa_i$ then we insert e into Q at position i, thus moving e_i to position $i+1$. Otherwise we next consider e_{i+1} and repeat the above comparison process. By hypothesis, $\kappa_1 \leq \kappa \leq \kappa_{n-1}$ and therefore inserting e into Q takes a worst-case runtime of $O(n)$.

4.2 Binary heaps

A sequence implementation of priority queues has the advantage of being simple to understand. Inserting an element into a sequence-based priority queue requires linear time, which can quickly become infeasible for queues containing hundreds of thousands or even millions of elements. Can we do any better? Rather than using a sorted sequence, we can use a binary tree to realize an implementation of priority queues that is much more efficient than a sequence-based implementation. In particular, we use a data structure called a *binary heap*, which allows for element insertion in logarithmic time.

In [**?**], Williams introduced the heapsort algorithm and described how to implement a priority queue using a binary heap. A basic idea is to consider queue elements as internal

vertices in a binary tree T, with external vertices or leaves being "place-holders". The tree T satisfies two further properties:

1. A relational property specifying the relative ordering and placement of queue elements.

2. A structural property that specifies the structure of T.

The relational property of T can be expressed as follows:

Definition 4.1. ***Heap-order property.*** *Let T be a binary tree and let v be a vertex of T other than the root. If p is the parent of v and these vertices have corresponding keys κ_p and κ_v, respectively, then $\kappa_p \leq \kappa_v$.*

The heap-order property is defined in terms of the total order used to compare the keys of the internal vertices. Taking the total order to be the ordinary "less than or equal to" relation, it follows from the heap-order property that the root of T is always the vertex with a minimum key. Similarly, if the total order is the usual "greater than or equal to" relation, then the root of T is always the vertex with a maximum key. In general, if $\leq$ is a total order defined on the keys of T and u and v are vertices of T, we say that u is less than or equal to v if and only if $u \leq v$. Furthermore, u is said to be a minimum vertex of T if and only if $u \leq v$ for all vertices of T. From our discussion above, the root is always a minimum vertex of T and is said to be "at the top of the heap", from which we derive the name "heap" for this data structure.

Another consequence of the heap-order property becomes apparent when we trace out a path from the root of T to any internal vertex. Let r be the root of T and let v be any internal vertex of T. If $r, v_0, v_1, \ldots, v_n, v$ is an r-v path with corresponding keys

$$\kappa_r, \kappa_{v_0}, \kappa_{v_1}, \ldots, \kappa_{v_n}, \kappa_v$$

then we have

$$\kappa_r \leq \kappa_{v_0} \leq \kappa_{v_1} \leq \cdots \leq \kappa_{v_n} \leq \kappa_v.$$

In other words, the keys encountered on the path from r to v are arranged in nondecreasing order.

The structural property of T is used to enforce that T be of as small a height as possible. Before stating the structural property, we first define the level of a binary tree. Recall that the depth of a vertex in T is its distance from the root. Level i of a binary tree T refers to all vertices of T that have the same depth i. We are now ready to state the heap-structure property.

Definition 4.2. ***Heap-structure property.*** *Let T be a binary tree with height h. Then T satisfies the heap-structure property if T is nearly a complete binary tree. That is, level $0 \leq i \leq h-1$ has 2^i vertices, whereas level h has $\leq 2^h$ vertices. The vertices at level h are filled from left to right.*

If a binary tree T satisfies both the heap-order and heap-structure properties, then T is referred to as a binary heap. By insisting that T satisfy the heap-order property, we are able to determine the minimum vertex of T in constant time $O(1)$. Requiring that T also satisfy the heap-structure property allows us to determine the last vertex of T. The last vertex of T is identified as the right-most internal vertex of T having the greatest depth. Figure 4.1 illustrates various examples of binary heaps. The heap-structure property together with Theorem 3.21 result in the following corollary on the height of a binary heap.

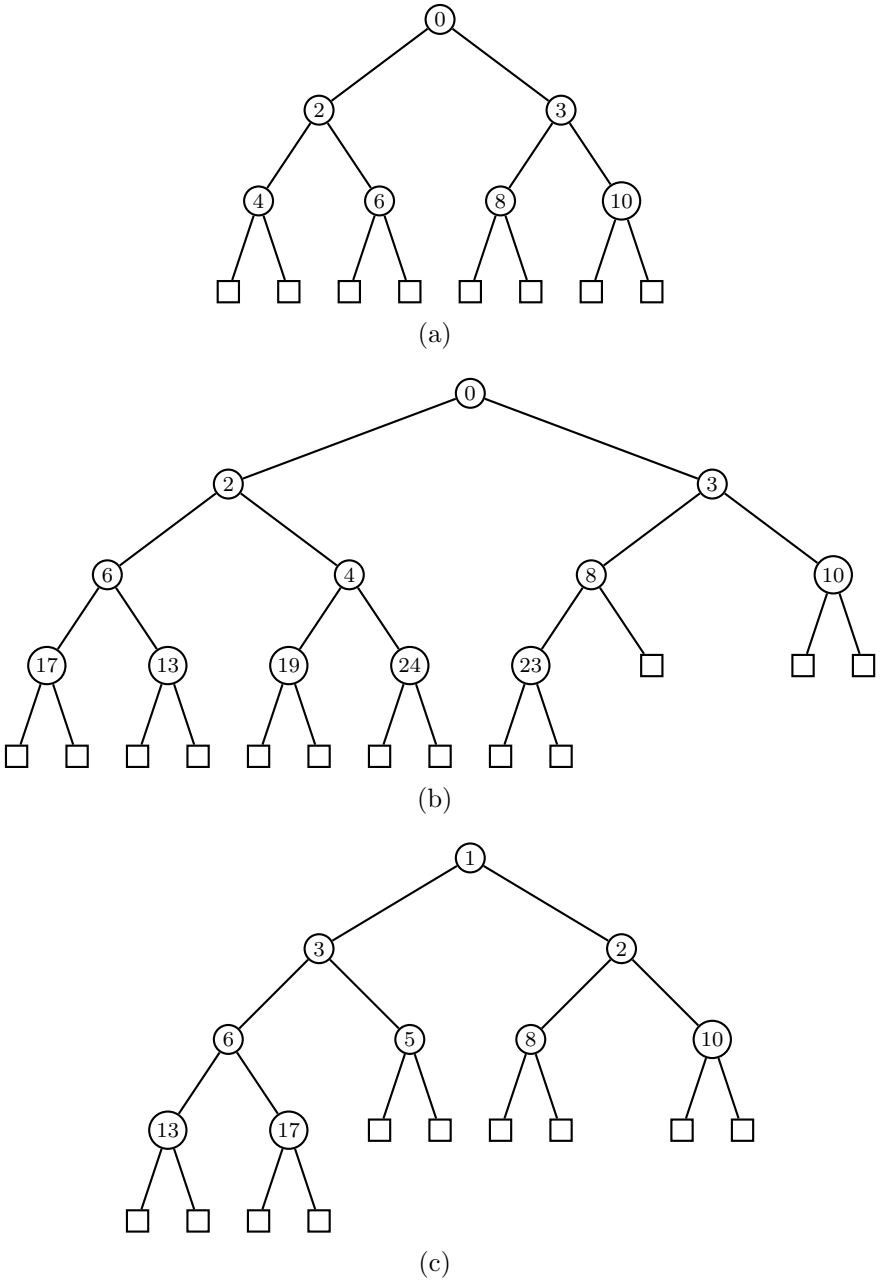


Figure 4.1: Examples of binary heaps with integer keys.

Corollary 4.3. *A binary heap T with n internal vertices has height*

$$h = \lceil \lg(n+1) \rceil.$$

Proof. Level $h-1$ has at least one internal vertex. Apply Theorem 3.21 to see that T has at least

$$2^{h-2+1} - 1 + 1 = 2^{h-1}$$

internal vertices. On the other hand, level $h-1$ has at most 2^{h-1} internal vertices. Another application of Theorem 3.21 shows that T has at most

$$2^{h-1+1} - 1 = 2^h - 1$$

internal vertices. Thus n is bounded by

$$2^{h-1} \leq n \leq 2^h - 1.$$

Taking logarithms of each side in the latter bound results in

$$\lg(n+1) \leq h \leq \lg n + 1$$

and the corollary follows. ■

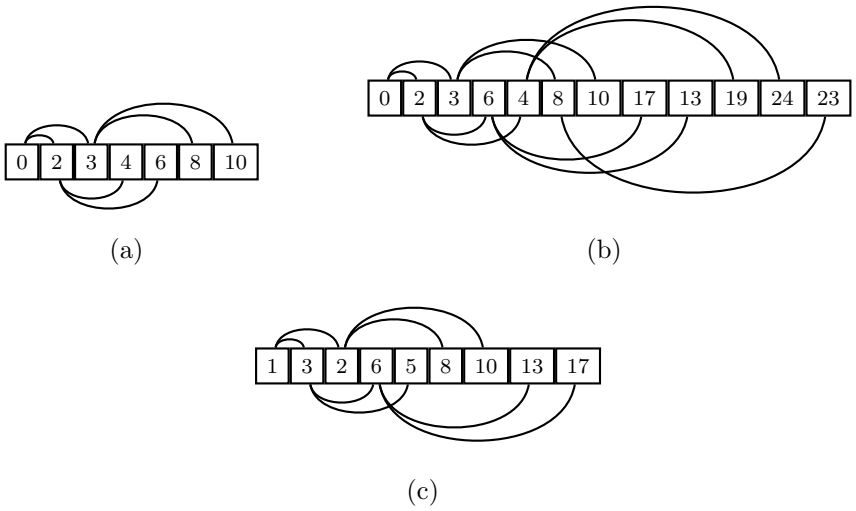


Figure 4.2: Sequence representations of various binary heaps.

4.2.1 Sequence representation

Any binary heap can be represented as a binary tree. Each vertex in the tree must know about its parent and its two children. However, a more common approach is to represent a binary heap as a sequence such as a list, array, or vector. Let T be a binary heap consisting of n internal vertices and let L be a list of n elements. The root vertex is represented as the list element $L[0]$. For each index i, the children of $L[i]$ are $L[2i+1]$ and $L[2i+2]$ and the parent of $L[i]$ is

$$L\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right].$$

With a sequence representation of a binary heap, each vertex needs not know about its parent and children. Such information can be obtained via simple arithmetic on sequence indices. For example, the binary heaps in Figure 4.1 can be represented as the corresponding lists in Figure 4.2. Note that it is not necessary to store the leaves of T in the sequence representation.

4.2.2 Insertion and sift-up

We now consider the problem of inserting a vertex v into a binary heap T. If T is empty, inserting a vertex simply involves the creation of a new internal vertex. We let that new internal vertex be v and let its two children be leaves. The resulting binary heap augmented with v has exactly one internal vertex and satisfies both the heap-order and heap-structure properties, as shown in Figure 4.3. In other words, any binary heap with one internal vertex trivially satisfies the heap-order property.

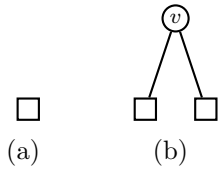


Figure 4.3: Inserting a vertex into an empty binary heap.

Let T now be a nonempty binary heap, i.e. T has at least one internal vertex, and suppose we want to insert into T an internal vertex v. We must identify the correct leaf of T at which to insert v. If the n internal vertices of T are $r = v_0, v_1, \ldots, v_{n-1}$, then by the sequence representation of T we can identify the last internal vertex v_{n-1} in constant time. The correct leaf at which to insert v is the sequence element immediately following v_{n-1}, i.e. the element at position n in the sequence representation of T. We replace with v the leaf at position n in the sequence so that v now becomes the last vertex of T.

The binary heap T augmented with the new last vertex v satisfies the heap-structure property, but may violate the heap-order property. To ensure that T satisfies the heap-order property, we perform an operation on T called *sift-up* that involves possibly moving v up through various levels of T. Let κ_v be the key of v and let $\kappa_{p(v)}$ be the key of v's parent. If the relation $\kappa_{p(v)} \leq \kappa_v$ holds, then T satisfies the heap-order property. Otherwise we swap v with its parent, effectively moving v up one level to be at the position previously occupied by its parent. The parent of v is moved down one level and now occupies the position where v was previously. With v in its new position, we perform the same key comparison process with v's new parent. The key comparison and swapping continue until the heap-order property holds for T. In the worst case, v would become the new root of T after undergoing a number of swaps that is proportional to the height of T. Therefore, inserting a new internal vertex into T can be achieved in time $O(\lg n)$. Figure 4.4 illustrates the insertion of a new internal vertex into a nonempty binary heap and the resulting sift-up operation to maintain the heap-order property. Algorithm 4.2 presents pseudocode of our discussion for inserting a new internal vertex into a nonempty binary heap. The pseudocode is adapted from Howard [**?**], which provides a C implementation of binary heaps.

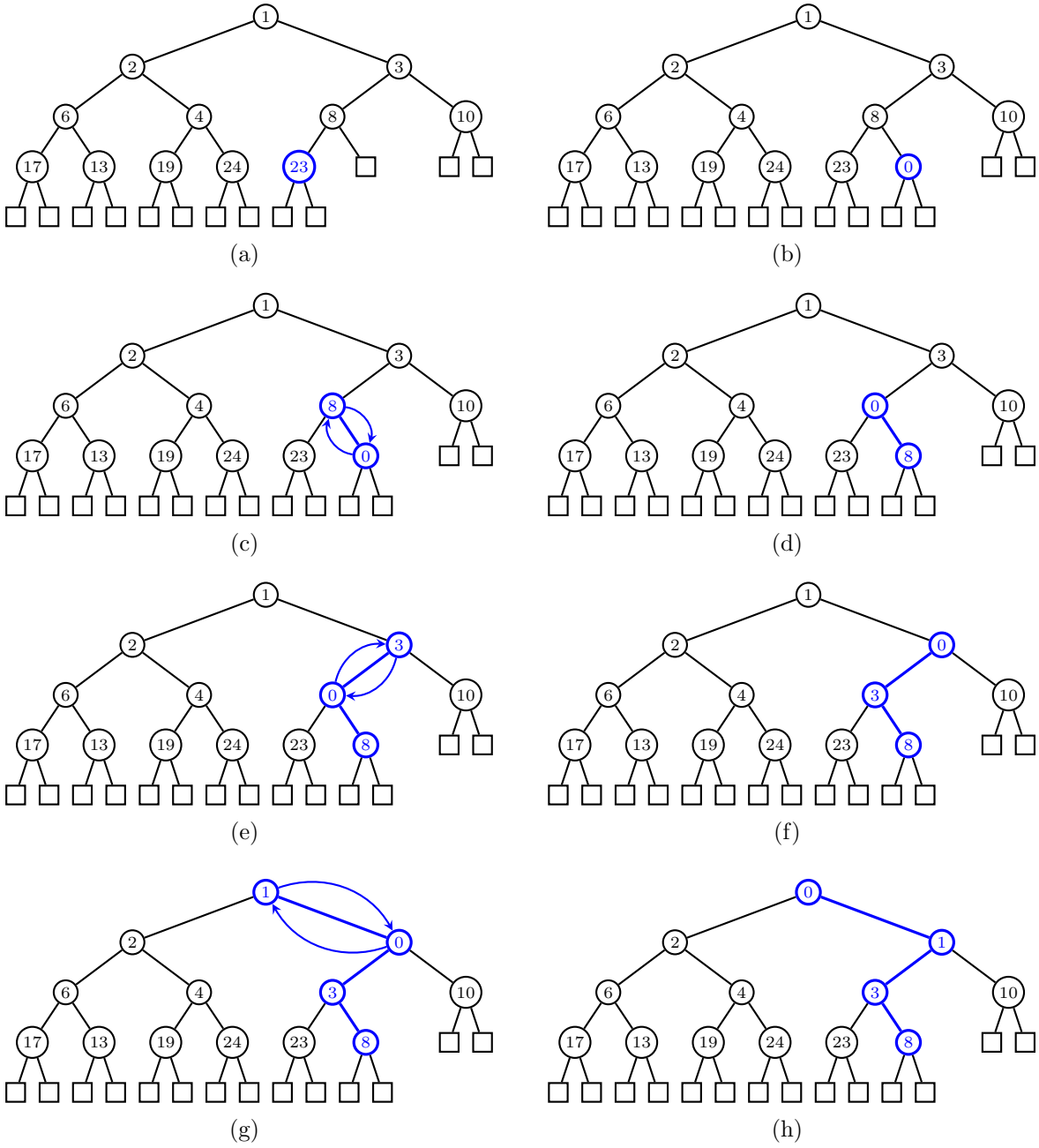


Figure 4.4: Insert and sift-up in a binary heap.

Algorithm 4.2: Inserting a new internal vertex into a binary heap.

Input: A nonempty binary heap T, in sequence representation, having n internal vertices. An element v that is to be inserted as a new internal vertex of T.

Output: The binary heap T augmented with the new internal vertex v.

```
1  i ← n
2  while i > 0 do
3      p ← ⌊(i − 1)/2⌋
4      if κ_{T[p]} ≤ κ_v then
5          exit the loop
6      else
7          T[i] ← T[p]
8          i ← p
9  T[i] ← v
10 return T
```

4.2.3 Deletion and sift-down

The process for deleting the minimum vertex of a binary heap bears some resemblance to that of inserting a new internal vertex into the heap. Having removed the minimum vertex, we must then ensure that the resulting binary heap satisfies the heap-order property. Let T be a binary heap. By the heap-order property, the root of T has a key that is minimum among all keys of internal vertices in T. If the root r of T is the only internal vertex of T, i.e. T is the trivial binary heap, we simply remove r and T now becomes the empty binary heap or the trivial tree, for which the heap-order property vacuously holds. Figure 4.5 illustrates the case of removing the root of a binary heap having one internal vertex.

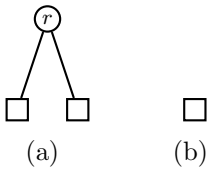


Figure 4.5: Deleting the root of a trivial binary heap.

We now turn to the case where T has $n > 1$ internal vertices. Let r be the root of T and let v be the last internal vertex of T. Deleting r would disconnect T. So we instead replace the key and information at r with the key and other relevant information pertaining to v. The root r now has the key of the last internal vertex, and v becomes a leaf.

At this point, T satisfies the heap-structure property but may violate the heap-order property. To restore the heap-order property, we perform an operation on T called *sift-down* that may possibly move r down through various levels of T. Let $c(r)$ be the child of r with key that is minimum among all the children of r, and let κ_r and $\kappa_{c(r)}$ be the keys of r and $c(r)$, respectively. If $\kappa_r \leq \kappa_{c(r)}$, then the heap-order property is satisfied. Otherwise we swap r with $c(r)$, moving r down one level to the position previously occupied by $c(r)$. Furthermore, $c(r)$ is moved up one level to the position previously occupied by r. With r in its new position, we perform the same key comparison process with a child of

r that has minimum key among all of r's children. The key comparison and swapping continue until the heap-order property holds for T. In the worst case, r would percolate all the way down to the level that is immediately above the last level after undergoing a number of swaps that is proportional to the height of T. Therefore, deleting the minimum vertex of T can be achieved in time $O(\lg n)$. Figure 4.6 illustrates the deletion of the minimum vertex of a binary heap with at least two internal vertices and the resulting sift-down process that percolates vertices down through various levels of the heap in order to maintain the heap-order property. Algorithm 4.3 summarizes our discussion of the process for extracting the minimum vertex of T while also ensuring that T satisfies the heap-order property. The pseudocode is adapted from the C implementation of binary heaps in Howard [**?**]. With some minor changes, Algorithm 4.3 can be used to change the key of the root vertex and maintain the heap-order property for the resulting binary tree.

Algorithm 4.3: Extract the minimum vertex of a binary heap.

Input: A binary heap T, given in sequence representation, having $n > 1$ internal vertices.

Output: Extract the minimum vertex of T. With one vertex removed, T must satisfy the heap-order property.

```
1  root ← T[0]
2  n ← n − 1
3  v ← T[n]
4  i ← 0
5  j ← 0
6  while True do
7      left ← 2i + 1
8      right ← 2i + 2
9      if left < n and κ_{T[left]} ≤ κ_v then
10         if right < n and κ_{T[right]} ≤ κ_{T[left]} then
11             j ← right
12         else
13             j ← left
14     else if right < n and κ_{T[right]} ≤ κ_v then
15         j ← right
16     else
17         T[i] ← v
18         exit the loop
19     T[i] ← T[j]
20     i ← j
21 return root
```

4.2.4 Constructing a binary heap

Given a collection of n vertices $v_0, v_1, \ldots, v_{n-1}$ with corresponding keys $\kappa_0, \kappa_1, \ldots, \kappa_{n-1}$, we want to construct a binary heap containing exactly those vertices. A basic approach is to start with a trivial tree and build up a binary heap via successive insertions. As each

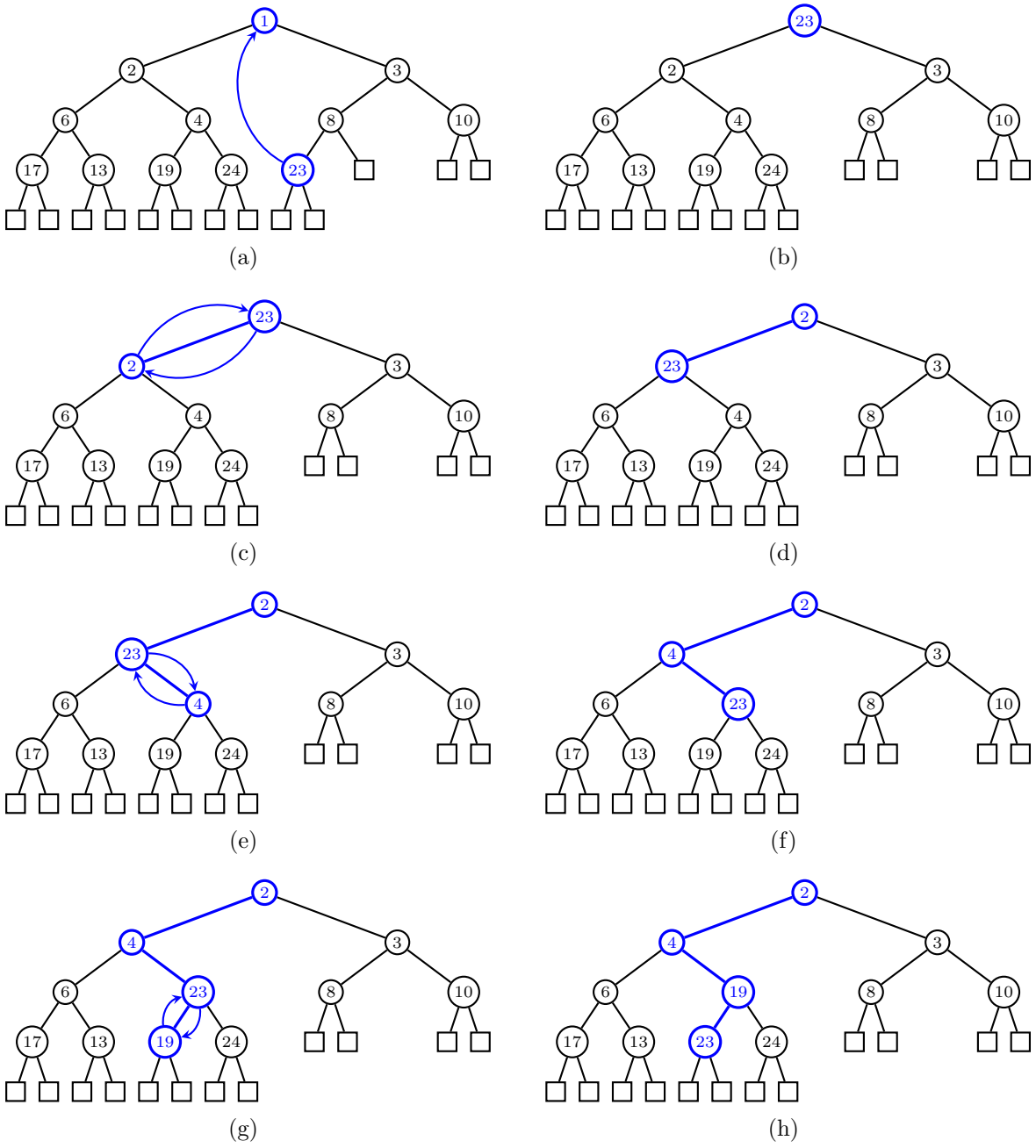


Figure 4.6: Delete and sift-down in a binary heap.

insertion requires $O(\lg n)$ time, the method of binary heap construction via successive insertion of each of the n vertices requires $O(n \cdot \lg n)$ time. It turns out we could do a bit better and achieve the same result in linear time.

Algorithm 4.4: Heapify a binary tree.

Input: A binary tree T, given in sequence representation, having $n > 1$ internal vertices.

Output: The binary tree T heapified so that it satisfies the heap-order property.

```
1  for i ← ⌊n/2⌋ − 1, ..., 0 do
2      v ← T[i]
3      j ← 0
4      while True do
5          left ← 2i + 1
6          right ← 2i + 2
7          if left < n and κ_{T[left]} ≤ κ_v then
8              if right < n and κ_{T[right]} ≤ κ_{T[left]} then
9                  j ← right
10             else
11                 j ← left
12         else if right < n and κ_{T[right]} ≤ κ_v then
13             j ← right
14         else
15             T[i] ← v
16             exit the while loop
17         T[i] ← T[j]
18         i ← j
19 return T
```

A better approach starts by letting $v_0, v_1, \ldots, v_{n-1}$ be the internal vertices of a binary tree T. The tree T need not satisfy the heap-order property, but it must satisfy the heap-structure property. Suppose T is given in sequence representation so that we have the correspondence $v_i = T[i]$ and the last internal vertex of T has index $n-1$. The parent of $T[n-1]$ has index

$$j = \left\lfloor \frac{n-1}{2} \right\rfloor .$$

Any vertex of T with sequence index beyond $n-1$ is a leaf. In other words, if an internal vertex has index $> j$, then the children of that vertex are leaves and have indices $\geq n$. Thus any internal vertex with index $\geq \lfloor n/2 \rfloor$ has leaves for its children. Conclude that internal vertices with indices

$$\left\lfloor \frac{n}{2} \right\rfloor , \left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \ldots, n-1 \tag{4.1}$$

have only leaves for their children.

Our next task is to ensure that the heap-order property holds for T. If v is an internal vertex with index in (4.1), then the subtree rooted at v is trivially a binary heap. Consider the indices from $\lfloor n/2 \rfloor - 1$ all the way down to 0 and let i be such an index, i.e. let $0 \leq i \leq \lfloor n/2 \rfloor - 1$. We heapify the subtree of T rooted at $T[i]$, effectively

performing a sift-down on this subtree. Once we have heapified all subtrees rooted at $T[i]$ for $0 \leq i \leq \lfloor n/2 \rfloor - 1$, the resulting tree T is a binary heap. Our discussion is summarized in Algorithm 4.4.

Earlier in this section, we claimed that Algorithm 4.4 can be used to construct a binary heap in worst-case linear time. To prove this, let T be a binary tree satisfying the heap-structure property and having n internal vertices. By Corollary 4.3, T has height $h = \lceil \lg(n+1) \rceil$. We perform a sift-down for at most 2^i vertices of depth i, where each sift-down for a subtree rooted at a vertex of depth i takes $O(h-i)$ time. Then the total time for Algorithm 4.4 is

$$\begin{aligned} O\left(\sum_{0 \leq i < h} 2^i (h-i)\right) &= O\left(2^h \sum_{0 \leq i < h} \frac{2-i}{2^{h-i}}\right) \\ &= O\left(2^h \sum_{k>0} \frac{k}{2^k}\right) \\ &= O\left(2^{h+1}\right) \\ &= O(n) \end{aligned}$$

where we used the closed form $\sum_{k>0} k/2^k = 2$ for a geometric series and Theorem 3.21.

4.3 Binomial heaps

We are given two binary heaps T_1 and T_2 and we want to merge them into a single heap. We could start by choosing to insert each element of T_2 into T_1, successively extracting the minimum element from T_2 and insert that minimum element into T_1. If T_1 and T_2 have m and n elements, respectively, we would perform n extractions from T_2 totalling

$$O\left(\sum_{0 < k \leq n} \lg k\right)$$

time and inserting all of the extracted elements from T_2 into T_1 requires a total runtime of

$$O\left(\sum_{n \leq k < n+m} \lg k\right). \tag{4.2}$$

We approximate the addition of the two sums by

$$\int_0^{n+m} \lg k \, dk = \frac{k \ln k - k}{\ln 2} + C \Bigg|_{k=0}^{k=n+m}$$

for some constant C. The above method of successive extraction and insertion therefore has a total runtime of

$$O\left(\frac{(n+m)\ln(n+m) - n - m}{\ln 2}\right)$$

for merging two binary heaps.

Alternatively, we could slightly improve the latter runtime for merging T_1 and T_2 by successively extracting the last internal vertex of T_2. The whole process of extracting

all elements from T_2 in this way takes $O(n)$ time and inserting each of the extracted elements into T_1 still requires the runtime in expression (4.2). We approximate the sum in (4.2) by

$$\int_{k=n}^{k=n+m} \lg k \, dk = \frac{k \ln k - k}{\ln 2} + C \Bigg|_{k=n}^{k=n+m}$$

for some constant C. Therefore the improved extraction and insertion method requires

$$O\left(\frac{(n+m)\ln(n+m) - n\ln n - m}{\ln 2} - n\right)$$

time in order to merge T_1 and T_2.

Can we improve on the latter runtime for merging two binary heaps? It turns out we can by using a type of mergeable heap called binomial heap that supports merging two heaps in logarithmic time.

4.3.1 Binomial trees

A binomial heap can be considered as a collection of binomial trees. The binomial tree of order k is denoted B_k and defined recursively as follows:

1. The binomial tree of order 0 is the trivial tree.

2. The binomial tree of order $k > 0$ is a rooted tree, where from left to right the children of the root of B_k are roots of $B_{k-1}, B_{k-2}, \ldots, B_0$.

Various examples of binomial trees are shown in Figure 4.7. The binomial tree B_k can also be defined as follows. Let T_1 and T_2 be two copies of B_{k-1} with root vertices r_1 and r_2, respectively. Then B_k is obtained by letting, say, r_1 be the left-most child of r_2. Lemma 4.4 lists various basic properties of binomial trees. Property (3) of Lemma 4.4 uses the binomial coefficient, from whence B_k derives its name.

Lemma 4.4. ***Basic properties of binomial trees.*** *Let B_k be a binomial tree of order $k \geq 0$. Then the following properties hold:*

1. *The order of B_k is 2^k.*

2. *The height of B_k is k.*

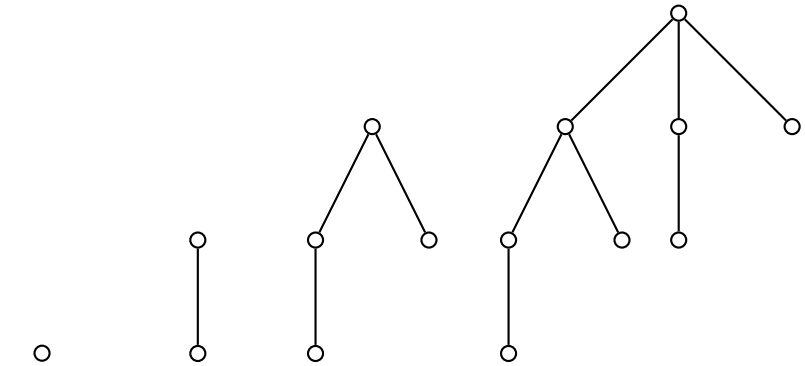
3. *For $0 \leq i \leq k$, we have $\binom{k}{i}$ vertices at depth i.*

4. *The root of B_k is the only vertex with maximum degree $\Delta(B_k) = k$. If the children of the root are numbered $k-1, k-2, \ldots, 0$ from left to right, then child i is the root of the subtree B_i.*

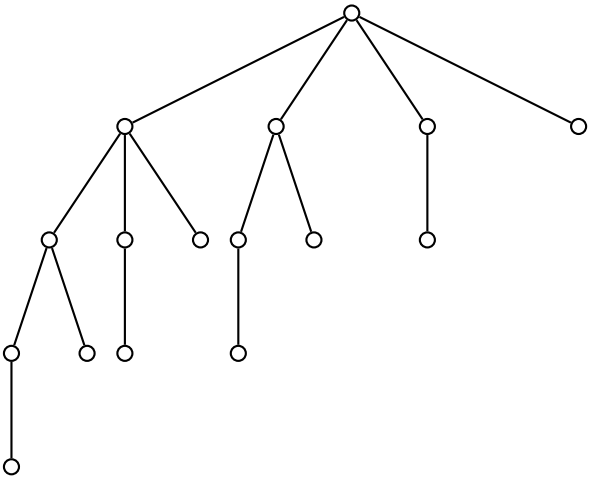
Proof. We use induction on k. The base case for each of the above properties is B_0, which trivially holds.

(1) By our inductive hypothesis, B_{k-1} has order 2^{k-1}. Since B_k is comprised of two copies of B_{k-1}, conclude that B_k has order

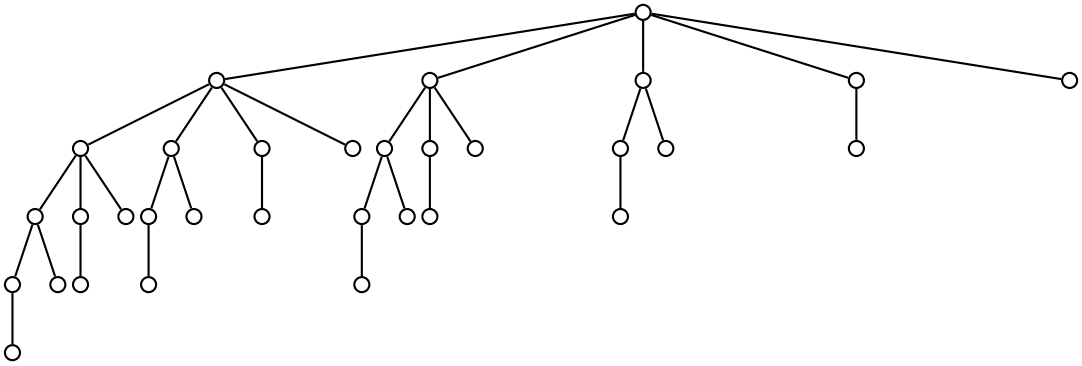
$$2^{k-1} + 2^{k-1} = 2^k.$$



(a) B_0 (b) B_1 (c) B_2 (d) B_3



(e) B_4



(f) B_5

Figure 4.7: Binomial trees B_k for $k = 0, 1, 2, 3, 4, 5$.

(2) The binomial tree B_k is comprised of two copies of B_{k-1}, the root of one copy being the left-most child of the root of the other copy. Then the height of B_k is one greater than the height of B_{k-1}. By our inductive hypothesis, B_{k-1} has height $k-1$ and therefore B_k has height $(k-1)+1=k$.

(3) Denote by $D(k,i)$ the number of vertices of depth i in B_k. As B_k is comprised of two copies of B_{k-1}, a vertex at depth i in B_{k-1} appears once in B_k at depth i and a second time at depth $i+1$. By our inductive hypothesis,

$$\begin{aligned} D(k,i) &= D(k-1,i) + D(k-1,i-1) \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} \\ &= \binom{k}{i} \end{aligned}$$

where we used Pascal's formula which states that

$$\binom{n+1}{r} = \binom{n}{r-1} + \binom{n}{r}$$

for any positive integers n and r with $r \leq n$.

(4) This property follows from the definition of B_k. ■

Corollary 4.5. *If a binomial tree has order $n \geq 0$, then the degree of any vertex i is bounded by* $\deg(i) \leq \lg n$.

Proof. Apply properties (1) and (4) of Lemma 4.4. ■

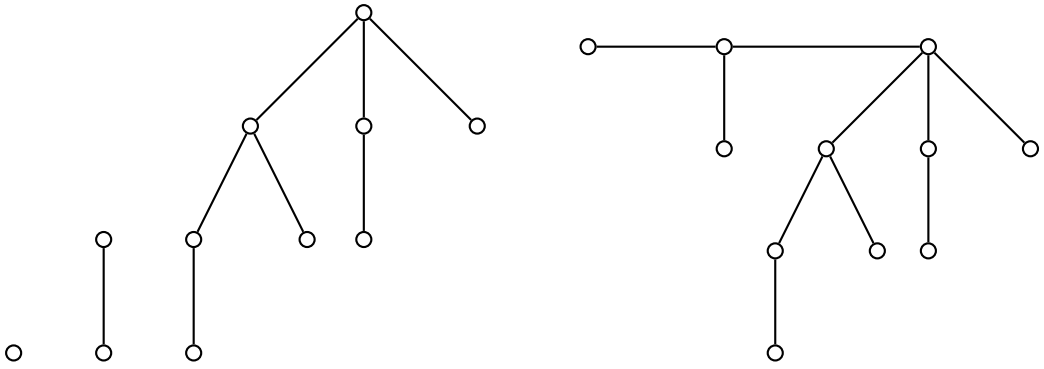
4.3.2 Binomial heaps

In 1978, Jean Vuillemin [**?**] introduced binomial heaps as a data structure for implementing priority queues. Mark R. Brown [**?**, **?**] subsequently extended Vuillemin's work, providing detailed analysis of binomial heaps and introducing an efficient implementation.

A binomial heap H can be considered as a collection of binomial trees. Each vertex in H has a corresponding key and all vertex keys of H belong to a totally ordered set having total order $\leq$. The heap also satisfies the following *binomial heap properties*:

- **Heap-order property.** Let B_k be a binomial tree in H. If v is a vertex of B_k other than the root and p is the parent of v and having corresponding keys κ_v and κ_p, respectively, then $\kappa_p \leq \kappa_v$.
- **Root-degree property.** For any integer $k \geq 0$, H contains at most one binomial tree whose root has degree k.

If H is comprised of the binomial trees $B_{k_0}, B_{k_1}, \ldots, B_{k_n}$ for nonnegative integers k_i, we can consider H as a forest made up of the trees B_{k_i}. We can also represent H as a tree in the following way. List the binomial trees of H as $B_{k_0}, B_{k_1}, \ldots, B_{k_n}$ in nondecreasing order of root degrees, i.e. the root of B_{k_i} has order less than or equal to the root of B_{k_j} if and only if $k_i \leq k_j$. The root of H is the root of B_{k_0} and the root of each B_{k_i} has for its child the root of $B_{k_{i+1}}$. Both the forest and tree representations are illustrated in Figure 4.8 for the binomial heap comprised of the binomial trees B_0, B_1, B_3.



(a) Binomial heap as a forest. (b) Binomial heap as a tree.

Figure 4.8: Forest and tree representations of a binomial heap.

The heap-order property for binomial heaps is analogous to the heap-order property for binary heaps. In the case of binomial heaps, the heap-order property implies that the root of a binomial tree has a key that is minimum among all vertices in that tree. However, the similarity more or less ends there. In a tree representation of a binomial heap, the root of the heap may not necessarily have the minimum key among all vertices of the heap.

The root-degree property can be used to derive an upper bound on the number of binomial trees in a binomial heap. If H is a binomial heap with n vertices, then H has at most $1 + \lfloor \lg n \rfloor$ binomial trees. To prove this result, note that (see Theorem 2.1 and Corollary 2.1.1 in [**?**, pp.40–42]) n can be uniquely written in binary representation as the polynomial

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_1 2^1 + a_0 2^0.$$

The binary representation of n requires $1 + \lfloor \lg n \rfloor$ bits, hence $n = \sum_{i=0}^{\lfloor \lg n \rfloor} a_i 2^i$. Apply property (1) of Lemma 4.4 to see that the binomial tree B_i is in H if and only if the i-th bit is $b_i = 1$. Conclude that H has at most $1 + \lfloor \lg n \rfloor$ binomial trees.

4.3.3 Construction and management

Let H be a binomial heap comprised of the binomial trees $B_{k_0}, B_{k_1}, \ldots, B_{k_n}$ where the root of B_{k_i} has order less than or equal to the root of B_{k_j} if and only if $k_i \leq k_j$. Denote by r_{k_i} the root of the binomial tree B_{k_i}. If v is a vertex of H, denote by child$[v]$ the left-most child of v and by sibling$[v]$ we mean the sibling immediately to the right of v. Furthermore, let parent$[v]$ be the parent of v and let degree$[v]$ denote the degree of v. If v has no children, we set child$[v]$ = `NULL`. If v is one of the roots r_{k_i}, we set parent$[v]$ = `NULL`. And if v is the right-most child of its parent, then we set sibling$[v]$ = `NULL`.

The roots $r_{k_0}, r_{k_1}, \ldots, r_{k_n}$ can be organized as a linked list, called a *root list*, with two functions for accessing the next root and the previous root. The root immediately following r_{k_i} is denoted next$[r_{k_i}]$ = sibling$[v] = r_{k_{i+1}}$ and the root immediately before r_{k_i} is written prev$[r_{k_i}] = r_{k_{i-1}}$. For r_{k_0} and r_{k_n}, we set next$[r_{k_n}]$ = sibling$[v]$ = `NULL` and prev$[r_{k_0}]$ = `NULL`. We also define the function head$[H]$ that simply returns r_{k_0} whenever H has at least one element, and head$[H]$ = `NULL` otherwise.

Minimum vertex

To find the minimum vertex, we find the minimum among $r_{k_0}, r_{k_1}, \ldots, r_{k_m}$ because by definition the root r_{k_i} is the minimum vertex of the binomial tree B_{k_i}. If H has n vertices, we need to check at most $1 + \lfloor \lg n \rfloor$ vertices to find the minimum vertex of H. Therefore determining the minimum vertex of H takes $O(\lg n)$ time. Algorithm 4.5 summarizes our discussion.

Algorithm 4.5: Determine the minimum vertex of a binomial heap.

Input: A binomial heap H of order $n > 0$.
Output: The minimum vertex of H.

```
1 u ← NULL
2 v ← head[H]
3 min ← ∞
4 while v ≠ NULL do
5     if κ_v < min then
6         min ← κ_v
7         u ← v
8     v ← sibling[v]
9 return u
```

Merging heaps

Recall that B_k is constructed by linking the root of one copy of B_{k-1} with the root of another copy of B_{k-1}. When merging two binomial heaps whose roots have the same degree, we need to repeatedly link the respective roots. The root linking procedure runs in constant time $O(1)$ and is rather straightforward, as presented in Algorithm 4.6.

Algorithm 4.6: Linking the roots of binomial heaps.

Input: Two copies of B_{k-1}, one rooted at u and the other at v.
Output: The respective roots of two copies of B_{k-1} linked, with one root becoming the parent of the other.

```
1 parent[u] ← v
2 sibling[u] ← child[v]
3 child[v] ← u
4 degree[v] ← degree[v] + 1
```

Besides linking the roots of two copies of B_{k-1}, we also need to merge the root lists of two binomial heaps H_1 and H_2. The resulting merged list is sorted in nondecreasing order of degree. Let L_1 be the root list of H_1 and let L_2 be the root list of H_2. First we create an empty list L. As the lists L_i are already sorted in nondecreasing order of vertex degree, we use merge sort to merge the L_i into a single sorted list. The whole procedure for merging the L_i takes linear time $O(n)$, where $n = |L_1| + |L_2| - 1$. Refer to Algorithm 4.7 for pseudocode of the procedure just described.

Having clarified the root linking and root lists merging procedures, we are now ready to describe a procedure for merging two nonempty binomial heaps H_1 and H_2 into a

Algorithm 4.7: Merging two root lists.

Input: Two root lists L_1 and L_2, each containing the roots of binomial trees in the binomial heaps H_1 and H_2, respectively. Each root list L_i is sorted in increasing order of vertex degree.

Output: A single list L that merges the root lists L_i and sorted in nondecreasing order of degree.

```
1  i ← 1
2  j ← 1
3  L ← [ ]
4  n ← |L_1| + |L_2| − 1
5  append(L_1, ∞)
6  append(L_2, ∞)
7  for k ← 0, 1, ..., n do
8      if deg(L_1[i]) ≤ deg(L_2[j]) then
9          append(L, L_1[i])
10         i ← i + 1
11     else
12         append(L, L_2[j])
13         j ← j + 1
14 return L
```

single binomial heap H. Initially there are at most two copies of B_0, one from each of the H_i. If two copies of B_0 are present, we let the root of one be the parent of the other as per Algorithm 4.6, producing B_1 as a result. From thereon, we generally have at most three copies of B_k for some integer $k > 0$: one from H_1, one from H_2, and the third from a previous merge of two copies of B_{k-1}. In the presence of two or more copies of B_k, we merge two copies as per Algorithm 4.6 to produce B_{k+1}. If H_i has n_i vertices, then H_i has at most $1 + \lfloor \lg n_i \rfloor$ binomial trees, from which it is clear that merging H_1 and H_2 requires

$$\max(1 + \lfloor \lg n_1 \rfloor, 1 + \lfloor \lg n_2 \rfloor)$$

steps. Letting $N = \max(n_1, n_2)$, we see that merging H_1 and H_2 takes logarithmic time $O(\lg N)$. The operation of merging two binomial heaps is presented in pseudocode as Algorithm 4.8, which is adapted from Cormen et al. [**?**, p.463] and the C implementation of binomial queues in [**?**]. A word of warning is order here. Algorithm 4.8 is destructive in the sense that it modifies the input heaps H_i in-place without making copies of those heaps.

Vertex insertion

Let v be a vertex with corresponding key κ_v and let H_1 be a binomial heap of n vertices. The single vertex v can be considered as a binomial heap H_2 comprised of exactly the binomial tree B_0. Then inserting v into H_1 is equivalent to merging the heaps H_i and can be accomplished in $O(\lg n)$ time. Refer to Algorithm 4.9 for pseudocode of this straightforward procedure.

Algorithm 4.8: Merging two binomial heaps.

Input: Two binomial heaps H_1 and H_2.
Output: A binomial heap H that results from merging the H_i.

```
1  H ← empty binomial heap
2  head[H] ← merge sort the root lists of H_1 and H_2
3  if head[H] = NULL then
4      return H
5  prevv ← NULL
6  v ← head[H]
7  nextv ← sibling[v]
8  while nextv ≠ NULL do
9      if degree[v] ≠ degree[nextv] or (sibling[nextv] ≠ NULL and
       degree[sibling[nextv]] = degree[v]) then
10         prevv ← v
11         v ← nextv
12     else if κ_v ≤ κ_nextv then
13         sibling[v] ← sibling[nextv]
14         link the roots nextv and v as per Algorithm 4.6
15     else
16         if prevv = NULL then
17             head[H] ← nextv
18         else
19             sibling[prevv] ← nextv
20         link the roots v and nextv as per Algorithm 4.6
21         v ← nextv
22     nextv ← sibling[v]
23 return H
```

Algorithm 4.9: Insert a vertex into a binomial heap.

Input: A binomial heap H and a vertex v.
Output: The heap H with v inserted into it.

```
1  H_1 ← empty binomial heap
2  head[H_1] ← v
3  parent[v] ← NULL
4  child[v] ← NULL
5  sibling[v] ← NULL
6  degree[v] ← 0
7  H ← merge H and H_1 as per Algorithm 4.8
```

Delete minimum vertex

Extracting the minimum vertex from a binomial heap H consists of several phases. Let H be comprised of the binomial trees $B_{k_0}, B_{k_1}, \ldots, B_{k_m}$ with corresponding roots $r_{k_0}, r_{k_1}, \ldots, r_{k_m}$ and let n be the number of vertices in H. In the first phase, from among the r_{k_i} we identify the root v with minimum key and remove v from H, an operation that runs in $O(\lg n)$ time because we need to process at most $1 + \lfloor \lg n \rfloor$ roots. With the binomial tree B_k rooted at v thus severed from H, we now have a forest consisting of the heap without B_k (denote this heap by H_1) and the binomial tree B_k. By construction, v is the root of B_k and the children of v from left to right can be considered as roots of binomial trees as well, say $B_{\ell_s}, B_{\ell_{s-1}}, \ldots, B_{\ell_0}$ where $\ell_s > \ell_{s-1} > \cdots > \ell_0$. Now sever the root v from its children. The B_{ℓ_j} together can be viewed as a binomial heap H_2 with, from left to right, binomial trees $B_{\ell_0}, B_{\ell_1}, \ldots, B_{\ell_s}$. Finally the binomial heap resulting from removing v can be obtained by merging H_1 and H_2 in $O(\lg n)$ time as per Algorithm 4.8. In total we can extract the minimum vertex of H in $O(\lg n)$ time. Our discussion is summarized in Algorithm 4.10 and an illustration of the extraction process is presented in Figure 4.9.

Algorithm 4.10: Extract the minimum vertex from a binomial heap.

Input: A binomial heap H.
Output: The minimum vertex of H removed.

```
1 v ← extract minimum vertex from root list of H
2 H_2 ← empty binomial heap
3 L ← list of v's children reversed
4 head[H_2] ← L[0]
5 H ← merge H and H_2 as per Algorithm 4.8
6 return v
```

4.4 Binary search trees

A *binary search tree* (BST) is a rooted binary tree $T = (V, E)$ having vertex weight function $\kappa : V \to \mathbf{R}$. The weight of each vertex v is referred to as its key, denoted κ_v. Each vertex v of T satisfies the following properties:

- **Left subtree property.** The left subtree of v contains only vertices whose keys are at most κ_v. That is, if u is a vertex in the left subtree of v, then $\kappa_u \leq \kappa_v$.
- **Right subtree property.** The right subtree of v contains only vertices whose keys are at least κ_v. In other words, any vertex u in the right subtree of v satisfies $\kappa_v \leq \kappa_u$.
- **Recursion property.** Both the left and right subtrees of v must also be binary search trees.

The above are collectively called the *binary search tree property*. See Figure 4.10 for an example of a binary search tree. Based on the binary search tree property, we can use in-order traversal (see Algorithm 3.12) to obtain a listing of the vertices of a binary search tree sorted in nondecreasing order of keys.

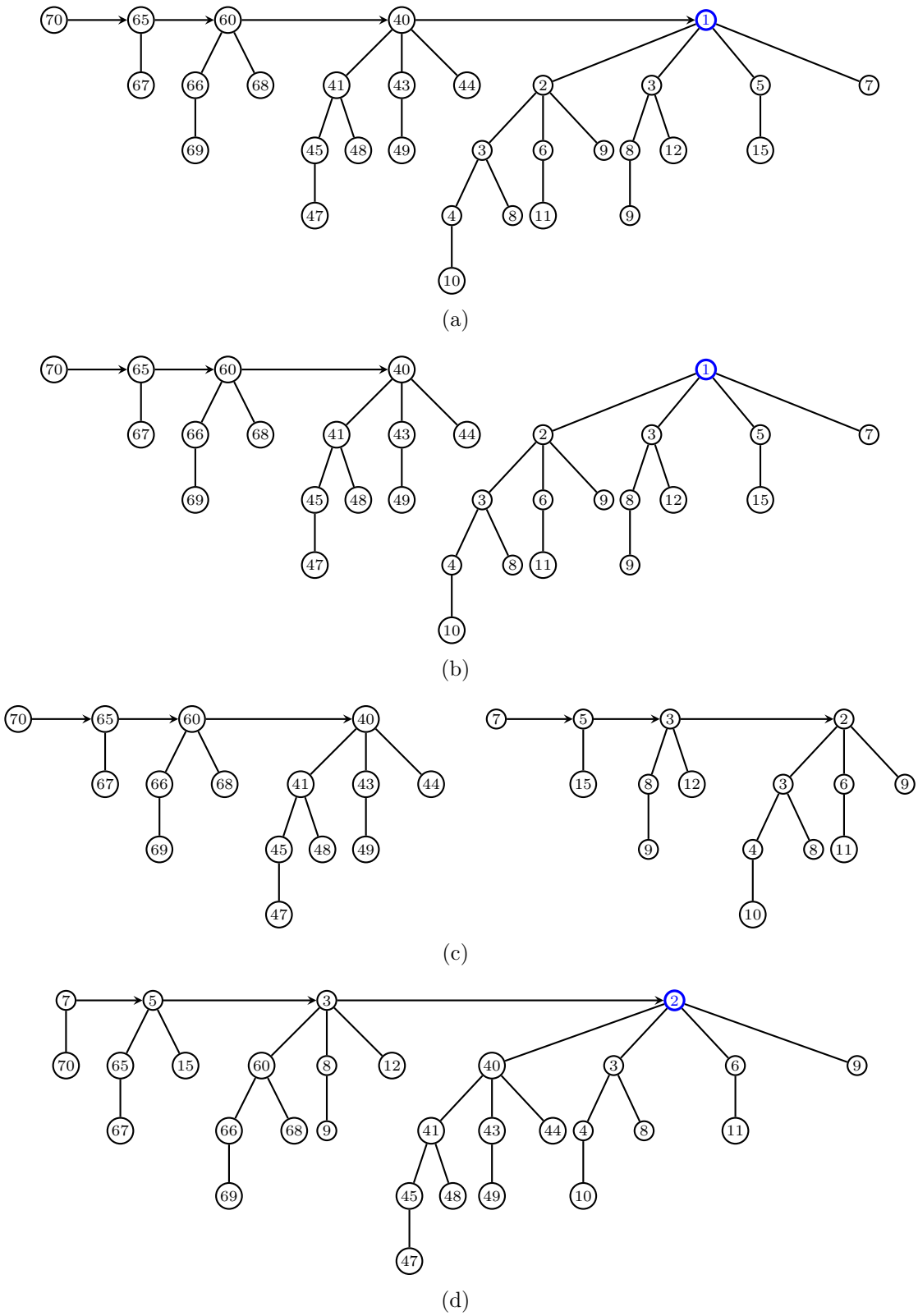


Figure 4.9: Extracting the minimum vertex from a binomial heap.

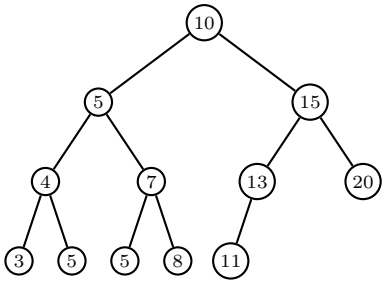


Figure 4.10: A binary search tree.

4.4.1 Searching

Given a BST T and a key k, we want to locate a vertex (if one exists) in T whose key is k. The search procedure for a BST is reminiscent of the binary search algorithm discussed in problem 2.10. We begin by examining the root v_0 of T. If $\kappa_{v_0} = k$, the search is successful. However, if $\kappa_{v_0} \neq k$ then we have two cases to consider. In the first case, if $k < \kappa_{v_0}$ then we search the left subtree of v_0. The second case occurs when $k > \kappa_{v_0}$, in which case we search the right subtree of v_0. Repeat the process until a vertex v in T is found for which $k = \kappa_v$ or the indicated subtree is empty. Whenever the target key is different from the key of the vertex we are currently considering, we move down one level of T. Thus if h is the height of T, it follows that searching T takes a worst-case runtime of $O(h)$. The above procedure is presented in pseudocode as Algorithm 4.11. Note that if a vertex v does not have a left subtree, the operation of locating the root of v's left subtree should return `NULL`. A similar comment applies when v does not have a right subtree. Furthermore, from the structure of Algorithm 4.11, if the input BST is empty then `NULL` is returned. See Figure 4.11 for an illustration of locating vertices with given keys in a BST.

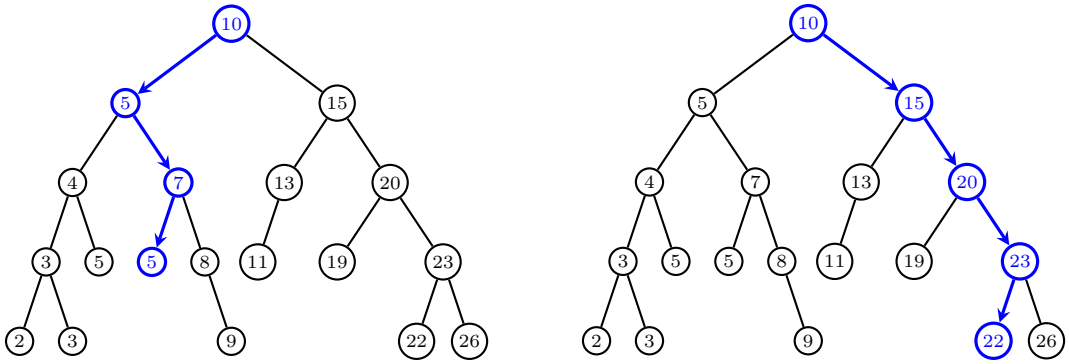
Algorithm 4.11: Locate a key in a binary search tree.

Input: A binary search tree T and a target key k.
Output: A vertex in T with key k. If no such vertex exists, return `NULL`.

```
1 v ← root[T]
2 while v ≠ NULL and k ≠ κ_v do
3     if k < κ_v then
4         v ← leftchild[v]
5     else
6         v ← rightchild[v]
7 return v
```

From the binary search tree property, deduce that a vertex of a BST T with minimum key can be found by starting from the root of T and repeatedly traversing left subtrees. When we have reached the left-most vertex v of T, querying for the left subtree of v should return `NULL`. At this point, we conclude that v is a vertex with minimum key. Each query for the left subtree moves us one level down T, resulting in a worst-case runtime of $O(h)$ with h being the height of T. See Algorithm 4.12 for pseudocode of the procedure.

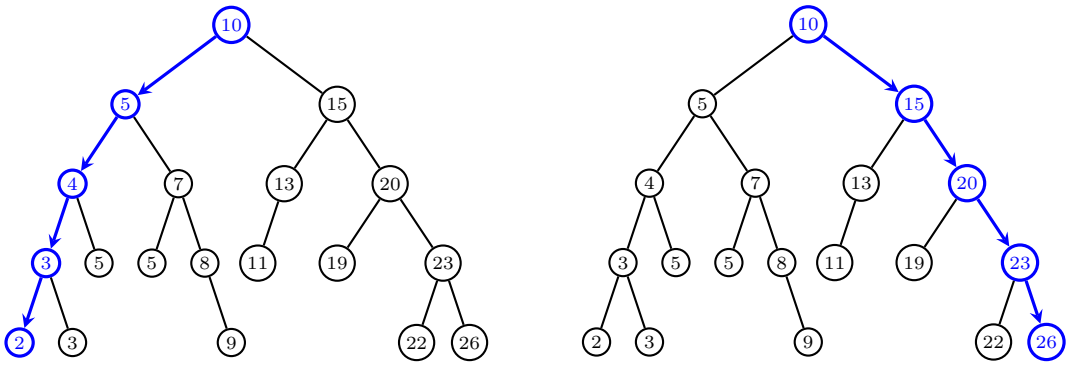
The procedure for finding a vertex with maximum key is analogous to that for finding



(a) Vertex with key 6: search fail.

(b) Vertex with key 22: search success.

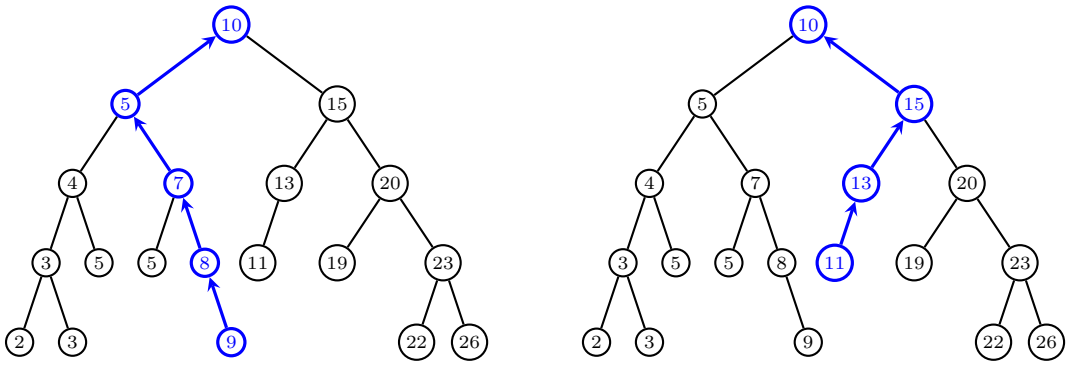
Figure 4.11: Finding vertices with given keys in a BST.



(a) Minimum vertex.

(b) Maximum vertex.

Figure 4.12: Locating minimum and maximum vertices in a BST.



(a) Successor of 9.

(b) Predecessor of 11.

Figure 4.13: Searching for successor and predecessor.

one with minimum key. Starting from the root of T, we repeatedly traverse right subtrees until we encounter the right-most vertex, which by the binary search tree property has maximum key. This procedure has the same worst-case runtime of $O(h)$. Figure 4.12 illustrates the process of locating the minimum and maximum vertices of a BST.

Algorithm 4.12: Finding a vertex with minimum key in a BST.

Input: A nonempty binary search tree T.
Output: A vertex of T with minimum key.

```
1 v ← root of T
2 while leftchild[v] ≠ NULL do
3     v ← leftchild[v]
4 return v
```

Corresponding to the notions of left- and right-children, we can also define successors and predecessors as follows. Suppose v is not a maximum vertex of a nonempty BST T. The *successor* of v is a vertex in T distinct from v with the smallest key greater than or equal to κ_v. Similarly, for a vertex v that is not a minimum vertex of T, the *predecessor* of v is a vertex in T distinct from v with the greatest key less than or equal to κ_v. The notions of successors and predecessors are concerned with relative key order, not a vertex's position within the hierarchical structure of a BST. For instance, from Figure 4.10 we see that the successor of the vertex u with key 8 is the vertex v with key 10, i.e. the root, even though v is an ancestor of u. The predecessor of the vertex a with key 4 is the vertex b with key 3, i.e. the minimum vertex, even though b is a descendant of a.

We now describe a method to systematically locate the successor of a given vertex. Let T be a nonempty BST and $v \in V(T)$ not a maximum vertex of T. If v has a right subtree, then we find a minimum vertex of v's right subtree. In case v does not have a right subtree, we backtrack up one level to v's parent $u = \text{parent}(v)$. If v is the root of the right subtree of u, we backtrack up one level again to u's parent, making the assignments $v \leftarrow u$ and $u \leftarrow \text{parent}(u)$. Otherwise we return v's parent. Repeat the above backtracking procedure until the required successor is found. Our discussion is summarized in Algorithm 4.13. Each time we backtrack to a vertex's parent, we move up one level, hence the worst-case runtime of Algorithm 4.13 is $O(h)$ with h being the height of T. The procedure for finding predecessors is similar. Refer to Figure 4.13 for an illustration of locating successors and predecessors.

4.4.2 Insertion

Inserting a vertex v into a BST T is rather straightforward. If T is empty, we let v be the root of T. Otherwise T has at least one vertex. In that case, we need to locate a vertex in T that can act as a parent and "adopt" v as a child. To find a candidate parent, let u be the root of T. If $\kappa_v < \kappa_u$ then we assign the root of the left subtree of u to u itself. Otherwise we assign the root of the right subtree of u to u. We then carry on the above key comparison process until the operation of locating the root of a left or right subtree returns `NULL`. At this point, a candidate parent for v is the last non-`NULL` value of u. If $\kappa_v < \kappa_u$ then we let v be u's left-child. Otherwise v is the right-child of u. After each key comparison, we move down at most one level so that in the worst-case inserting a vertex

Algorithm 4.13: Finding successors in a binary search tree.

Input: A nonempty binary search tree T and a vertex v that is not a maximum of T.
Output: The successor of v.

```
1 if rightchild[v] ≠ NULL then
2     return minimum vertex of v's right subtree as per Algorithm 4.12
3 u ← parent(v)
4 while u ≠ NULL and v = rightchild[u] do
5     v ← u
6     u ← parent(u)
7 return u
```

into T takes $O(h)$ time, where h is the height of T. Algorithm 4.14 presents pseudocode of our discussion and Figure 4.14 illustrates how to insert a vertex into a BST.

Algorithm 4.14: Inserting a vertex into a binary search tree.

Input: A binary search tree T and a vertex x to be inserted into T.
Output: The same BST T but augmeneted with x.

```
1  u ← NULL
2  v ← root of T
3  while v ≠ NULL do
4      u ← v
5      if κ_x < κ_v then
6          v ← leftchild[v]
7      else
8          v ← rightchild[v]
9  parent[x] ← u
10 if u = NULL then
11     root[T] ← x
12 else
13     if κ_x < κ_u then
14         leftchild[u] ← x
15     else
16         rightchild[u] ← x
```

4.4.3 Deletion

Whereas insertion into a BST is straightforward, removing a vertex requires much more work. Let T be a nonempty binary search tree and suppose we want to remove $v \in V(T)$ from T. Having located the position that v occupies within T, we need to consider three separate cases: (1) v is a leaf; (2) v has one child; (3) v has two children.

1. If v is a leaf, we simply remove v from T and the procedure is complete. The resulting tree without v satisfies the binary search tree property.

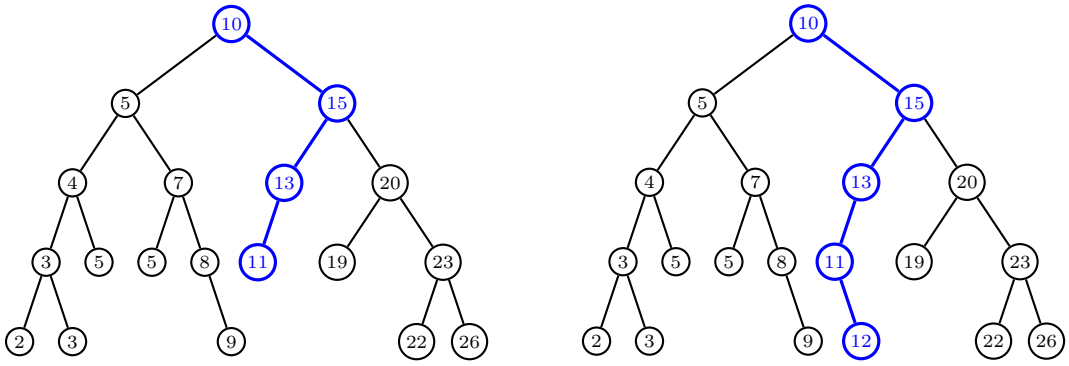


Figure 4.14: Inserting into a binary search tree.

Algorithm 4.15: Deleting a vertex from a binary search tree.

Input: A nonempty binary search tree T and a vertex $x \in V(T)$ to be removed from T.

Output: The same BST T but without x.

```
1  u ← NULL
2  v ← NULL
3  if leftchild[x] ≠ NULL or rightchild[x] ≠ NULL then
4      v ← x
5  else
6      v ← successor of x
7  if leftchild[v] ≠ NULL then
8      u ← leftchild[v]
9  else
10     u ← rightchild[v]
11 if u ≠ NULL then
12     parent[u] ← parent[v]
13 if parent[v] = NULL then
14     root[T] ← u
15 else
16     if v = leftchild[parent[v]] then
17         leftchild[parent[v]] ← u
18     else
19         rightchild[parent[v]] ← u
20 if v ≠ x then
21     κ_x ← κ_v
22     copy v's auxilary data into x
```

2. Suppose v has the single child u. Removing v would disconnect T, a situation that can be prevented by splicing out u and letting u occupy the position previously held by v. The resulting tree with v removed as described satisfies the binary search tree property.

3. Finally suppose v has two children and let s and p be the successor and predecessor of v, respectively. It can be shown that s has no left-child and p has no right-child. We can choose to either splice out s or p. Say we choose to splice out s. Then we remove v and let s hold the position previously occupied by v. The resulting tree with v thus removed satisfies the binary search tree property.

The above procedure is summarized in Algorithm 4.15, which is adapted from [**?**, p.262]. Figure 4.15 illustrates the various cases to be considered when removing a vertex from a BST. Note that in Algorithm 4.15, the process of finding the successor dominates the runtime of the entire algorithm. Other operations in the algorithm take at most constant time. Therefore deleting a vertex from a binary search tree can be accomplished in worst-case $O(h)$ time, where h is the height of the BST under consideration.

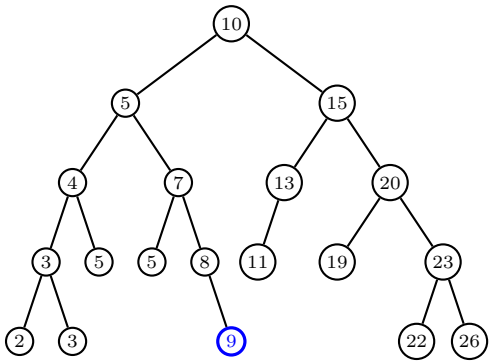
4.5 AVL trees

To motivate the need for AVL trees, note the lack of a structural property for binary search trees similar to the structural property for binary heaps. Unlike binary heaps, a BST is not required to have as small a height as possible. As a consequence, any given nonempty collection $C = \{v_0, v_1, \ldots, v_k\}$ of weighted vertices can be represented by various BSTs with different heights; see Figure 4.16. Some BST representations of C have heights smaller than other BST representations of C. Those BST representations with smaller heights can result in reduced time for basic operations such as search, insertion, and deletion and out-perform BST representations having larger heights. To achieve logarithmic or near-logarithmic time complexity for basic operations, it is desirable to maintain a BST with as small a height as possible.

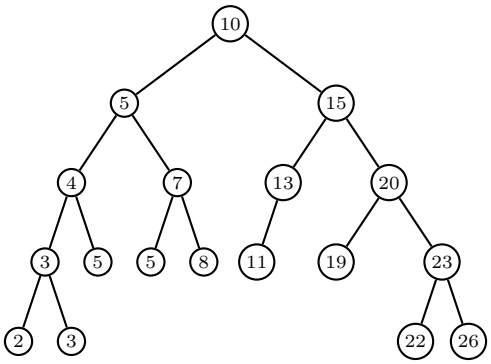
Adelson-Velskiĭ and Landis [**?**] introduced in 1962 a criterion for constructing and maintaining binary search trees having logarithmic heights. Recall that the height of a tree is the maximum depth of the tree. Then the Adelson-Velskiĭ-Landis criterion can be expressed as follows.

Definition 4.6. ***Height-balance property.*** *Let T be a binary tree and suppose v is an internal vertex of T. Let h_ℓ be the height of the left subtree of v and let h_r be the height of v's right subtree. Then v is said to be* height-balanced *if $|h_\ell - h_r| \leq 1$. For each internal vertex u of T, if u is height-balanced then the whole tree T is height-balanced.*

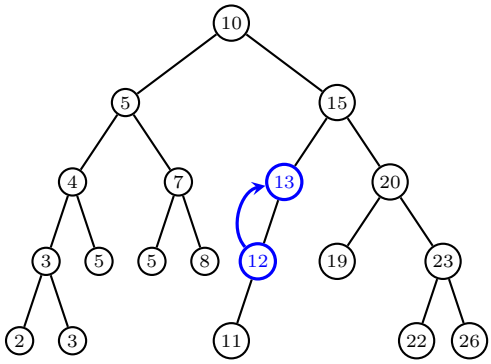
Binary trees having the height-balance property are called AVL trees. The structure of such trees is such that given any internal vertex v of an AVL tree, the heights of the left and right subtrees of v differ by at most 1. Complete binary trees are trivial examples of AVL trees, as are nearly complete binary trees. A less trivial example of AVL trees are what is known as *Fibonacci trees*, so named because the construction of Fibonacci trees bears some resemblance to how Fibonacci numbers are produced. Fibonacci trees can be constructed recursively in the following manner. The Fibonacci tree $\mathcal{F}_0$ of height 0 is the trivial tree. The Fibonacci tree $\mathcal{F}_1$ of height 1 is a binary tree whose left and right subtrees are both $\mathcal{F}_0$. For $n > 1$, the Fibonacci tree $\mathcal{F}_n$ of height n is a binary



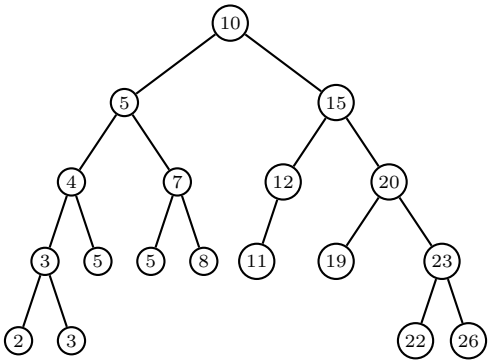
(a) Target vertex 9 is a leaf.



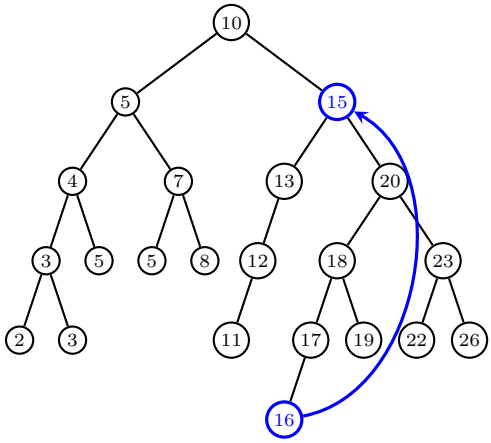
(b) Leaf deleted.



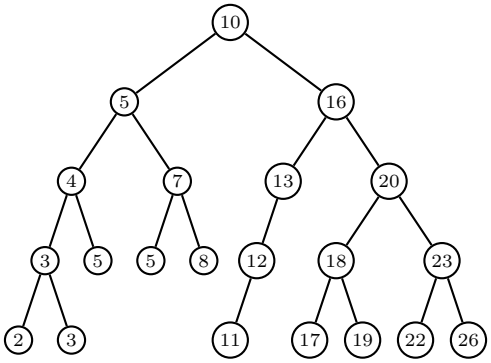
(c) Target vertex 13 has one child.



(d) Vertex deleted.



(e) Target vertex 15 has two children.



(f) Vertex deleted.

Figure 4.15: Deleting a vertex from a binary search tree.

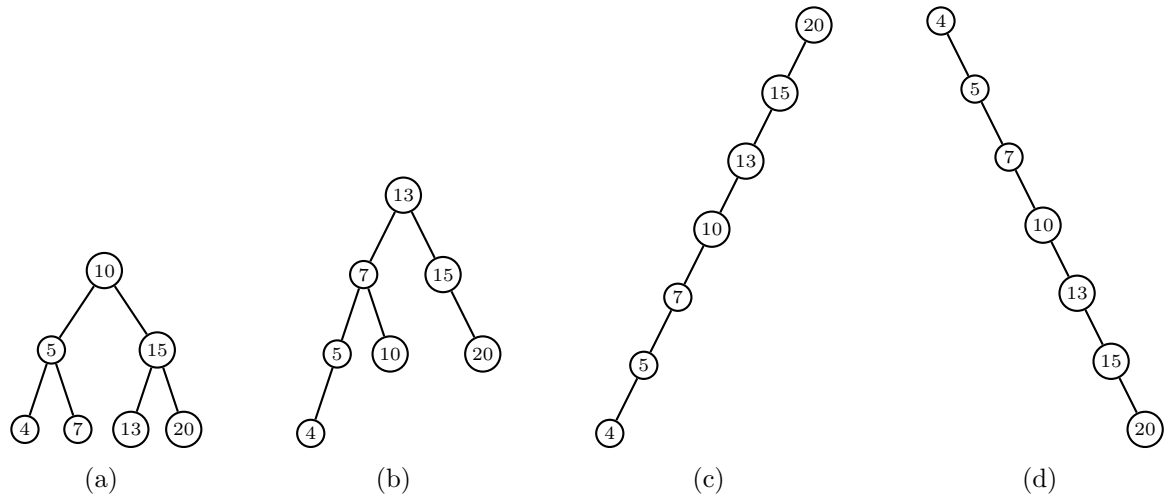


Figure 4.16: Different structural representations of a BST.

tree whose left and right subtrees are $\mathcal{F}_{n-2}$ and $\mathcal{F}_{n-1}$, respectively. Refer to Figure 4.17 for examples of Fibonacci trees; Figure 4.18 shows $\mathcal{F}_6$ together with subtree heights for vertex labels.

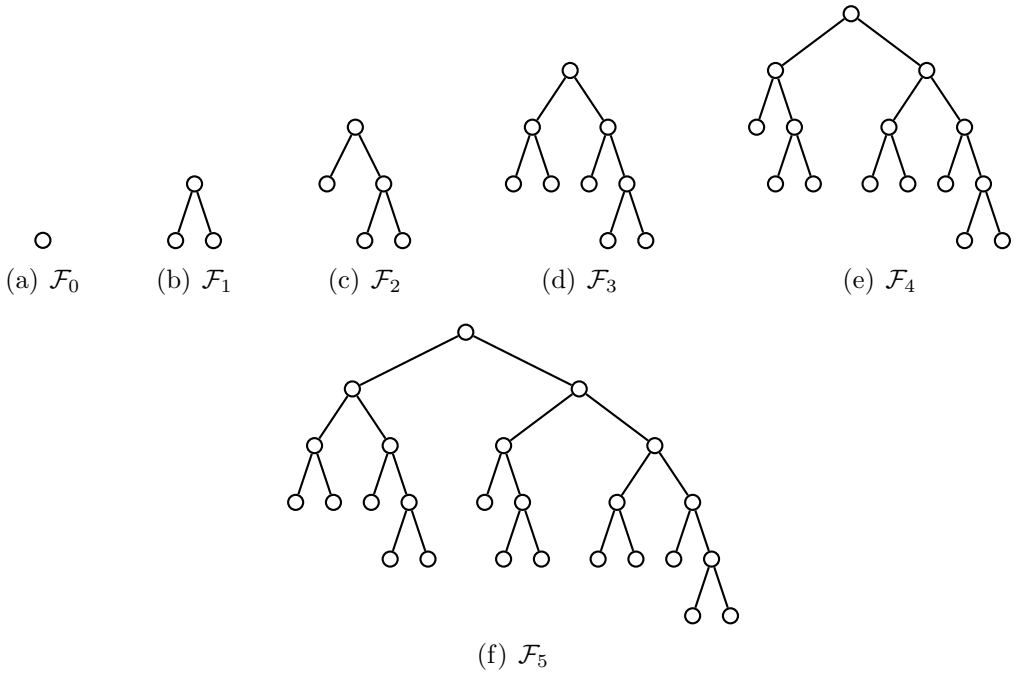


Figure 4.17: Fibonacci trees of heights $n = 0, 1, 2, 3, 4, 5$.

Theorem 4.7. ***Logarithmic height.*** *The height h of an AVL tree with n internal vertices is bounded by*

$$\lg(n+1) \leq h < 2 \cdot \lg n + 1.$$

Proof. Any binary tree of height h has at most 2^i leaves. From the proof of Corollary 4.3, we see that n is bounded by $2^{h-1} \leq n \leq 2^h - 1$ and in particular $n + 1 \leq 2^h$. Take the logarithm of both sides to get $h \geq \lg(n+1)$.

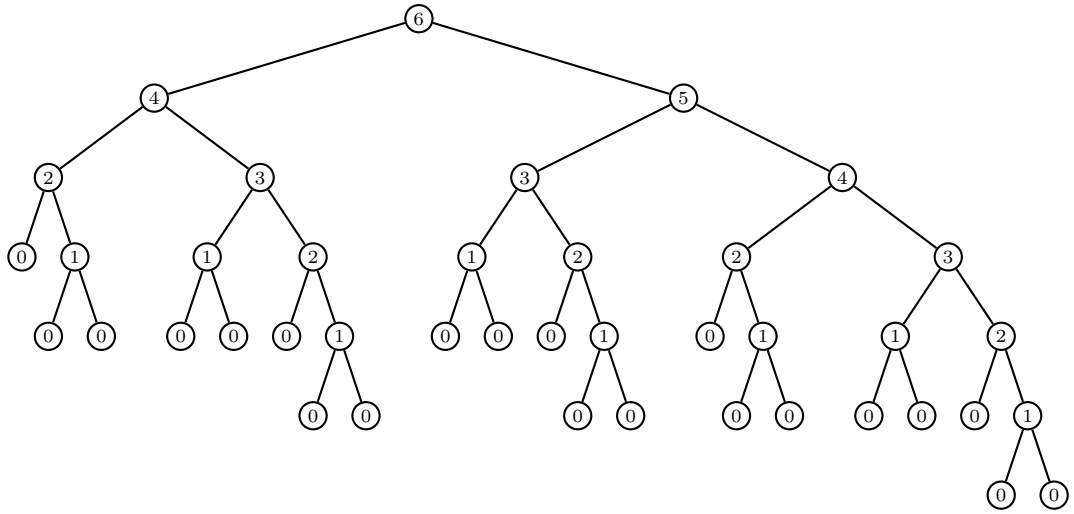


Figure 4.18: Fibonacci tree $\mathcal{F}_6$ with subtree heights for vertex labels.

Now instead of deriving an upper bound for h, we find the minimum order of an AVL tree and from there derive the required upper bound for h. Let T be an AVL tree of minimum order. One subtree of T has height $h-1$. The other subtree has height $h-1$ or $h-2$. Our objective is to construct T to have as small a number of vertices as possible. Without loss of generality, let the left and right subtrees of T have heights $h-2$ and $h-1$, respectively. The Fibonacci tree $\mathcal{F}_h$ of height h fits the above requirements for T. If $N(h)$ denote the number of internal vertices of $\mathcal{F}_h$, then $N(h) = 1+N(h-1)+N(h-2)$ is strictly increasing so

$$N(h) > N(h-2) + N(h-2) = 2 \cdot N(h-2). \tag{4.3}$$

Repeated application of (4.3) shows that

$$N(h) > 2^i \cdot N(h-2i) \tag{4.4}$$

for any integer i such that $h - 2i \geq 1$. Choose i so that $h - 2i = 1$ or $h - 2i = 2$, say the former. Substitute $i = (h-1)/2$ into (4.4) yields $N(h) > 2^{(h-1)/2}$. That is, $n > 2^{(h-1)/2}$ and taking logarithm of both sides yields $h < 2 \cdot \lg n + 1$. ■

An immediate consequence of Theorem 4.7 is that any binary search tree implemented as an AVL tree should have at most logarithmic height. Contrast this with a general BST of order N_1, whose height can be as low as logarithmic in N_1 or as high as linear in N_1. Translating to search time, we see that searching a general BST using Algorithm 4.11 is in the worst case $O(N_1)$, which is no better than searching a sorted list. However, if N_2 is the order of an AVL tree endowed with the binary search tree property, then searching the AVL tree using Algorithm 4.11 has worst-case $O(\lg N_2)$ runtime. While the worst-case runtime of searching a general BST can vary between $O(\lg N_1)$ and $O(N_1)$, that for an AVL tree with the binary search tree property is at most $O(\lg N_2)$.

4.5.1 Insertion

The algorithm for insertion into a BST can be modified and extended to support insertion into an AVL tree. Let T be an AVL tree having the binary search tree property, and v

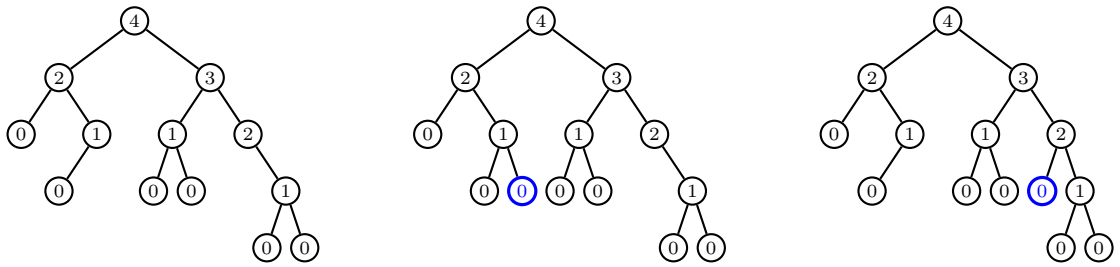
a vertex to be inserted into T. In the trivial case, T is the null tree so inserting v into T is equivalent to letting T be the trivial tree rooted at v. Consider now the case where T has at least one vertex. Apply Algorithm 4.14 to insert v into T and call the resulting augmented tree T_v. But our problem is not yet over; T_v may violate the height-balance property. To complete the insertion procedure, we require a technique to restore, if necessary, the height-balance property to T_v.

To see why the augmented tree T_v may not necessarily be height-balanced, let u be the parent of v in T_v, where previously u was a vertex T (and possibly a leaf). In the original AVL tree T, let $P_u : r = u_0, u_1, \ldots, u_k = u$ be the path from the root r of T to u with corresponding subtree heights $H(u_i) = h_i$ for $i = 0, 1, \ldots, k$. An effect of the insertion is to extend the path P_u to the longer path $P_v : r = u_0, u_1, \ldots, u_k = u, v$ and possibly increase subtree heights by one. One of two cases can occur with respect to T_v.

1. Height-balanced: T_v is height-balanced so no need to do anything further. A simple way to detect this is to consider the subtree S rooted at u, the parent of v. If S has two children, then no height adjustment need to take place for vertices in P_u, hence T_v is an AVL tree (see Figure 4.19). Otherwise we perform any necessary height adjustment for vertices in P_u, starting from $u_k = u$ and working our way up to the root $r = u_0$. After adjusting the height of u_i, we test to see whether u_i (with its new height) is height-balanced. If each of the u_i with their new heights are height-balanced, then T_v is height-balanced.

2. Height-unbalanced: During the height adjustment phase, it may happen that some u_j with its new height is not height-balanced. Among all such height-unbalanced vertices, let u_ℓ be the first height-unbalanced vertex detected during the process of height adjustment starting from $u_k = u$ and going up towards $r = u_0$. We need to rebalance the subtree rooted at u_ℓ. Then we continue on adjusting heights of the remaining vertices in P_u, also performing height-rebalancing where necessary.

Case 1 is relatively straightforward, but it is case 2 that involves much intricate work.



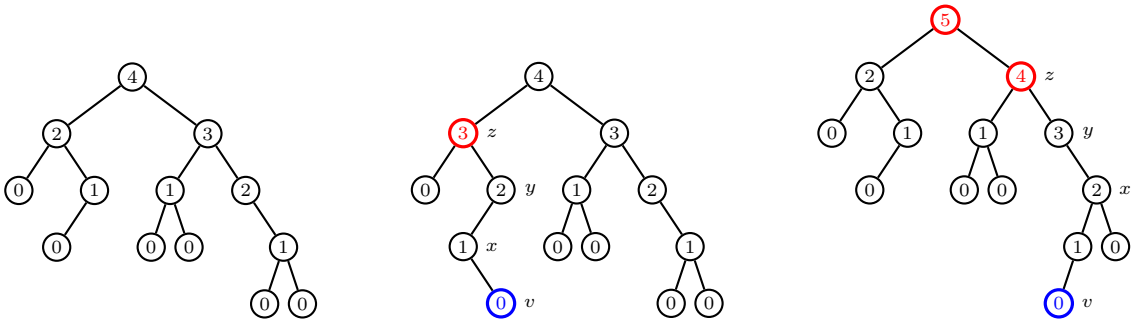
(a) Insert a vertex. (b) Vertex inserted. (c) Vertex inserted.

Figure 4.19: Augmented tree is balanced after insertion; vertex labels are heights.

We now turn to the case where inserting a vertex v into a nonempty AVL tree T results in an augmented tree T_v that is not height-balanced. A general idea for rebalancing (and hence restoring the height-balance property to) T_v is to determine where in T_v the height-balance property is first violated (the search phase), and then to locally rebalance subtrees at and around the point of violation (the repair phase). A description of the search phase follows. Let

$$P_v : r = u_0, u_1, \ldots, u_k = u, v$$

be the path from the root r of T_v (and hence of T) to v. Traversing upward from v to r, let z be the first height-unbalanced vertex. Among the children of z, let y be the child of higher height and hence an ancestor of v. Similarly, among the children of y let x be the child of higher height. In case a tie occurs, let x be the child of y that is also an ancestor of v. As each vertex is an ancestor of itself, it is possible that $x = v$. Furthermore, x is a grandchild of z because x is a child of y, which in turn is a child of z. The vertex z is not height-balanced due to inserting v into the subtree rooted at y, hence the height of y is 2 greater than its sibling (see Figure 4.20, where height-unbalanced vertices are colored red). We have determined the location at which the height-balance property is first violated.



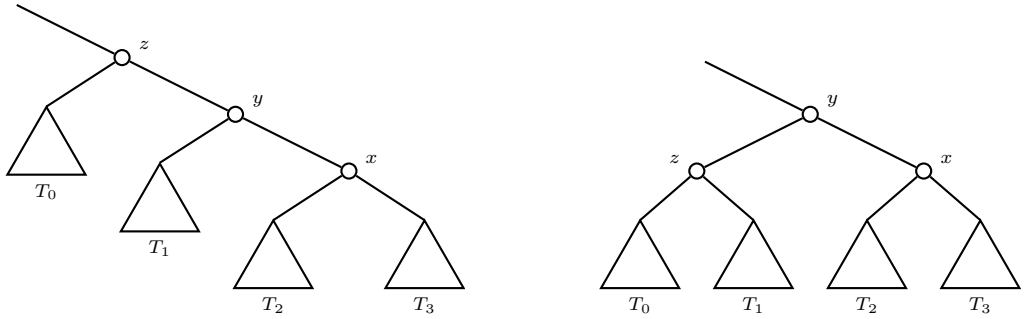
(a) Insert a vertex. (b) Vertex inserted. (c) Vertex inserted.

Figure 4.20: Augmented tree is unbalanced after insertion; vertex labels are heights.

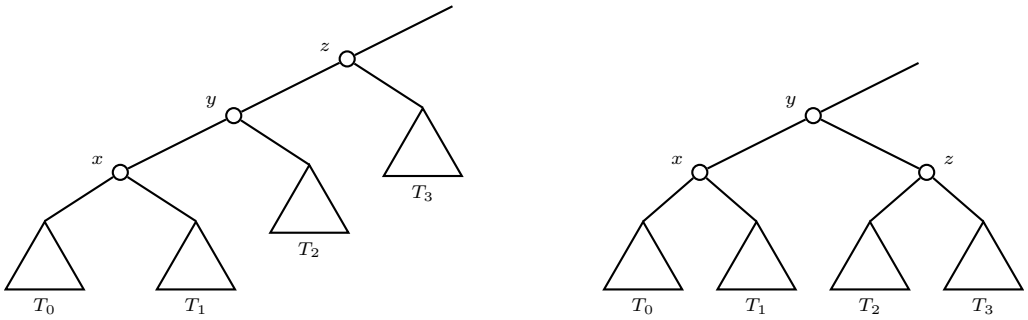
We now turn to the repair phase. The central question is: How are we to restore the height-balance property to the subtree rooted at z? By *trinode restructuring* is meant the process whereby the height-balance property is restored; the prefix "tri" refers to the three vertices x, y, z that are central to this process. A common name for the trinode restructuring is *rotation* in view of the geometric interpretation of the process. Figure 4.21 distinguishes four rotation possibilities, two of which are symmetrical to the other two. The single left rotation in Figure 4.21(a) occurs when $\text{height}(x) = \text{height}(\text{root}(T_0)) + 1$ and detailed in Algorithm 4.16. The single right rotation in Figure 4.21(b) occurs when $\text{height}(x) = \text{height}(\text{root}(T_3)) + 1$; see Algorithm 4.17 for pseudocode. Figure 4.21(c) illustrates the case of a right-left double rotation and occurs when $\text{height}(\text{root}(T_3)) = \text{height}(\text{root}(T_0))$; see Algorithm 4.18 for pseudocode to handle the rotation. The fourth case is illustrated in Figure 4.21(d) and occurs when $\text{height}(\text{root}(T_0)) = \text{height}(\text{root}(T_3))$; refer to Algorithm 4.19 for pseudocode to handle this left-right double rotation. Each of the four algorithms mentioned above run in constant time $O(1)$ and preserves the in-order traversal ordering of all vertices in T_v. In all, the insertion procedure is summarized in Algorithm 4.20. If h is the height T, locating and inserting the vertex v takes worst-case $O(h)$ time, which is also the worst-case runtime for the search-and-repair phase. Thus letting n be the number of vertices in T, insertion takes worst-case $O(\lg n)$ time.

4.5.2 Deletion

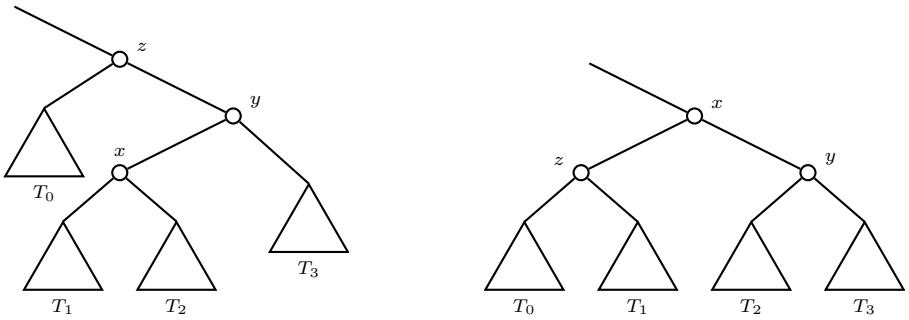
The process of removing a vertex from an AVL tree is similar to the insertion procedure. However, instead of using the insertion algorithm for BST, we use the deletion



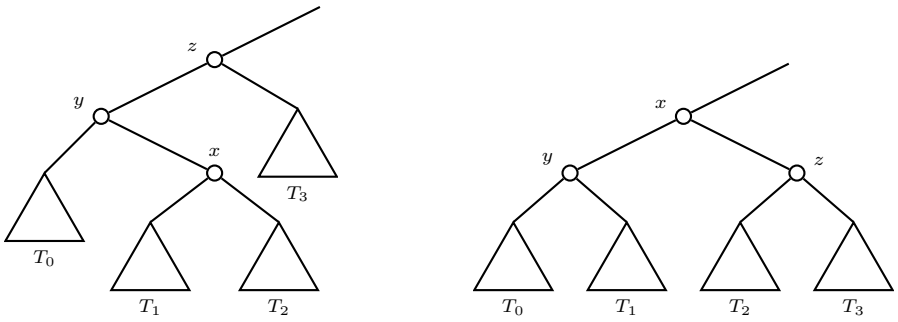
(a) Left rotation of y over z.



(b) Right rotation of y over z.



(c) Double rotation: right rotation of x over y, then left rotation over z.



(d) Double rotation: left rotation of x over y, then right rotation over z.

Figure 4.21: Rotations in the trinode restructuring process.

Algorithm 4.16: Single left rotation in the trinode restructure process.

Input: Three vertices x, y, z of an augmented AVL tree T_v, where z is the first height-unbalanced vertex in the path from v up to the root of T_v. The left subtree of z is denoted T_0 and the left subtree of y is T_1. The left and right subtrees of x are T_2 and T_3, respectively.

Output: A single left rotation to height-balance the subtree rooted at z.

1 rightchild[parent[z]] $\leftarrow y$
2 parent[y] $\leftarrow$ parent[z]
3 parent[z] $\leftarrow y$
4 leftchild[y] $\leftarrow z$
5 parent[root[T_1]] $\leftarrow z$
6 rightchild[z] $\leftarrow$ root[T_1]
7 height[z] $\leftarrow 1 + \max($height[root[T_0]], height[root[T_1]]$)$
8 height[x] $\leftarrow 1 + \max($height[root[T_2]], height[root[T_3]]$)$
9 height[y] $\leftarrow 1 + \max($height[x], height[z]$)$

Algorithm 4.17: Single right rotation in the trinode restructure process.

Input: Three vertices x, y, z of an augmented AVL tree T_v, where z is the first height-unbalanced vertex in the path from v up to the root of T_v. The left subtree of z is T_3 and the right subtree of y is T_2. The left and right subtrees of x are T_0 and T_1, respectively.

Output: A single right rotation to height-balance the subtree rooted at z.

1 leftchild[parent[z]] $\leftarrow y$
2 parent[y] $\leftarrow$ parent[z]
3 parent[z] $\leftarrow y$
4 rightchild[y] $\leftarrow z$
5 parent[root[T_2]] $\leftarrow z$
6 leftchild[z] $\leftarrow$ root[T_2]
7 height[x] $\leftarrow 1 + \max($height[root[T_0]], height[root[T_1]]$)$
8 height[z] $\leftarrow 1 + \max($height[root[T_2]], height[root[T_3]]$)$
9 height[y] $\leftarrow 1 + \max($height[x], height[z]$)$

Algorithm 4.18: Double rotation: right rotation followed by left rotation.

Input: Three vertices x, y, z of an augmented AVL tree T_v, where z is the first height-unbalanced vertex in the path from v up to the root of T_v. The left subtree of z is T_0 and the right subtree of y is T_3. The roots of the left and right subtrees of x are denoted T_1 and T_2, respectively.

Output: A right-left double rotation to height-balance the subtree rooted at z.

1 rightchild[parent[z]] $\leftarrow x$
2 parent[x] $\leftarrow$ parent[z]
3 parent[z] $\leftarrow x$
4 leftchild[x] $\leftarrow z$
5 rightchild[x] $\leftarrow y$
6 rightchild[z] $\leftarrow$ root[T_1]
7 parent[root[T_1]] $\leftarrow z$
8 parent[y] $\leftarrow x$
9 leftchild[y] $\leftarrow$ root[T_2]
10 parent[root[T_2]] $\leftarrow y$
11 height[z] $\leftarrow 1 + \max($height[root[T_0]], height[root[T_1]]$)$
12 height[y] $\leftarrow 1 + \max($height[root[T_2]], height[root[T_3]]$)$
13 height[x] $\leftarrow 1 + \max($height[y], height[z]$)$

Algorithm 4.19: Double rotation: left rotation followed by right rotation.

Input: Three vertices x, y, z of an augmented AVL tree T_v, where z is the first height-unbalanced vertex in the path from v up to the root of T_v. The left subtree of y is T_0 and the right subtree of z is T_3. The roots of the left and right subtrees of x are denoted T_1 and T_2, respectively.

Output: A left-right double rotation to height-balance the subtree rooted at z.

1 leftchild[parent[z]] $\leftarrow x$
2 parent[x] $\leftarrow$ parent[z]
3 parent[z] $\leftarrow x$
4 rightchild[x] $\leftarrow z$
5 leftchild[z] $\leftarrow$ root[T_2]
6 parent[T_2] $\leftarrow z$
7 leftchild[x] $\leftarrow y$
8 parent[y] $\leftarrow x$
9 rightchild[y] $\leftarrow$ root[T_1]
10 parent[root[T_1]] $\leftarrow y$
11 height[z] $\leftarrow 1 + \max($height[root[T_2]], height[root[T_3]]$)$
12 height[y] $\leftarrow 1 + \max($height[root[T_0]], height[root[T_1]]$)$
13 height[x] $\leftarrow 1 + \max($height[y], height[z]$)$

Algorithm 4.20: Insert a vertex into an AVL tree.

Input: An AVL tree T and a vertex v.
Output: The AVL tree T with v inserted into it.

```
1  insert v into T as per Algorithm 4.14
2  height[v] ← 0
3  u ← v                          /* begin height adjustment */
4  x ← NULL
5  y ← NULL
6  z ← NULL
7  while parent[u] ≠ NULL do
8      u ← parent[u]
9      if leftchild[u] ≠ NULL and rightchild[u] ≠ NULL then
10         h_ℓ ← height[leftchild[u]]
11         h_r ← height[rightchild[u]]
12         height[u] ← 1 + max(h_ℓ, h_r)
13         if |h_ℓ − h_r| > 1 then
14             if height[rightchild[rightchild[u]]] = height[leftchild[u]] + 1 then
15                 z ← u
16                 y ← rightchild[z]
17                 x ← rightchild[y]
18                 trinode restructuring as per Algorithm 4.16
19                 continue with next iteration of loop
20             if height[leftchild[leftchild[u]]] = height[rightchild[u]] + 1 then
21                 z ← u
22                 y ← leftchild[z]
23                 x ← leftchild[y]
24                 trinode restructuring as per Algorithm 4.17
25                 continue with next iteration of loop
26             if height[rightchild[rightchild[u]]] = height[leftchild[u]] then
27                 z ← u
28                 y ← rightchild[z]
29                 x ← leftchild[y]
30                 trinode restructuring as per Algorithm 4.18
31                 continue with next iteration of loop
32             if height[leftchild[leftchild[u]]] = height[rightchild[u]] then
33                 z ← u
34                 y ← leftchild[z]
35                 x ← rightchild[y]
36                 trinode restructuring as per Algorithm 4.19
37                 continue with next iteration of loop
38     if leftchild[u] ≠ NULL then
39         height[u] ← 1 + height[leftchild[u]]
40         continue with next iteration of loop
41     if rightchild[u] ≠ NULL then
42         height[u] ← 1 + height[rightchild[u]]
43         continue with next iteration of loop
```

Algorithm 4.15 for BST to remove the target vertex from an AVL tree. The resulting tree may violate the height-balance property, which can be restored using trinode restructuring.

Let T be an AVL tree having vertex v and suppose we want to remove v from T. In the trivial case, T is the trivial tree whose sole vertex is v. Deleting v is simply removing it from T so that T becomes the null tree. On the other hand, suppose T has at least $n > 1$ vertices. Apply Algorithm 4.15 to remove v from T and call the resulting tree with v removed T_v. It is possible that T_v does not satisfy the height-balance property. To restore the height-balance property to T_v, let u be the parent of v in T prior to deleting v from T. Having deleted v from T, let $P : r = u_0, u_1, \ldots, u_k = u$ be the path from the root r of T_v to u. Adjust the height of u and, traversing from u up to r, perform height adjustment to each vertex in P and where necessary carry out trinode restructuring. The resulting algorithm is very similar to Algorithm 4.20; see Algorithm 4.21 for pseudocode. The deletion procedure via Algorithm 4.15 requires worst-case runtime $O(\lg n)$, where n is the number of vertices in T, and the height-adjustment process runs in worst-case $O(\lg n)$ time as well. Thus Algorithm 4.21 has worst-case runtime of $O(\lg n)$.

4.6 Problems

> No problem is so formidable that you can't walk away from it.
> — Charles M. Schulz

4.1. Let Q be a priority queue of $n > 1$ elements, given in sequence representation. From section 4.1.1, we know that inserting an element into Q takes $O(n)$ time and deleting an element from Q takes $O(1)$ time.

(a) Suppose Q is an empty priority queue and let $e_0, e_1, \ldots, e_n$ be $n + 1$ elements we want to insert into Q. What is the total runtime required to insert all the e_i into Q while also ensuring that the resulting queue is a priority queue?

(b) Let $Q = [e_0, e_1, \ldots, e_n]$ be a priority queue of $n + 1$ elements. What is the total time required to remove all the elements of Q?

4.2. Prove the correctness of Algorithms 4.2 and 4.3.

4.3. Describe a variant of Algorithm 4.3 for modifying the key of the root of a binary heap, without extracting any vertex from the heap.

4.4. Section 4.2.2 describes how to insert an element into a binary heap T. The general strategy is to choose the first leaf following the last internal vertex of T, replace that leaf with the new element so that it becomes an internal vertex, and perform a sift-up operation from there. If instead we choose any leaf of T and replace that leaf with the new element, explain why we cannot do any better than Algorithm 4.2.

4.5. Section 4.2.3 shows how to extract the minimum vertex from a binary heap T. Instead of replacing the root with the last internal vertex of T, we could replace the root with any other vertex of T that is not a leaf and then proceed to maintain the heap-structure and heap-order properties. Explain why the latter strategy is not better than Algorithm 4.3.

Algorithm 4.21: Delete a vertex from an AVL tree.

Input: An AVL tree T and a vertex $v \in V(T)$.
Output: The AVL tree T with v removed from it.

```
1  u ← parent[v]
2  delete v from T as per Algorithm 4.15
3  adjust the height of u                    /* begin height adjustment */
4  x ← NULL
5  y ← NULL
6  z ← NULL
7  while parent[u] ≠ NULL do
8      u ← parent[u]
9      if leftchild[u] ≠ NULL and rightchild[u] ≠ NULL then
10         h_ℓ ← height[leftchild[u]]
11         h_r ← height[rightchild[u]]
12         height[u] ← 1 + max(h_ℓ, h_r)
13         if |h_ℓ − h_r| > 1 then
14             if height[rightchild[rightchild[u]]] = height[leftchild[u]] + 1 then
15                 z ← u
16                 y ← rightchild[z]
17                 x ← rightchild[y]
18                 trinode restructuring as per Algorithm 4.16
19                 continue with next iteration of loop
20             if height[leftchild[leftchild[u]]] = height[rightchild[u]] + 1 then
21                 z ← u
22                 y ← leftchild[z]
23                 x ← leftchild[y]
24                 trinode restructuring as per Algorithm 4.17
25                 continue with next iteration of loop
26             if height[rightchild[rightchild[u]]] = height[leftchild[u]] then
27                 z ← u
28                 y ← rightchild[z]
29                 x ← leftchild[y]
30                 trinode restructuring as per Algorithm 4.18
31                 continue with next iteration of loop
32             if height[leftchild[leftchild[u]]] = height[rightchild[u]] then
33                 z ← u
34                 y ← leftchild[z]
35                 x ← rightchild[y]
36                 trinode restructuring as per Algorithm 4.19
37                 continue with next iteration of loop
38     if leftchild[u] ≠ NULL then
39         height[u] ← 1 + height[leftchild[u]]
40         continue with next iteration of loop
41     if rightchild[u] ≠ NULL then
42         height[u] ← 1 + height[rightchild[u]]
43         continue with next iteration of loop
```

4.6. Let S be a sequence of $n > 1$ real numbers. How can we use algorithms described in section 4.2 to sort S?

4.7. The binary heaps discussed in section 4.2 are properly called minimum binary heaps because the root of the heap is always the minimum vertex. A corresponding notion is that of maximum binary heaps, where the root is always the maximum element. Describe algorithms analogous to those in section 4.2 for managing maximum binary heaps.

4.8. What is the total time required to extract all elements from a binary heap?

4.9. Numbers of the form $\binom{n}{r}$ are called binomial coefficients. They also count the number of r-combinations from a set of n objects. Algorithm 4.22 presents pseudocode to generate all the r-combinations of a set of n distinct objects. What is the worst-case runtime of Algorithm 4.22? Prove the correctness of Algorithm 4.22.

4.10. In contrast to enumerating all the r-combinations of a set of n objects, we may only want to generate a random r-combination. Describe and present pseudocode of a procedure to generate a random r-combination of $\{1, 2, \ldots, n\}$.

4.11. A problem related to the r-combinations of the set $S = \{1, 2, \ldots, n\}$ is that of generating the permutations of S. Algorithm 4.23 presents pseudocode to generate all the permutations of S in increasing lexicographic order. Find the worst-case runtime of this algorithm and prove its correctness.

4.12. Provide a description and pseudocode of an algorithm to generate a random permutation of $\{1, 2, \ldots, n\}$.

4.13. Takaoka [**?**] presents a general method for combinatorial generation that runs in $O(1)$ time. How can Takaoka's method be applied to generating combinations and permutations?

4.14. The proof of Lemma 4.4 relies on Pascal's formula, which states that for any positive integers n and r such that $r \leq n$, the following identity holds:

$$\binom{n+1}{r} = \binom{n}{r-1} + \binom{n}{r}.$$

Prove Pascal's formula.

4.15. Let m, n, r be nonnegative integers such that $r \leq n$. Prove the Vandermonde convolution

$$\binom{m+n}{r} = \sum_{k=0}^{r} \binom{m}{k}\binom{n}{r-k}.$$

The latter equation, also known as Vandermonde's identity, was already known as early as 1303 in China by Chu Shi-Chieh. Alexandre-Théophile Vandermonde independently discovered it and his result was published in 1772.

4.16. If m and n are nonnegative integers, prove that

$$\binom{m+n+1}{n} = \sum_{k=0}^{n} \binom{m+k}{k}.$$

Algorithm 4.22: Generating all the r-combinations of $\{1, 2, \ldots, n\}$.

Input: Two nonnegative integers n and r.
Output: A list L containing all the r-combinations of the set $\{1, 2, \ldots, n\}$ in increasing lexicographic order.

```
1  L ← [ ]
2  c_i ← i for i = 1, 2, ..., r
3  append(L, c_1 c_2 ··· c_r)
4  for i ← 2, 3, ..., (n choose r) do
5      m ← r
6      max ← n
7      while c_m = max do
8          m ← m − 1
9          max ← max − 1
10     c_m ← c_m + 1
11     c_j ← c_{j−1} + 1 for j = m + 1, m + 2, ..., r
12     append(L, c_1 c_2 ··· c_r)
13 return L
```

Algorithm 4.23: Generating all the permutations of $\{1, 2, \ldots, n\}$.

Input: A positive integer n.
Output: A list L containing all the permutations of $\{1, 2, \ldots, n\}$ in increasing lexicographic order.

```
1  L ← [ ]
2  c_i ← i for i = 1, 2, ..., n
3  append(L, c_1 c_2 ··· c_n)
4  for i ← 2, 3, ..., n! do
5      m ← n − 1
6      while c_m > c_{m+1} do
7          m ← m − 1
8      k ← n
9      while c_m > c_k do
10         k ← k − 1
11     swap the values of c_m and c_k
12     p ← m + 1
13     q ← n
14     while p < q do
15         swap the values of c_p and c_q
16         p ← p + 1
17         q ← q − 1
18     append(L, c_1 c_2 ··· c_n)
19 return L
```

4.17. Let n be a positive integer. How many distinct binomial heaps having n vertices are there?

4.18. The algorithms described in section 4.3 are formally for minimum binomial heaps because the vertex at the top of the heap is always the minimum vertex. Describe analogous algorithms for maximum binomial heaps.

4.19. If H is a binomial heap, what is the total time required to extract all elements from H?

4.20. Frederickson [**?**] describes an $O(k)$ time algorithm for finding the k-th smallest element in a binary heap. Provide a description and pseudocode of Frederickson's algorithm and prove its correctness.

4.21. Fibonacci heaps [**?**] allow for amortized $O(1)$ time with respect to finding the minimum element, inserting an element, and merging two Fibonacci heaps. Deleting the minimum element takes amortized time $O(\lg n)$, where n is the number of vertices in the heap. Describe and provide pseudocode of the above Fibonacci heap operations and prove the correctness of the procedures.

4.22. Takaoka [**?**] introduces another type of heap called a 2-3 heap. Deleting the minimum element takes amortized $O(\lg n)$ time with n being the number of vertices in the 2-3 heap. Inserting an element into the heap takes amortized $O(1)$ time. Describe and provide pseudocode of the above 2-3 heap operations. Under which conditions would 2-3 heaps be more efficient than Fibonacci heaps?

4.23. In 2000, Chazelle [**?**] introduced the soft heap, which can perform common heap operations in amortized $O(1)$ time. He then applied [**?**] the soft heap to realize a very efficient implementation of an algorithm for finding minimum spanning trees. In 2009, Kaplan and Zwick [**?**] provided a simple implementation and analysis of Chazelle's soft heap. Describe soft heaps and provide pseudocode of common heap operations. Prove the correctness of the algorithms and provide runtime analyses. Describe how to use soft heap to realize an efficient implementation of an algorithm to produce minimum spanning trees.

4.24. Explain any differences between the binary heap-order property, the binomial heap-order property, and the binary search tree property. Can in-order traversal be used to list the vertices of a binary heap in sorted order? Explain why or why not.

4.25. Present pseudocode of an algorithm to find a vertex with maximum key in a binary search tree.

4.26. Compare and contrast algorithms for locating minimum and maximum elements in a list with their counterparts for a binary search tree.

4.27. Let T be a nonempty BST and suppose $v \in V(T)$ is not a minimum vertex of T. If h is the height of T, describe and present pseudocode of an algorithm to find the predecessor of v in worst-case time $O(h)$.

4.28. Let $L = [v_0, v_1, \ldots, v_n]$ be the in-order listing of a BST T. Present an algorithm to find the successor of $v \in V(T)$ in constant time $O(1)$. How can we find the predecessor of v in constant time as well?

4.29. Modify Algorithm 4.15 to extract a minimum vertex of a binary search tree. Now do the same to extract a maximum vertex. How can Algorithm 4.15 be modified to extract a vertex from a binary search tree?

4.30. Let v be a vertex of a BST and suppose v has two children. If s and p are the successor and predecessor of v, respectively, show that s has no left-child and p has no right-child.

4.31. Let $L = [e_0, e_1, \ldots, e_n]$ be a list of $n+1$ elements from a totally ordered set X with total order $\leq$. How can binary search trees be used to sort L?

4.32. Describe and present pseudocode of a recursive algorithm for each of the following operations on a BST.

(a) Find a vertex with a given key.

(b) Locate a minimum vertex.

(c) Locate a maximum vertex.

(d) Insert a vertex.

4.33. Are the algorithms presented in section 4.4 able to handle a BST having duplicate keys? If not, modify the relevant algorithm(s) to account for the case where two vertices in a BST have the same key.

4.34. The notion of vertex level for binary trees can be extended to general rooted trees as follows. Let T be a rooted tree with $n > 0$ vertices and height h. Then level $0 \leq i \leq h$ of T consists of all those vertices in T that have the same depth i. If each vertex at level i has $i + m$ children for some fixed integer $m > 0$, what is the number of vertices at each level of T?

4.35. Compare the search, insertion, and deletion times of AVL trees and random binary search trees. Provide empirical results of your comparative study.

4.36. Describe and present pseudocode of an algorithm to construct a Fibonacci tree of height n for some integer $n \geq 0$. Analyze the worst-case runtime of your algorithm.

4.37. The upper bound in Theorem 4.7 can be improved as follows. From the proof of the theorem, we have the recurrence relation $N(h) > N(h-1) + N(h-2)$.

(a) If $h \leq 2$, show that there exists some $c > 0$ such that $N(h) \geq c^h$.

(b) Assume for induction that

$$N(h) > N(h-1) + N(h-2) \geq c^{h-1} + c^{h-2}$$

for some $h > 2$. If $c > 0$, show that $c^2 - c - 1 = 0$ is a solution to the recurrence relation $c^{h-1} + c^{h-2}$ and that

$$N(h) > \left(\frac{1+\sqrt{5}}{2}\right)^h.$$

(c) Use the previous two parts to show that

$$h < \frac{1}{\lg \varphi} \cdot \lg n$$

where $\varphi = (1+\sqrt{5})/2$ is the golden ratio and n counts the number of internal vertices of an AVL tree of height h.

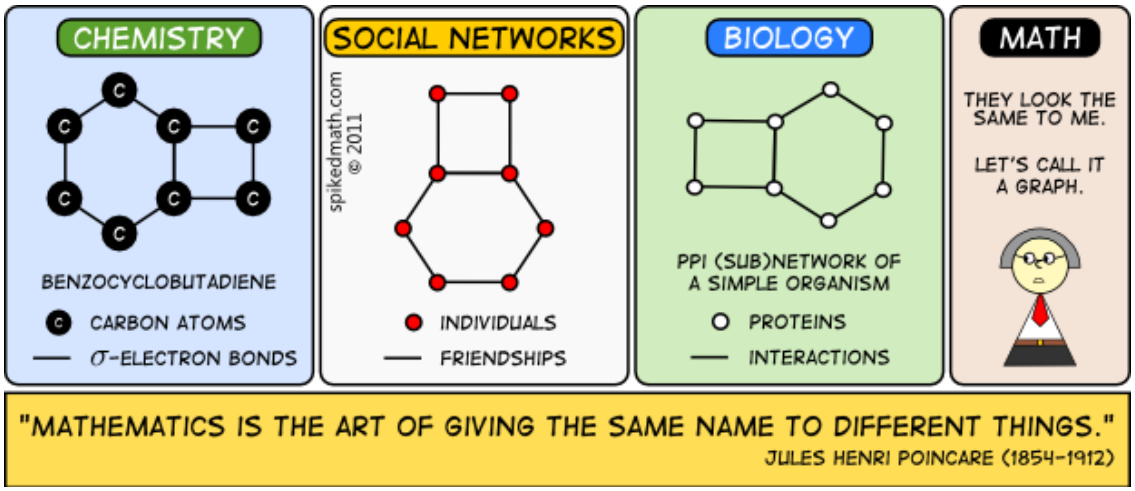
4.38. The Fibonacci sequence F_n is defined as follows. We have initial values $F_0 = 0$ and $F_1 = 1$. For $n > 1$, the n-th term in the sequence can be obtained via the recurrence relation $F_n = F_{n-1} + F_{n-2}$. Show that

$$F_n = \frac{\varphi^n - (-1/\varphi)^n}{\sqrt{5}} \tag{4.5}$$

where φ is the golden ratio. The closed form solution (4.5) to the Fibonacci sequence is known as Binet's formula, named after Jacques Philippe Marie Binet, even though Abraham de Moivre knew about this formula long before Binet did.

Chapter 5

Distance and connectivity



— Spiked Math, http://spikedmath.com/382.html

5.1 Paths and distance

5.1.1 Distance and metrics

Consider an edge-weighted simple graph $G = (V, E, i, h)$ without negative weight cycles. Here $E \subseteq V^{(2)}$, $i : E \to V^{(2)}$ is an incidence function as in (1.2), which we regard as the identity function, and $h : E \to V$ is an orientation function as in (1.3). Let $W : E \to \mathbf{R}$ be the weight function. (If G is not provided with a weight function on the edges, we assume that each edge has unit weight.) If $v_1, v_2 \in V$ are two vertices and $P = (e_1, e_2, \ldots, e_m)$ is a v_1-v_2 path (so v_1 is incident to e_1 and v_2 is incident to e_m), define the *weight* of P to be the sum of the weights of the edges in P:

$$W(P) = \sum_{i=1}^{m} W(e_i).$$

The *distance function* $d : V \times V \to \mathbf{R} \cup \{\infty\}$ on G is defined by

$$d(v_1, v_2) = \infty$$

if v_1 and v_2 lie in distinct connected components of G, and by

$$d(v_1, v_2) = \min_P W(P) \tag{5.1}$$

otherwise, where the minimum is taken over all paths P from v_1 to v_2. By hypothesis, G has no negative weight cycles so the minimum in (5.1) exists. It follows by definition of the distance function that $d(u, v) = \infty$ if and only if there is no path between u and v.

How we interpret the distance function d depends on the meaning of the weight function W. In practical applications, vertices can represent physical locations such as cities, sea ports, or landmarks. An edge weight could be interpreted as the physical distance in kilometers between two cities, the monetary cost of shipping goods from one sea port to another, or the time required to travel from one landmark to another. Then $d(u, v)$ could mean the shortest route in kilometers between two cities, the lowest cost incurred in transporting goods from one sea port to another, or the least time required to travel from one landmark to another.

The distance function d is not in general a metric, i.e. the triangle inequality does not in general hold for d. However, when the distance function is a metric then G is called a *metric graph*. The theory of metric graphs, due to their close connection with tropical curves, is an active area of research. For more information on metric graphs, see Baker and Faber [**?**].

5.1.2 Radius and diameter

A new hospital is to be built in a large city. Construction has not yet started and a number of urban planners are discussing the future location of the new hospital. What is a possible location for the new hospital and how are we to determine this location? This is an example of a class of problems known as facility location problems. Suppose our objective in selecting a location for the hospital is to minimize the maximum response time between the new hospital and the site of an emergency. To help with our decision making, we could use the notion of the center of a graph.

The center of a graph $G = (V, E)$ is defined in terms of the eccentricity of the graph under consideration. The *eccentricity* $\epsilon : V \to \mathbf{R}$ is defined as follows. For any vertex v, the eccentricity $\epsilon(v)$ is the greatest distance between v and any other vertex in G. In symbols, the eccentricity is expressible as

$$\epsilon(v) = \max_{u \in V} d(u, v).$$

For example, in a tree T with root r the eccentricity of r is the height of T. In the graph of Figure 5.1, the eccentricity of 2 is 5 and the shortest paths that yield $\epsilon(2)$ are

$$P_1 : 2, 3, 4, 14, 15, 16$$
$$P_2 : 2, 3, 4, 14, 15, 17.$$

The eccentricity of a vertex v can be thought of as an upper bound on the distance from v to any other vertex in G. Furthermore, we have at least one vertex in G whose distance from v is $\epsilon(v)$.

v	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\epsilon(v)$	6	5	4	4	5	6	7	7	5	6	7	7	6	5	6	7	7

Table 5.1: Eccentricity distribution for the graph in Figure 5.1.

To motivate the notion of the radius of a graph, consider the distribution of eccentricity among vertices of the graph G in Figure 5.1. The required eccentricity distribution

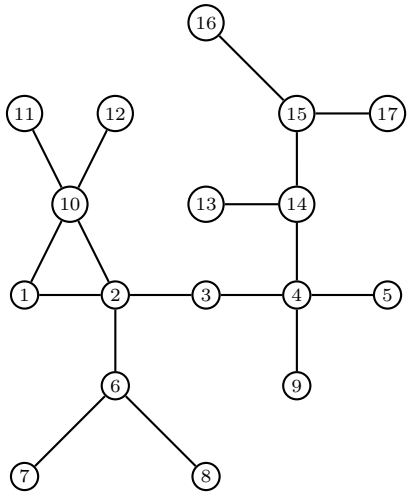


Figure 5.1: Determine the eccentricity, center, radius, and diameter.

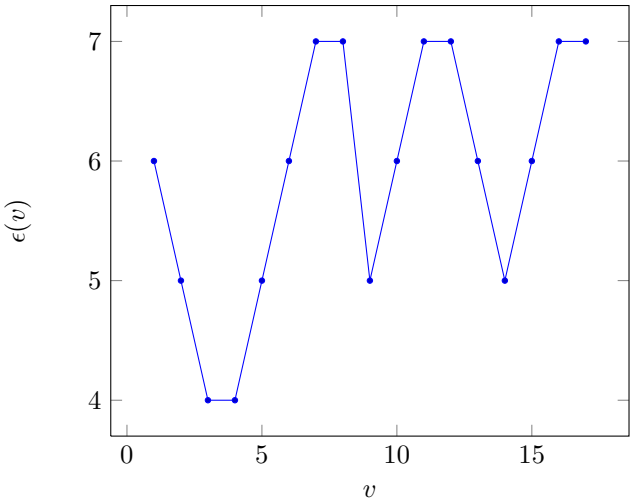


Figure 5.2: Eccentricity distribution of the graph in Figure 5.1. The horizontal axis represents the vertex name, while the vertical axis is the corresponding eccentricity.

is shown in Table 5.1. Among the eccentricities in the latter table, the minimum eccentricity is $\epsilon(3) = \epsilon(4) = 4$. An intuitive interpretation is that both of the vertices 3 and 4 have the shortest distance to any other vertices in G. We can invoke an analogy with plane geometry as follows. If a circle has radius r, then the distance from the center of the circle to any point within the circle is at most r. The minimum eccentricity in graph theory plays a role similar to the radius of a circle. If an object is strategically positioned—e.g. a vertex with minimum eccentricity or the center of a circle—then its greatest distance to any other object is guaranteed to be minimum. With the above analogy in mind, we define the *radius* of a graph $G = (V, E)$, written $\text{rad}(G)$, to be the minimum eccentricity among the eccentricity distribution of G. In symbols,

$$\text{rad}(G) = \min_{v \in V} \epsilon(v).$$

The *center* of G, written $C(G)$, is the set of vertices with minimum eccentricity. Thus the graph in Figure 5.1 has radius 4 and center $\{3, 4\}$. As should be clear from the latter example, the radius is a number whereas the center is a set. Refer to the beginning of the section where we mentioned the problem of selecting a location for a new hospital. We could use a graph to represent the geography of the city wherein the hospital is to be situated and select a location that is in the center of the graph.

Consider now the maximum eccentricity of a graph. In (2.5) we defined the *diameter* of a graph $G = (V, E)$ by

$$\text{diam}(G) = \max_{\substack{u,v \in V \\ u \neq v}} d(u, v).$$

The diameter of G can also be defined as the maximum eccentricity of any vertex in G:

$$\text{diam}(G) = \max_{v \in V} \epsilon(v).$$

In case G is disconnected, define its diameter to be $\text{diam}(G) = \infty$. To compute $\text{diam}(G)$, use the Floyd-Roy-Warshall algorithm (see section 2.6) to compute the shortest distance between each pair of vertices. The maximum of these distances is the diameter. The set of vertices of G with maximum eccentricity is called the *periphery* of G, written $\text{per}(G)$. The graph in Figure 5.1 has diameter 7 and periphery $\{7, 8, 11, 12, 16, 17\}$.

Theorem 5.1. ***Eccentricities of adjacent vertices.*** *Let $G = (V, E)$ be an undirected, connected graph having nonnegative edge weights. If $uv \in E$ and W is a weight function for G, then $|\epsilon(u) - \epsilon(v)| \leq W(uv)$.*

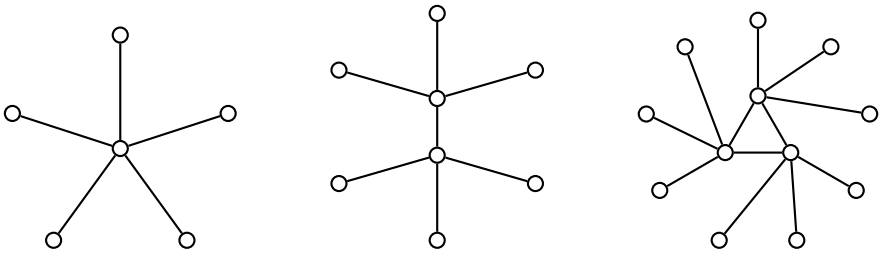
Proof. By definition, we have $d(u, x) \leq \epsilon(u)$ and $d(v, x) \leq \epsilon(v)$ for all $x \in V$. Let $w \in V$ such that $d(u, w) = \epsilon(u)$. Apply the triangle inequality to obtain

$$\begin{aligned} d(u, w) &\leq d(u, v) + d(v, w) \\ \epsilon(u) &\leq W(uv) + d(v, w) \\ &\leq W(uv) + \epsilon(v) \end{aligned}$$

from which we have $\epsilon(u) - \epsilon(v) \leq W(uv)$. Repeating the above argument with the role of u and v interchanged yields $\epsilon(v) - \epsilon(u) \leq W(uv)$. Both $\epsilon(u) - \epsilon(v) \leq W(uv)$ and $\epsilon(v) - \epsilon(u) \leq W(uv)$ together yields the inequality $|\epsilon(u) - \epsilon(v)| \leq W(uv)$ as required. ■

5.1.3 Center of trees

Given a tree T of order ≥ 3, we want to derive a bound on the number of vertices that comprise the center of T. A graph in general can have one, two, or more number of vertices for its center. Indeed, for any integer $n > 0$ we can construct a graph whose center has cardinality n. The cases for $n = 1, 2, 3$ are illustrated in Figure 5.3. But can we do the same for trees? That is, given any positive integer n does there exist a tree whose center has n vertices? It turns out that the center of a tree cannot have more than two vertices, a result first discovered [**?**] by Camille Jordan in 1869.



(a) $|C(G)| = 1$ (b) $|C(G)| = 2$ (c) $|C(G)| = 3$

Figure 5.3: Constructing graphs with arbitrarily large centers.

Theorem 5.2. ***Jordan [?].*** *If a tree T has order ≥ 3, then the center of T is either a single vertex or two adjacent vertices.*

Proof. As all eccentric vertices of T are leaves (see problem 5.7), removing all the leaves of T decreases the eccentricities of the remaining vertices by one. The tree comprised of the surviving vertices has the same center as T. Continue pruning leaves as described above and note that the tree comprised of the surviving vertices has the same center as the previous tree. After a finite number of leaf pruning stages, we eventually end up with a tree made up of either one vertex or two adjacent vertices. The vertex set of this final tree is the center of T. ■

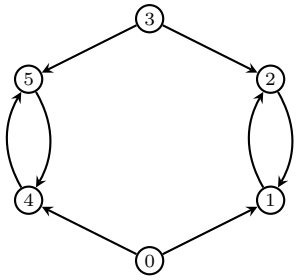
5.1.4 Distance matrix

In sections 1.3.4 and 2.3, the distance matrix D of a graph G was defined to be $D = [d_{ij}]$, where $d_{ij} = d(v_i, v_j)$ and the vertices of G are indexed by $V = \{v_0, v_1, \ldots, v_k\}$. The matrix D is square where we set $d_{ij} = 0$ for entries along the main diagonal. If there is no path from v_i to v_j, then we set $d_{ij} = \infty$. If G is undirected, then D is symmetric and is equal to its transpose, i.e. $D^T = D$. To compute the distance matrix D, apply the Floyd-Roy-Warshall algorithm to determine the distances between all pairs of vertices. Refer to Figure 5.4 for examples of distance matrices of directed and undirected graphs. In the remainder of this section, "graph" refers to an undirected graph unless otherwise specified.

Instead of one distance matrix, we can define several distance matrices on G. Consider an edge-weighted graph $G = (V, E)$ without negative weight cycles and let

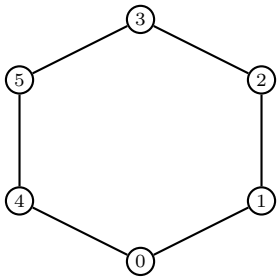
$$d : V \times V \to \mathbf{R} \cup \{\infty\}$$

be a distance function of G. Let $\partial = \text{diam}(G)$ be the diameter of G and index the vertices of G in some arbitrary but fixed manner, say $V = \{v_0, v_1, \ldots, v_n\}$. The sequence



$$\begin{bmatrix} 0 & 1 & 2 & \infty & 1 & 2 \\ \infty & 0 & 1 & \infty & \infty & \infty \\ \infty & 1 & 0 & \infty & \infty & \infty \\ \infty & 2 & 1 & 0 & 2 & 1 \\ \infty & \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{bmatrix}$$

(a)



$$\begin{bmatrix} 0 & 1 & 2 & 3 & 1 & 2 \\ 1 & 0 & 1 & 2 & 2 & 3 \\ 2 & 1 & 0 & 1 & 3 & 2 \\ 3 & 2 & 1 & 0 & 2 & 1 \\ 1 & 2 & 3 & 2 & 0 & 1 \\ 2 & 3 & 2 & 1 & 1 & 0 \end{bmatrix}$$

(b)

Figure 5.4: Distance matrices of directed and undirected graphs.

of *distance matrices* of G are a sequence of $(n-1) \times (n-1)$ matrices $A_1, A_2, \ldots, A_\partial$ where

$$(A_k)_{ij} = \begin{cases} 1, & \text{if } d(v_i, v_j) = k, \\ 0, & \text{otherwise.} \end{cases}$$

In particular, A_1 is the usual adjacency matrix A. To compute the sequence of distance matrices of G, use the Floyd-Roy-Warshall algorithm to compute the distance between each pair of vertices and assign the resulting distance to the corresponding matrix A_i.

The distance matrix arises in several applications, including communication network design [**?**] and network flow algorithms [**?**]. Thanks to Graham and Pollak [**?**], the following unusual fact is known. If T is any tree then

$$\det D(T) = (-1)^{n-1}(n-1)2^{n-2}$$

where n denotes the order of T. In particular, the determinant of the distance matrix of a tree is independent of the structure of the tree. This fact is proven in the paper [**?**], but see also [**?**].

5.2 Vertex and edge connectivity

If $G = (V, E)$ is a graph and $U \subseteq V$ is a vertex set with the property that $G - U$ has more connected components than G, then we call U a *vertex-cut*. The term *cut-vertex* or *cut-point* is used when the vertex-cut consists of exactly one vertex. For an intuitive appreciation of vertex-cut, suppose $G = (V, E)$ is a connected graph. Then $U \subseteq V$ is a vertex-cut if the vertex deletion subgraph $G - U$ is disconnected. For example, the cut-vertex of the graph in Figure 5.5 is the vertex 0. By $\kappa_v(G)$ we mean the *vertex connectivity* of a connected graph G, defined as the minimum number of vertices whose removal would either disconnect G or reduce G to the trivial graph. The

vertex connectivity $\kappa_v(G)$ is also written as $\kappa(G)$. The vertex connectivity of the graph in Figure 5.5 is $\kappa_v(G) = 1$ because we only need to remove vertex 0 in order to disconnect the graph. The vertex connectivity of a connected graph G is thus the vertex-cut of minimum cardinality. And G is said to be *k-connected* if $\kappa_v(G) \geq k$. From the latter definition, it immediately follows that if G has at least 3 vertices and is k-connected then any vertex-cut of G has at least cardinality k. For instance, the graph in Figure 5.5 is 1-connected. In other words, G is k-connected if the graph remains connected even after removing any $k-1$ or fewer vertices from G.

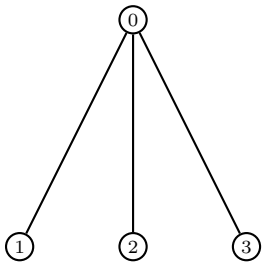


Figure 5.5: A claw graph with 4 vertices.

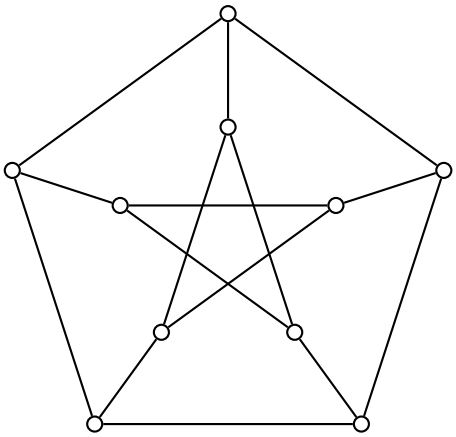


Figure 5.6: The Petersen graph on 10 vertices.

Example 5.3. Here is a Sage example concerning $\kappa(G)$ using the Petersen graph depicted in Figure 5.6. A linear programming Sage package, such as GLPK, must be installed for the commands below to work.

```
sage: G = graphs.PetersenGraph()
sage: len(G.vertices())
10
sage: G.vertex_connectivity()
3.0
sage: G.delete_vertex(0)
sage: len(G.vertices())
9
sage: G.vertex_connectivity()
2.0
```

■

The notions of edge-cut and cut-edge are similarly defined. Let $G = (V, E)$ be a graph and $D \subseteq E$ an edge set such that the edge deletion subgraph $G - D$ has more components than G. Then D is called an *edge-cut*. An edge-cut D is said to be *minimal*

if no proper subset of D is an edge-cut. The term *cut-edge* or *bridge* is reserved for the case where the set D is a singleton. Think of a cut-edge as an edge whose removal from a connected graph would result in that graph being disconnected. Going back to the case of the graph in Figure 5.5, each edge of the graph is a cut-edge. A graph having no cut-edge is called *bridgeless*. An open question as of 2010 involving bridges is the *cycle double cover conjecture*, due to Paul Seymour and G. Szekeres, which states that every bridgeless graph admits a set of cycles that contains each edge exactly twice. The *edge connectivity* of a connected graph G, written $\kappa_e(G)$ and sometimes denoted by $\lambda(G)$, is the minimum number of edges whose removal would disconnect G. In other words, $\kappa_e(G)$ is the minimum cardinality among all edge-cuts of G. Furthermore, G is said to be *k-edge-connected* if $\kappa_e(G) \geq k$. A connected graph that is k-edge-connected is guaranteed to be connected after removing $\leq k-1$ edges from it. When we have removed k or more edges, then the graph would become disconnected. By convention, a 1-edge-connected graph is simply a connected graph. The graph in Figure 5.5 has edge connectivity $\kappa_e(G) = 1$ and is 1-edge-connected.

Example 5.4. Here is a Sage example concerning $\lambda(G)$ using the Petersen graph shown in Figure 5.6. You must install an optional linear programming Sage package such as GLPK for the commands below to work.

```
sage: G = graphs.PetersenGraph()
sage: len(G.vertices())
10
sage: E = G.edges(); len(E)
15
sage: G.edge_connectivity()
3.0
sage: G.delete_edge(E[0])
sage: len(G.edges())
14
sage: G.edge_connectivity()
2.0
```

■

Vertex and edge connectivity are intimately related to the reliability and survivability of computer networks. If a computer network G (which is a connected graph) is k-connected, then it would remain connected despite the failure of at most $k-1$ network nodes. Similarly, G is k-edge-connected if the network remains connected after the failure of at most $k-1$ network links. In practical terms, a network with redundant nodes and/or links can afford to endure the failure of a number of nodes and/or links and still be connected, whereas a network with very few redundant nodes and/or links (e.g. something close to a spanning tree) is more prone to be disconnected. A k-connected or k-edge-connected network is more robust (i.e. can withstand) against node and/or link failures than is a j-connected or j-edge-connected network, where $j < k$.

Proposition 5.5. *If $\delta(G)$ is the minimum degree of an undirected connected graph $G = (V, E)$, then the edge connectivity of G satisfies $\lambda(G) \leq \delta(G)$.*

Proof. Choose a vertex $v \in V$ whose degree is $\deg(v) = \delta(G)$. Deleting the $\delta(G)$ edges incident on v suffices to disconnect G as v is now an isolated vertex. It is possible that G has an edge-cut whose cardinality is smaller than $\delta(G)$. Hence the result follows. ■

Let $G = (V, E)$ be a graph and suppose X_1 and X_2 comprise a partition of V. A *partition-cut* of G, denoted $\langle X_1, X_2 \rangle$, is the set of all edges of G with one endpoint in X_1 and the other endpoint in X_2. If G is a bipartite graph with bipartition X_1 and X_2, then $\langle X_1, X_2 \rangle$ is a partition-cut of G. It follows that a partition-cut is also an edge-cut.

Proposition 5.6. *An undirected connected graph G is k-edge-connected if and only if any partition-cut of G has at least k edges.*

Proof. Assume that G is k-edge-connected. Then each edge-cut has at least k edges. As a partition-cut is an edge-cut, then any partition-cut of G has at least k edges.

On the other hand, suppose each partition-cut has at least k edges. If D is a minimal edge-cut of G and X_1 and X_2 are the vertex sets of the two components of $G - D$, then $D = \langle X_1, X_2 \rangle$. To see this, note that $D \subseteq \langle X_1, X_2 \rangle$. If $\langle X_1, X_2 \rangle - D \neq \emptyset$ then choose some $e \in \langle X_1, X_2 \rangle$ such that $e \notin D$. The endpoints of e belong to the same component of $G - D$, in contradiction of the definition of X_1 and X_2. Thus any minimal edge-cut is a partition-cut and conclude that any edge-cut has at least k edges. ■

Proposition 5.7. *If $G = (V, E)$ is an undirected connected graph with vertex connectivity $\kappa(G)$ and edge connectivity $\lambda(G)$, then we have $\kappa(G) \leq \lambda(G)$.*

Proof. Let S be an edge-cut of G with cardinality $k = |S| = \lambda(G)$. Removing k suitably chosen vertices of G suffice to delete the edges of S and hence disconnect G. It is also possible to have a smaller vertex-cut elsewhere in G. Hence the inequality follows. ■

Taking together Propositions 5.5 and 5.7, we have Whitney's inequality.

Theorem 5.8. ***Whitney's inequality [?].*** *Let G be an undirected connected graph with vertex connectivity $\kappa(G)$, edge connectivity $\lambda(G)$, and minimum degree $\delta(G)$. Then we have the following inequality:*

$$\kappa(G) \leq \lambda(G) \leq \delta(G).$$

Proposition 5.9. *Let G be an undirected connected graph that is k-connected for some $k \geq 3$. If e is an edge of G, then the edge-deletion subgraph $G - e$ is $(k-1)$-connected.*

Proof. Let $V = \{v_1, v_2, \ldots, v_{k-2}\}$ be a set of $k-2$ vertices in $G - e$. It suffice to show the existence of a u-v walk in $(G-e)-V$ for any distinct vertices u and v in $(G-e)-V$. We need to consider two cases: (i) at least one of the endpoints of e is in V; and (ii) neither endpoints of e is in V.

(i) Assume that V has at least one endpoint of e. As $G - V$ is 2-connected, any distinct pair of vertices u and v in $G - V$ is connected by a u-v path that excludes e. Hence the u-v path is also in $(G - e) - V$.

(ii) Assume that neither endpoints of e is in V. If u and v are distinct vertices in $(G - e) - V$, then either: (1) both u and v are endpoints of e; or (2) at least one of u and v is an endpoint of e.

(1) Suppose u and v are both endpoints of e. As G is k-connected, then G has at least $k + 1$ vertices so that the vertex set of $G - \{v_1, v_2, \ldots, v_{k-2}, u, v\}$ is nonempty. Let w be a vertex of $G - \{v_1, v_2, \ldots, v_{k-2}, u, v\}$. Then there is a u-w path in $G - \{v_1, v_2, \ldots, v_{k-2}, v\}$ and a w-v path in $G - \{v_1, v_2, \ldots, v_{k-2}, u\}$. Neither the u-w nor the w-v paths contain e. The concatenation of these two paths is a u-v walk in $(G - e) - V$.

(2) Now suppose at least one of u and v, say u, is an endpoint of e. Let w be the other endpoint of e. As G is k-connected, then $G - \{v_1, v_2, \ldots, v_{k-2}, w\}$ is connected and we can find a u-v path P in $G - \{v_1, v_2, \ldots, v_{k-2}, w\}$. Furthermore P is a u-v path in $G - \{v_1, v_2, \ldots, v_{k-2}\}$ that neither contain w nor e. Hence P is a u-v path in $(G - e) - V$.

Conclude that $G - e$ is $(k-1)$-connected. ■

Repeated application of Proposition 5.9 results in the following corollary.

Corollary 5.10. *Let G be an undirected connected graph that is k-connected for some $k \geq 3$. If E is any set of m edges of G, for $m \leq k-1$, then the edge-deletion subgraph $G - E$ is $(k-m)$-connected.*

What does it mean for a communications network to be fault-tolerant? In 1932, Hassler Whitney provided [**?**] a characterization of 2-connected graphs whereby he showed that a graph G is 2-connected if and only if each pair of distinct vertices in G has two different paths connecting those two vertices. A key to understanding Whitney's characterization of 2-connected graphs is the notion of internal vertex of a path. Given a path P in a graph, a vertex along that path is said to be an *internal vertex* if it is neither the initial nor the final vertex of P. In other words, a path P has an internal vertex if and only if P has at least two edges. Building upon the notion of internal vertices, we now discuss what it means for two paths to be internally disjoint. Let u and v be distinct vertices in a graph G and suppose P_1 and P_2 are two paths from u to v. Then P_1 and P_2 are said to be *internally disjoint* if they do not share any common internal vertex. Two u-v paths are internally disjoint in the sense that both u and v are the only vertices to be found in common between those paths. The notion of internally disjoint paths can be easily extended to a collection of u-v paths. Whitney's characterization essentially says that a graph is 2-connected if and only if any two u-v paths are internally disjoint.

Consider the notion of internally disjoint paths within the context of communications network. As a first requirement for fault-tolerant communications network, we want the network to remain connected despite the failure of any network node. By Whitney's characterization, this is possible if the original communications network is 2-connected. That is, we say that a communications network is *fault-tolerant* provided that any pair of distinct nodes is connected by two internally disjoint paths. The failure of any node should at least guarantee that any two distinct nodes are still connected.

Theorem 5.11. ***Whitney's characterization of 2-connected graphs [?].*** *Let G be an undirected connected graph having at least 3 vertices. Then G is 2-connected if and only if any two distinct vertices in G are connected by two internally disjoint paths.*

Proof. ($\Longleftarrow$) For the case of necessity, argue by contraposition. That is, suppose G is not 2-connected. Let v be a cut-vertex of G, from which it follows that $G-v$ is disconnected. We can find two vertices w and x such that there is no w-x path in $G - v$. Therefore v is an internal vertex of any w-x path in G.

($\Longrightarrow$) For the case of sufficiency, let G be 2-connected and let u and v be any two distinct vertices in G. Argue by induction on $d(u,v)$ that G has at least two internally disjoint u-v paths. For the base case, suppose u and v are connected by an edge e so that $d(u,v) = 1$. Adapt the proof of Proposition 5.9 to see that $G-e$ is connected. Hence we can find a u-v path P in $G - e$ such that P and e are two internally disjoint u-v paths in G.

Assume for induction that G has two internally disjoint u-v paths where $d(u,v) < k$ for some $k \geq 2$. Let w and x be two distinct vertices in G such that $d(w,x) = k$ and hence there is a w-x path in G of length k, i.e. we have a w-x path

$$W : w = w_1, w_2, \ldots, w_{k-1}, w_k = x.$$

Note that $d(w, w_{k-1}) < k$ and apply the induction hypothesis to see that we have two internally disjoint w-w_{k-1} paths in G; call these paths P and Q. As G is 2-connected, we have a w-x path R in $G - w_{k-1}$ and hence R is also a w-x path in G. Let z be the vertex on R that immediately precedes x and assume without loss of generality that z is on P. We claim that G has two internally disjoint w-x paths. One of these paths is the concatenation of the subpath of P from w to z with the subpath of R from z to x. If x is not on Q, then construct a second w-x path, internally disjoint from the first one, as follows: concatenate the path Q with the edge $w_{k-1}w$. In case x is on Q, take the subpath of Q from w to x as the required second path. ■

From Theorem 5.11, an undirected connected graph G is 2-connected if and only if any two distinct vertices of G are connected by two internally disjoint paths. In particular, let u and v be any two distinct vertices of G and let P and Q be two internally disjoint u-v paths as guaranteed by Theorem 5.11. Starting from u, travel along the path P to arrive at v. Then start from v and travel along the path Q to arrive at u. The concatenation of the internally disjoint paths P and Q is hence a cycle passing through u and v. We have proved the following corollary to Theorem 5.11.

Corollary 5.12. *Let G be an undirected connected graph having at least 3 vertices. Then G is 2-connected if and only if any two distinct vertices of G lie on a common cycle.*

The following theorem provides further characterizations of 2-connected graphs, in addition to Whitney's characterization.

Theorem 5.13. ***Characterizations of 2-connected graphs.*** *Let $G = (V, E)$ be an undirected connected graph having at least 3 vertices. Then the following are equivalent.*

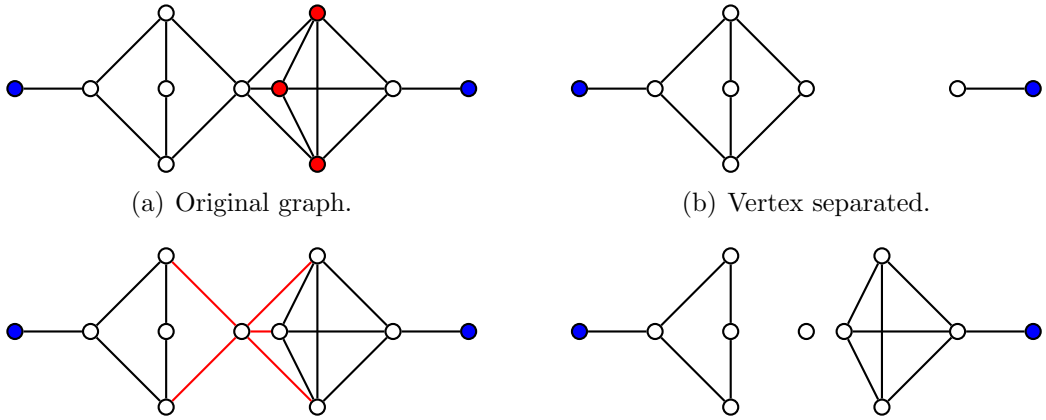
1. *G is 2-connected.*
2. *If $u, v \in V$ are distinct vertices of G, then u and v lie on a common cycle.*
3. *If $v \in V$ and $e \in E$, then v and e lie on a common cycle.*
4. *If $e_1, e_2 \in E$ are distinct edges of G, then e_1 and e_2 lie on a common cycle.*
5. *If $u, v \in V$ are distinct vertices and $e \in E$, then they lie on a common path.*
6. *If $u, v, w \in V$ are distinct vertices, then they lie on a common path.*
7. *If $u, v, w \in V$ are distinct vertices, then there is a path containing any two of these vertices but excluding the third.*

5.3 Menger's theorem

Menger's theorem has a number of different versions: an undirected, vertex-connectivity version; a directed, vertex-connectivity version; an undirected, edge-connectivity version; and a directed, edge-connectivity version. In this section, we will prove the undirected, vertex-connectivity version. But first, let's consider a few technical results that will be of use for the purpose of this section.

Let u and v be distinct vertices in a connected graph $G = (V, E)$ and let $S \subseteq V$. Then S is said to be u-v *separating* if u and v lie in different components of the vertex

deletion subgraph $G - S$. The vertices u and v are positioned such that after removing vertices in S from G and the corresponding edges, u and v are no longer connected nor strongly connected to each other. It is clear by definition that $u, v \notin S$. We also say that S *separates* u and v, or S is a vertex separating set. Similarly an edge set $T \subseteq E$ is u-v separating (or separates u and v) if u and v lie in different components of the edge deletion subgraph $G - T$. But unlike the case of vertex separating sets, it is possible for u and v to be endpoints of edges in T because the removal of edges does not result in deleting the corresponding endpoints. The set T is also called an edge separating set. In other words, S is a vertex cut and T is an edge cut. When it is clear from context, we simply refer to a separating set. See Figure 5.7 for illustrations of separating sets.



(a) Original graph. (b) Vertex separated.

(c) Original graph. (d) Edge separated.

Figure 5.7: Vertex and edge separating sets. Blue-colored vertices are those we want to separate. The red-colored vertices form a vertex separating set or vertex cut; the red-colored edges constitute an edge separating set or edge cut.

Proposition 5.14. *Consider two distinct, non-adjacent vertices u, v in a connected graph G. If $\mathcal{P}_{uv}$ is a collection of internally disjoint u-v paths in G and S_{uv} is a u-v separating set of vertices in G, then*

$$|\mathcal{P}_{uv}| \leq |S_{uv}|. \tag{5.2}$$

Proof. Each u-v path in $\mathcal{P}_{uv}$ must include at least one vertex from S_{uv} because S_{uv} is a vertex cut of G. Any two distinct paths in $\mathcal{P}_{uv}$ cannot contain the same vertex from S_{uv}. Thus the number of internally disjoint u-v paths is at most $|S_{uv}|$. ■

The bound (5.2) holds for any u-v separating set S_{uv} of vertices in G. In particular, we can choose S_{uv} to be of minimum cardinality among all u-v separating sets of vertices in G. Thus we have the following corollary. Menger's Theorem 5.18 provides a much stronger statement of Corollary 5.15, saying in effect that the two quantities $\max(|\mathcal{P}_{uv}|)$ and $\min(|S_{uv}|)$ are equal.

Corollary 5.15. *Consider any two distinct, non-adjacent vertices u, v in a connected graph G. Let $\max(|\mathcal{P}_{uv}|)$ be the maximum number of internally disjoint u-v paths in G and denote by $\min(|S_{uv}|)$ the minimum cardinality of a u-v separating set of vertices in G. Then we have $\max(|\mathcal{P}_{uv}|) \leq \min(|S_{uv}|)$.*

Corollary 5.16. *Consider any two distinct, non-adjacent vertices u, v in a connected graph G. Let $\mathcal{P}_{uv}$ be a collection of internally disjoint u-v paths in G and let S_{uv} be a u-v separating set of vertices in G. If $|\mathcal{P}_{uv}| = |S_{uv}|$, then $\mathcal{P}_{uv}$ has maximum cardinality among all collections of internally disjoint u-v paths in G and S_{uv} has minimum cardinality among all u-v separating sets of vertices in G.*

Proof. Argue by contradiction. Let $\mathcal{Q}_{uv}$ be another collection of internally disjoint u-v paths in G such that $|\mathcal{Q}_{uv}| \geq |\mathcal{P}_{uv}|$. Then $|\mathcal{P}_{uv}| \leq |\mathcal{Q}_{uv}| \leq |S_{uv}|$ by Proposition 5.14. We cannot have $|\mathcal{Q}_{uv}| > |\mathcal{P}_{uv}|$, which would be contradictory to our hypothesis that $\mathcal{P}_{uv} = |S_{uv}|$. Thus $|\mathcal{Q}_{uv}| = |\mathcal{P}_{uv}|$. Let T_{uv} be another u-v separating set of vertices in G such that $|T_{uv}| \leq |S_{uv}|$. Then we have $|\mathcal{P}_{uv}| \leq |T_{uv}| \leq |S_{uv}|$ by Proposition 5.14. We cannot have $|T_{uv}| < |S_{uv}|$ because we would then end up with $|\mathcal{P}_{uv}| \leq |T_{uv}|$ and $\mathcal{P}_{uv} = |S_{uv}|$, a contradiction. Therefore $|T_{uv}| = |S_{uv}|$. ■

Lemma 5.17. *Consider two distinct, non-adjacent vertices u, v in a connected graph G and let k be the minimum number of vertices required to separate u and v. If G has a u-v path of length 2, then G has k internally disjoint u-v paths.*

Proof. Argue by induction on k. For the base case, assume $k = 1$. Hence G has a cut vertex x such that u and v are disconnected in $G - x$. Any u-v path must contain x. In particular, there can be only one internally disjoint u-v path.

Assume for induction that $k \geq 2$. Let $P : u, x, v$ be a path in G having length 2 and suppose S is a smallest u-v separating set for $G - x$. Then $S \cup \{x\}$ is a u-v separating set for G. By the minimality of k, we have $|S| \geq k - 1$. By the induction hypothesis, we have at least $k - 1$ internally disjoint u-v paths in $G - x$. As P is internally disjoint from any of the latter paths, conclude that G has k internally disjoint u-v paths. ■

Theorem 5.18. ***Menger's theorem.*** *Let G be an undirected connected graph and let u and v be distinct, non-adjacent vertices of G. Then the maximum number of internally disjoint u-v paths in G equals the minimum number of vertices needed to separate u and v.*

Proof. Suppose that the maximum number of independent u-v paths in G is attained by u-v paths $P_1, \ldots, P_k$. To obtain a separating set $W \subset V$, we must at least remove one point in each path P_i. This implies the minimum number of vertices needed to separate u and v is at least k. Therefore, we have an upper bound:

$$\#\{\text{indep. } u - v \text{ paths}\} \leq \#\{\text{min. number of vertices needed to separate } u \text{ and } v\}.$$

We show that equality holds. Let n denote the number of edges of G. The proof is by induction on n. By hypothesis, $n \geq 2$. If $n = 2$ the statement holds by inspection, since in that case G is a line graph with 3 vertices $V = \{u, v, w\}$ and 2 edges, $E = \{uw.wv\}$. In that situation, there is only 1 u-v path (namely, uwv) and only one vertex separating u and v (namely, w).

Suppose now $n > 3$ and assume the statement holds for each graph with $< n$ edges. Let

$$k = \#\{\text{independent } u - v \text{ paths}\}$$

and let

$$\ell = \#\{\text{min. number of vertices needed to separate } u \text{ and } v\},$$

so that $k \leq \ell$. Let $e \in E$ and let G/e be the contraction graph having edges $E - \{e\}$ and vertices the same as those of G, except that the endpoints of e have been identified.

Suppose that $k < \ell$ and G does not have ℓ independent u-v paths. The contraction graph G/e does not have ℓ independent u-v paths either (where now, if e contains u or v then we must appropriately redefine u or v, if needed). However, by the induction hypothesis G/e does have the property that the maximum number of internally disjoint u-v paths equals the minimum number of vertices needed to separate u and v. Therefore,

$$\begin{aligned} &\#\{\text{independent } u-v \text{ paths in } G/e\} \\ &< \#\{\text{min. number of vertices needed to separate } u \text{ and } v \text{ in } G\}. \end{aligned}$$

By induction,

$$\begin{aligned} &\#\{\text{independent } u-v \text{ paths in } G/e\} \\ &= \#\{\text{min. number of vertices needed to separate } u \text{ and } v \text{ in } G/e\}. \end{aligned}$$

Now, we *claim* we can pick e such that e does contain u or v and in such a way that

$$\begin{aligned} &\#\{\text{minimum number of vertices needed to separate } u \text{ and } v \text{ in } G\} \\ &\geq \#\{\text{minimum number of vertices needed to separate } u \text{ and } v \text{ in } G/e\}. \end{aligned}$$

Proof: Indeed, since $n > 3$ any separating set realizing the minimum number of vertices needed to separate u and v in G cannot contain both a vertex in G adjacent to u and a vertex in G adjacent to v. Therefore, we may pick e accordingly. (Q.E.D. claim)

The result follows from the claim and the above inequalities. ■

The following statement is the undirected, edge-connectivity version of Menger's theorem.

Theorem 5.19. ***Menger's theorem (edge-connectivity form).*** Let G be an undirected graph, and let s and t be vertices in G. Then, the maximum number of edge-disjoint (s,t)-paths in G equals the minimum number of edges from $E(G)$ whose deletion separates s and t.

This is proven the same way as the previous version but using the generalized min-cut/max-flow theorem (see Remark 9.16 above).

Theorem 5.20. ***Dirac's theorem.*** *Let $G = (V, E)$ be an undirected k-connected graph with $|V| \geq k+1$ vertices for $k \geq 3$. If $S \subseteq V$ is any set of k vertices, then G has a cycle containing the vertices of S.*

Proof. ■

5.4 Whitney's Theorem

Theorem 5.21. ***Whitney's theorem (vertex version).*** Suppose $G = (V, E)$ is a graph with $|V| \geq k+1$. The following are equivalent:

- G is k-vertex-connected,
- Any pair of distinct vertices $v, w \in V$ are connected by at least k independent paths.

Solution. ...

■

Theorem 5.22. ***Whitney's theorem (edge version).*** Suppose $G = (V, E)$ is a graph with $|V| \geq k + 1$. The following are equivalent:

- the graph G is k-edge-connected,
- any pair of vertices are connected by at least k edge-disjoint paths.

Solution. ...

■

Theorem 5.23. ***Whitney's Theorem.*** *Let $G = (V, E)$ be a connected graph such that $|V| \geq 3$. Then G is 2-connected if and only if any pair $u, v \in V$ has two internally disjoint paths between them.*

5.5 Centrality of a vertex

> Louis, I think this is the beginning of a beautiful friendship.
> — Rick from the 1942 film *Casablanca*

- degree centrality
- betweenness centrality; for efficient algorithms, see [**?**, **?**]
- closeness centrality
- eigenvector centrality

The *degree centrality* of a graph $G = (V, E)$ is the list parameterized by the vertex set V of G whose v-th entry is the fraction of vertices connected to $v \in V$. The centrality of a vertex within a graph determines the relative importance of that vertex to its graph. Degree centrality measures the number of edges incident upon a vertex.

```
sage: G = graphs.RandomNewmanWattsStrogatz(6,2,1/2)
sage: G
Graph on 6 vertices
sage: D = G.degree_sequence()
sage: D
[5, 4, 3, 3, 3, 2]
sage: VG = G.vertices()
sage: VG
[0, 1, 2, 3, 4, 5]
sage: DC = [QQ(x)/len(VG) for x in D]
sage: DC
[5/6, 2/3, 1/2, 1/2, 1/2, 1/3]
```

This graph is shown in Figure 5.8.

The *closeness centrality* is defined to be

$$\frac{1}{\text{average distance to all vertices}}.$$

Closeness centrality is an inverse measure of centrality in that a larger value indicates a less central vertex while a smaller value indicates a more central vertex.

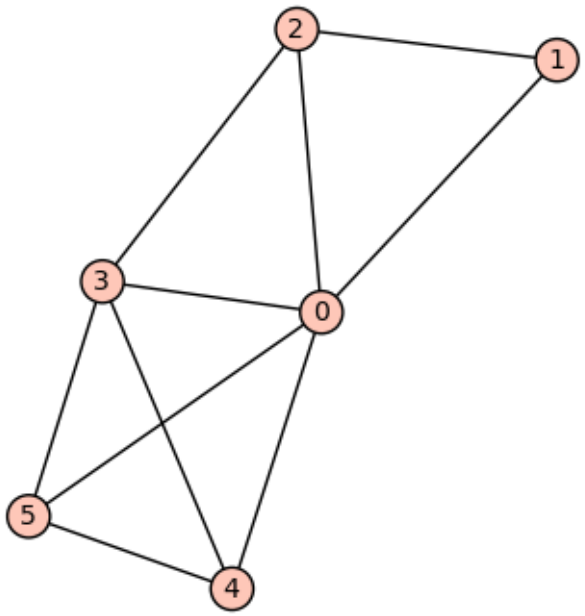


Figure 5.8: A randomly choosen graph whose "central-most" vertex is 0, with degree centrality equal to 5/6.

Algorithm 5.1: Friendship graph.

Input: A positive integer n.
Output: The friendship graph F_n.

if $n = 1$ **then**
    return C_3
$G \leftarrow$ null graph
$N \leftarrow 2n + 1$
for $i \leftarrow 0, 1, \ldots, N - 3$ **do**
    if i is odd **then**
        add edges $(i,\ i + 1)$ and $(i,\ N - 1)$ to G
    else
        add edge $(i, N - 1)$ to G
add edges $(N - 2,\ 0)$ and $(N - 2,\ N - 1)$ to G
return E

5.6 Network reliability

- Whitney synthesis
- Tutte's synthesis of 3-connected graphs
- Harary graphs
- constructing an optimal k-connected n-vertex graph

5.7 The spectrum of a graph

We use the notes "The spectrum of a graph" by Andries Brouwer [**?**] as a basic reference.

- Spectrum of a graph
- Laplacian spectrum of a graph
- Applications
- Examples from Boolean functions

Let $G = (V, E)$ be a (possibly directed) finite graph on $n = |V|$ vertices. The adjacency matrix of G is the $n \times n$ matrix $A = A(G) = (a_{v,w})_{v,w \in V}$ with rows and columns indexed by V and entries $a_{v,w}$ denoting the number of edges from v to w.

The *spectrum* of G, $\mathrm{spec}(G)$, is by definition the spectrum of A, that is, its multi-set of eigenvalues together with their multiplicities. The *characteristic polynomial* of G is that of A, that is, the polynomial p_A defined by $p_A(x) = \det(A - xI)$.

5.7.1 The Laplacian spectrum

Recall from section 1.3.3 that, given a simple graph G with n vertices $V = \{v_1, \ldots, v_n\}$, its (vertex) *Laplacian matrix* $L = (\ell_{i,j})_{n \times n}$ is defined as:

$$\ell_{i,j} = \begin{cases} \deg(v_i), & \text{if } i = j, \\ -1, & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j, \\ 0, & \text{otherwise.} \end{cases}$$

The *Laplacian spectrum* is by definition the spectrum of the vertex Laplacian of G, that is, its multi-set of eigenvalues together with their multiplicities.

For a graph G and its Laplacian matrix L with eigenvalues $\lambda_n \leq \lambda_{n-1} \leq \cdots \leq \lambda_1$:

- For all i, $\lambda_i \geq 0$ and $\lambda_n = 0$.
- The number of times 0 appears as an eigenvalue in the Laplacian is the number of connected components in the graph.
- $\lambda_n = 0$ because every Laplacian matrix has an eigenvector of $[1, 1, \ldots, 1]$,

- If we define a *signed edge adjacency matrix* M with element $m_{e,v}$ for edge $e \in E$ (connecting vertex v_i and v_j, with $i < j$) and vertex $v \in V$ given by

$$M_{ev} = \begin{cases} 1, & \text{if } v = v_i, \\ -1, & \text{if } v = v_j, \\ 0, & \text{otherwise} \end{cases}$$

  then the Laplacian matrix L satisifies $L = M^T M$, where M^T is the matrix transpose of M.

These are left as an exercise.

5.7.2 Applications of the (ordinary) spectrum

The following is a basic fact about the largest eigenvalue of a graph.

Theorem 5.24. *Each graph G has a real eigenvalue $\lambda_1 > 0$ with nonnegative real corresponding eigenvector, and such that for each eigenvalue λ we have $|\lambda| \leq \lambda_1$.*

We shall mostly be interested in the case where G is undirected, without loops or multiple edges. This means that A is symmetric, has zero diagonal ($a_{v,v} = 0$), and is a 0-1 matrix ($a_{v,w} \in \{0, 1\}$).

A number λ is eigenvalue of A if and only if it is a zero of the polynomial p_A. Since A is real and symmetric, all its eigenvalues are real and A is diagonalizable. In particular, for each eigenvalue, its algebraic multiplicity (that is, its multiplicity as a root of the characteristic polynomial) coincides with its geometric multiplicity (that is, the dimension of the corresponding eigenspace).

Theorem 5.25. *Let G be a connected graph of diameter d. Then G has at least $d + 1$ distinct eigenvalues.*

Proof. Let the distinct eigenvalues of the adjacency matrix A of G be $\lambda_1, ..., \lambda_r$. Then $(A - \lambda_1 I)...(A - \lambda_r I) = 0$, so that A^r is a linear combination of $I, A, ..., A^{r-1}$. But if the distance from the vertex $v \in V$ to the vertex $w \in V$ is r, then $(A^i)_{v,w} = 0$ for $0 \leq i \leq r - 1$ and $(A^r)_{v,w} > 0$, contradiction. Hence $d > r$. ■

5.7.3 Examples from Boolean functions

Let f be a Boolean function on $GF(2)^n$. The *Cayley graph of f* is defined to be the graph

$$\Gamma_f = (GF(2)^n, E_f),$$

whose vertex set is $GF(2)^n$ and the set of edges is defined by

$$E_f = \{(u, v) \in GF(2)^n \times GF(2)^n \mid f(u + v) = 1\}.$$

The adjacency matrix A_f of this graph is the matrix whose entries are

$$A_{i,j} = f(b(i) + b(j)),$$

where $b(k)$ is the binary representation of the integer k. Note Γ_f is a regular graph of degree $\mathrm{wt}(f)$, where wt denotes the Hamming weight of f when regarded as a vector of values (of length 2^n).

Recall that, given a graph Γ and its adjacency matrix A, the spectrum $\mathrm{Spec}(\Gamma)$ is the multi-set of eigenvalues of A.

The *Walsh transform* of a Boolean function f is an integer-valued function over $GF(2)^n$ that can be defined as

$$W_f(u) = \sum_{x in GF(2)^n} (-1)^{f(x)+\langle u,x\rangle}.$$

A Boolean function f is *bent* if $|W_f(a)| = 2^{n/2}$ (this only makes sense if n is even). This property says, roughly speaking, that f is "as non-linear as possible." The *Hadamard transform* of a integer-valued function f is an integer-valued function over $GF(2)^n$ that can be defined as

$$H_f(u) = \sum_{x in GF(2)^n} f(x)(-1)^{\langle u,x\rangle}.$$

It turns out that the spectrum of Γ_f is equal to the Hadamard transform of f when regarded as a vector of (integer) $0, 1$-values. (This nice fact seems to have first appeared in [**?**], [**?**].)

Recall that a graph is regular of degree r (or r-regular) if every vertex has degree r. We say that an r-regular graph Γ is a *strongly regular graph with parameters* (v, r, d, e) (for nonnegative integers e, d) provided, for all vertices u, v the number of vertices adjacent to both u, v is equal to

$$\begin{cases} e, & \text{if } u, v \text{ are adjacent,} \\ d, & \text{if } u, v \text{ are nonadjacent.} \end{cases}$$

It turns out tht f is bent if and only if Γ_f is strongly regular and $e = d$ (see [**?**], [**?**]).

The following Sage computations illustrate these and other theorems in [**?**], [**?**], [**?**], [**?**].

First, consider the Boolean function $f : GF(2)^4 \to GF(2)$ given by $f(x_0, x_1, x_2) = x_0x_1 + x_2x_3$.

```
sage: V = GF(2)^4
sage: f = lambda x: x[0]*x[1]+x[2]*x[3]
sage: CartesianProduct(range(16), range(16))
Cartesian product of [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15],
                     [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: C = CartesianProduct(range(16), range(16))
sage: Vlist = V.list()
sage: E = [(x[0],x[1]) for x in C if f(Vlist[x[0]]+Vlist[x[1]])==1]
sage: len(E)
96
sage: E = Set([Set(s) for s in E])
sage: E = [tuple(s) for s in E]
sage: Gamma = Graph(E)
sage: Gamma
Graph on 16 vertices
sage: VG = Gamma.vertices()
sage: L1 = []
sage: L2 = []
sage: for v1 in VG:
....:         for v2 in VG:
....:             N1 = Gamma.neighbors(v1)
....:         N2 = Gamma.neighbors(v2)
....:         if v1 in N2:
....:                 L1 = L1+[len([x for x in N1 if x in N2])]
....:         if not(v1 in N2) and v1!=v2:
```

```
....:                   L2 = L2+[len([x for x in N1 if x in N2])]
....:
....:
sage: L1; L2
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2]
```

This implies the graph is strongly regular with $d = e = 2$. Let us use Sage to determine some properties of this graph.

```
sage: Gamma.spectrum()
[6, 2, 2, 2, 2, 2, 2, -2, -2, -2, -2, -2, -2, -2, -2, -2]
sage: [walsh_transform(f, a) for a in V]
[4, 4, 4, -4, 4, 4, 4, -4, 4, 4, 4, -4, -4, -4, -4, 4]
sage: Omega_f = [v for v in V if f(v)==1]
sage: len(Omega_f)
6
sage: Gamma.is_bipartite()
False
sage: Gamma.is_hamiltonian()
True
sage: Gamma.is_planar()
False
sage: Gamma.is_regular()
True
sage: Gamma.is_eulerian()
True
sage: Gamma.is_connected()
True
sage: Gamma.is_triangle_free()
False
sage: Gamma.diameter()
2
sage: Gamma.degree_sequence()
[6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
```

The picture of the graph is given in Figure 5.9.

```
sage: H = matrix(QQ, 16, 16, [(-1)^(Vlist[x[0]]).dot_product(Vlist[x[1]]) for x in C])
sage: H
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1]
[ 1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1]
[ 1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1]
[ 1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1  1 -1]
[ 1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1]
[ 1 -1  1 -1  1 -1  1 -1 -1  1 -1  1 -1  1 -1  1]
[ 1  1 -1 -1  1  1 -1 -1 -1 -1  1  1 -1 -1  1  1]
[ 1 -1 -1  1  1 -1 -1  1 -1  1  1 -1 -1  1  1 -1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1]
[ 1 -1  1 -1 -1  1 -1  1 -1  1 -1  1  1 -1  1 -1]
[ 1  1 -1 -1 -1 -1  1  1 -1 -1  1  1  1  1 -1 -1]
[ 1 -1 -1  1 -1  1  1 -1 -1  1  1 -1  1 -1 -1  1]
sage: flist = vector(QQ, [int(f(v)) for v in V])
sage: H*flist
(6, -2, -2, 2, -2, -2, -2, 2, -2, -2, -2, 2, 2, 2, 2, -2)
sage: A = matrix(QQ, 16, 16, [f(Vlist[x[0]]+Vlist[x[1]]) for x in C])
sage: A.eigenvalues()
[6, 2, 2, 2, 2, 2, 2, -2, -2, -2, -2, -2, -2, -2, -2, -2]
```

The Hadamard transform of the Boolean function does indeed determine the spectrum of its Cayley graph.

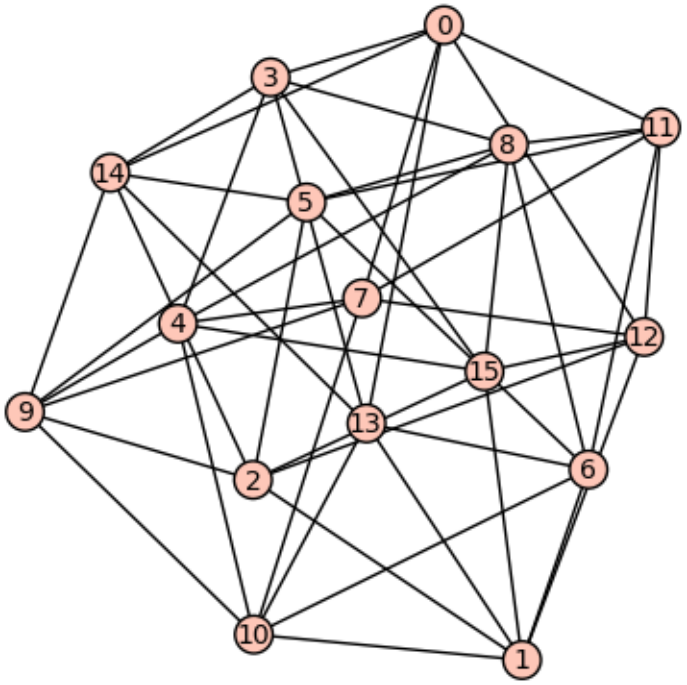


Figure 5.9: A Cayley graph of a Boolean function.

Here is another example: $f : GF(2)^3 \to GF(2)$ given by $f(x_0, x_1, x_2) = x_0x_1 + x_2$.

```
sage: V = GF(2)^3
sage: f = lambda x: x[0]*x[1]+x[2]
sage: Omega_f = [v for v in V if f(v)==1]
sage: len(Omega_f)
4
sage: C = CartesianProduct(range(8), range(8))
sage: Vlist = V.list()
sage: E = [(x[0],x[1]) for x in C if f(Vlist[x[0]]+Vlist[x[1]])==1]
sage: E = Set([Set(s) for s in E])
sage: E = [tuple(s) for s in E]
sage: Gamma = Graph(E)
sage: Gamma
Graph on 8 vertices
sage:
sage: VG = Gamma.vertices()
sage: L1 = []
sage: L2 = []
sage: for v1 in VG:
....:         for v2 in VG:
....:             N1 = Gamma.neighbors(v1)
....:         N2 = Gamma.neighbors(v2)
....:         if v1 in N2:
....:                 L1 = L1+[len([x for x in N1 if x in N2])]
....:         if not(v1 in N2) and v1!=v2:
....:                 L2 = L2+[len([x for x in N1 if x in N2])]
....:
sage: L1; L2
[2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 2, 0, 0, 2, 2, 2,
 2, 0, 2, 2, 2, 0, 2, 2, 2, 2, 0, 2]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

This implies that the graph is not strongly regular, therefore f is not bent. (There are other reasons why f cannot be bent as well.) Again, let us use Sage to determine some properties of this graph.

```
sage: Gamma.spectrum()
[4, 2, 0, 0, 0, -2, -2, -2]
sage:
```

```
sage: Gamma.is_bipartite()
False
sage: Gamma.is_hamiltonian()
True
sage: Gamma.is_planar()
False
sage: Gamma.is_regular()
True
sage: Gamma.is_eulerian()
True
sage: Gamma.is_connected()
True
sage: Gamma.is_triangle_free()
False
sage: Gamma.diameter()
2
sage: Gamma.degree_sequence()
[4, 4, 4, 4, 4, 4, 4, 4]
sage: H = matrix(QQ, 8, 8, [(-1)^(Vlist[x[0]]).dot_product(Vlist[x[1]]) for x in C])
sage: H
[ 1  1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1 -1]
[ 1  1 -1 -1  1  1 -1 -1]
[ 1 -1 -1  1  1 -1 -1  1]
[ 1  1  1  1 -1 -1 -1 -1]
[ 1 -1  1 -1 -1  1 -1  1]
[ 1  1 -1 -1 -1 -1  1  1]
[ 1 -1 -1  1 -1  1  1 -1]
sage: flist = vector(QQ, [int(f(v)) for v in V])
sage: H*flist
(4, 0, 0, 0, -2, -2, -2, 2)
sage: Gamma.spectrum()
[4, 2, 0, 0, 0, -2, -2, -2]
sage: A = matrix(QQ, 8, 8, [f(Vlist[x[0]]+Vlist[x[1]]) for x in C])
sage: A.eigenvalues()
[4, 2, 0, 0, 0, -2, -2, -2]
```

Again, we see the Hadamard transform does indeed determine the graph spectrum. The picture of the graph is given in Figure 5.10.

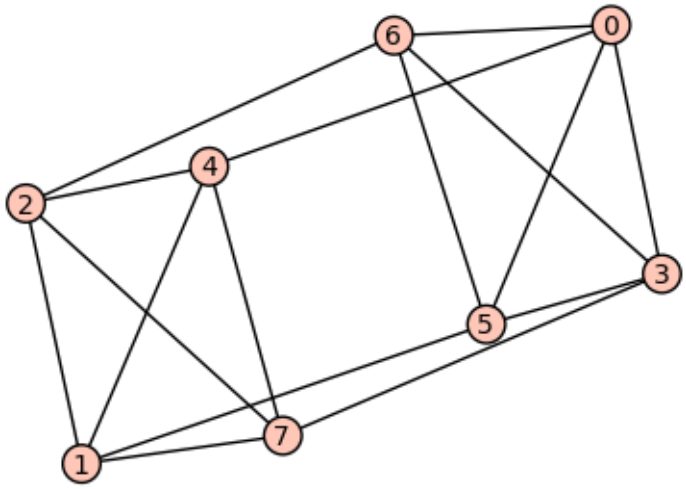


Figure 5.10: Another Cayley graph of a Boolean function.

5.8 Expander graphs and Ramanujan graphs

In combinatorics, an expander graph is a sparse graph that has strong connectivity properties. Expander graphs have many applications - for example, to cryptography, and the theory of error-correcting codes.

The *edge expansion* $h(G)$ of a graph $G = (V, E)$ is defined as

$$h(G) = \min_{0<|S|\leq\frac{|V|}{2}} \frac{|\partial(S)|}{|S|},$$

where the minimum is over all nonempty sets S of at most $|V|/2$ vertices and $\partial(S)$ is the edge boundary of S, i.e., the set of edges with exactly one endpoint in S.

The *vertex expansion* (or *vertex isoperimetric number*) $h_{out}(G)$ of a graph G is defined as

$$h_{\text{out}}(G) = \min_{0<|S|\leq\frac{|V|}{2}} \frac{|\partial_{\text{out}}(S)|}{|S|},$$

where $\partial_{\text{out}}(S)$ is the outer boundary of S, i.e., the set of vertices in $V(G) \setminus S$ with at least one neighbor in S.

```
sage: G = PSL(2, 5)
sage: X = G.cayley_graph()
sage: V = X.vertices()
sage: S = [V[1], V[3], V[7], V[10], V[13], V[ 14], V[23]]
sage: delS = X.edge_boundary(S)
sage: edge_expan_XS = len(delS)/len(S); RR(edge_expan_XS)
1.00000000000000
sage: S = [V[1], V[3], V[7], V[12], V[24], V[37]]
sage: delS = X.edge_boundary(S)
sage: edge_expan_XS = len(delS)/len(S); RR(edge_expan_XS)
1.50000000000000
sage: S = [V[2], V[8], V[13], V[27], V[32], V[44], V[57]]
sage: delS = X.edge_boundary(S)
sage: edge_expan_XS = len(delS)/len(S); RR(edge_expan_XS)
1.42857142857143
sage: S = [V[0], V[6], V[11], V[16], V[21], V[29], V[35], V[45],V[53]]
sage: delS = X.edge_boundary(S)
sage: edge_expan_XS = len(delS)/len(S); RR(edge_expan_XS)
1.77777777777778
sage: n = len(X.vertices())
sage: J = range(n)
sage: J30 = Subsets(J, int(n/2))
sage: K = J30.random_element()
sage: K
{0, 2, 3, 4, 5, 6, 8, 9, 11, 13, 16, 18, 19, 21, 24, 25, 26, 28, 29,
 30, 36, 37, 38, 40, 42, 45, 46, 49, 53, 57}
sage: S = [V[i] for i in K]                    #  30 vertices, randomly selected
sage: delS = [v for v in V if min([X.distance(a,v) for a in S]) == 1]
sage: RR(len(delS))/RR(len(S))
0.800000000000000
```

A family $\mathcal{G} = \{G_1, G_2, \ldots\}$ of d-regular graphs is an *edge expander family* if there is a constant $c > 0$ such that $h(G) \geq c$ for each $G \in \mathcal{G}$. A vertex expander family is defined similarly, using $h_{out}(G)$ instead.

5.8.1 Ramanujan graphs

Let G be a connected d-regular graph with n vertices, and let $\lambda_0 \geq \lambda_1 \geq \ldots \geq \lambda_{n-1}$ be the eigenvalues of the adjacency matrix of G. Because G is connected and d-regular, its eigenvalues satisfy $d = \lambda_0 > \lambda_1 \geq \ldots \geq \lambda_{n-1} \geq -d$. Whenever there exists λ_i with $|\lambda_i| < d$, define

$$\lambda(G) = \max_{|\lambda_i|<d} |\lambda_i|.$$

A d-regular graph G is a *Ramanujan graph* if $\lambda(G)$ is defined and $\lambda(G) \leq 2\sqrt{d-1}$.

Let q be a prime power such that $q \equiv 1 \pmod 4$. Note that this implies that the finite field $GF(q)$ contains a square root of -1.

Now let $V = GF(q)$ and $E = \{\{a, b\} \in GF(q) \times GF(q) \mid (a - b) \in GF(q)^{\times})^2\}$. This set is well defined since $a - b = (-1) \cdot (b - a)$, and since -1 is a square, it follows that $a-$b is a square if and only if $b - a$ is a square.

By definition G = (V, E) is the *Paley graph* of order q.

The following facts are known about Paley graphs.

- The eigenvalues of Paley graphs are $\frac{q-1}{2}$ (with multiplicity 1) and $\frac{-1\pm\sqrt{q}}{2}$ (both with multiplicity $\frac{q-1}{2}$).
- It is known that a Paley graph is a Ramanujan graph.
- It is known that the family of Paley graphs of prime order is a vertex expander graph family.
- If $q = p^r$, where p is prime, then $Aut(G)$ has order $q(q-1)/2$.

Here is Sage code for the Paley graph[1]:

```
def Paley(q):
    K.<a> = GF(q)
    return Graph([K, lambda i,j: i != j and (i-j).is_square()])
```

Below is an example.

```
sage: X = Paley(13)
sage: X.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: X.is_vertex_transitive()
True
sage: X.degree_sequence()
[6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
sage: X.spectrum()
[6, 1.302775637731995?, 1.302775637731995?, 1.302775637731995?,
1.302775637731995?, 1.302775637731995?, 1.302775637731995?,
-2.302775637731995?, -2.302775637731995?, -2.302775637731995?,
-2.302775637731995?, -2.302775637731995?, -2.302775637731995?]
sage: G = X.automorphism_group()
sage: G.cardinality()
78
sage: 13*12/2
78
```

5.9 Problems

> When you don't share your problems, you resent hearing the problems of other people.
> — Chuck Palahniuk, *Invisible Monsters*, 1999

5.1. Let $G = (V, E)$ be an undirected, unweighted simple graph. Show that V and the distance function on G form a metric space if and only if G is connected.

5.2. Let u and v be two distinct vertices in the same connected component of G. If P is a u-v path such that $d(u, v) = \epsilon(u)$, we say that P is an *eccentricity path* for u.

(a) If r is the root of a tree, show that the end-vertex of an eccentricity path for r is a leaf.

[1]Thanks to Chris Godsil; see [?].

(b) If v is a vertex of a tree distinct from the root r, show that any eccentricity path for v must contain r or provide an example to the contrary.

(c) A vertex w is said to be an *eccentric vertex* of v if $d(v, w) = \epsilon(v)$. Intuitively, an eccentric vertex of v can be considered as being as far away from v as possible. If w is an eccentric vertex of v and vice versa, then v and w are said to be *mutually eccentric*. See Buckley and Lau [**?**] for detailed discussions of mutual eccentricity. If w is an eccentric vertex of v, explain why v is also an eccentric vertex of w or show that this does not in general hold.

5.3. If u and v are vertices of a connected graph G such that $d(u, v) = \text{diam}(G)$, show that u and v are mutually eccentric.

5.4. If uv is an edge of a tree T and w is a vertex of T distinct from u and v, show that $|d(u, w) - d(w, v)| = W(uv)$ with $W(uv)$ being the weight of uv.

5.5. If u and v are vertices of a tree T such that $d(u, v) = \text{diam}(T)$, show that u and v are leaves.

5.6. Let $v_1, v_2, \ldots, v_k$ be the leaves of a tree T. Show that $\text{per}(T) = \{v_1, v_2, \ldots, v_k\}$.

5.7. Show that all the eccentric vertices of a tree are leaves.

5.8. If G is a connected graph, show that $\text{rad}(G) \leq \text{diam}(G) \leq 2 \cdot \text{rad}(G)$.

5.9. Let T be a tree of order ≥ 3. If the center of T has one vertex, show that $\text{diam}(T) = 2 \cdot \text{rad}(T)$. If the center of T has two vertices, show that $\text{diam}(T) = 2 \cdot \text{rad}(T) - 1$.

5.10. Let $G = (V, E)$ be a simple undirected, connected graph. Define the distance of a vertex $v \in V$ by

$$d(v) = \sum_{x \in V} d(v, x)$$

and define the distance of the graph G itself by

$$d(G) = \frac{1}{2} \sum_{v \in V} d(v).$$

For any vertex $v \in V$, show that $d(G) \leq d(v) + d(G - v)$ with $G - v$ being a vertex deletion subgraph of G. This result appeared in Entringer et al. [**?**, p.284].

5.11. Determine the sequence of distance matrices for the graphs in Figure 5.4.

5.12. If $G = (V, E)$ is an undirected connected graph and $v \in V$, prove the following vertex connectivity inequality:

$$\kappa(G) - 1 \leq \kappa(G - v) \leq \kappa(G).$$

5.13. If $G = (V, E)$ is an undirected connected graph and $e \in E$, prove the following edge connectivity inequality:

$$\lambda(G) - 1 \leq \lambda(G - e) \leq \lambda(G).$$

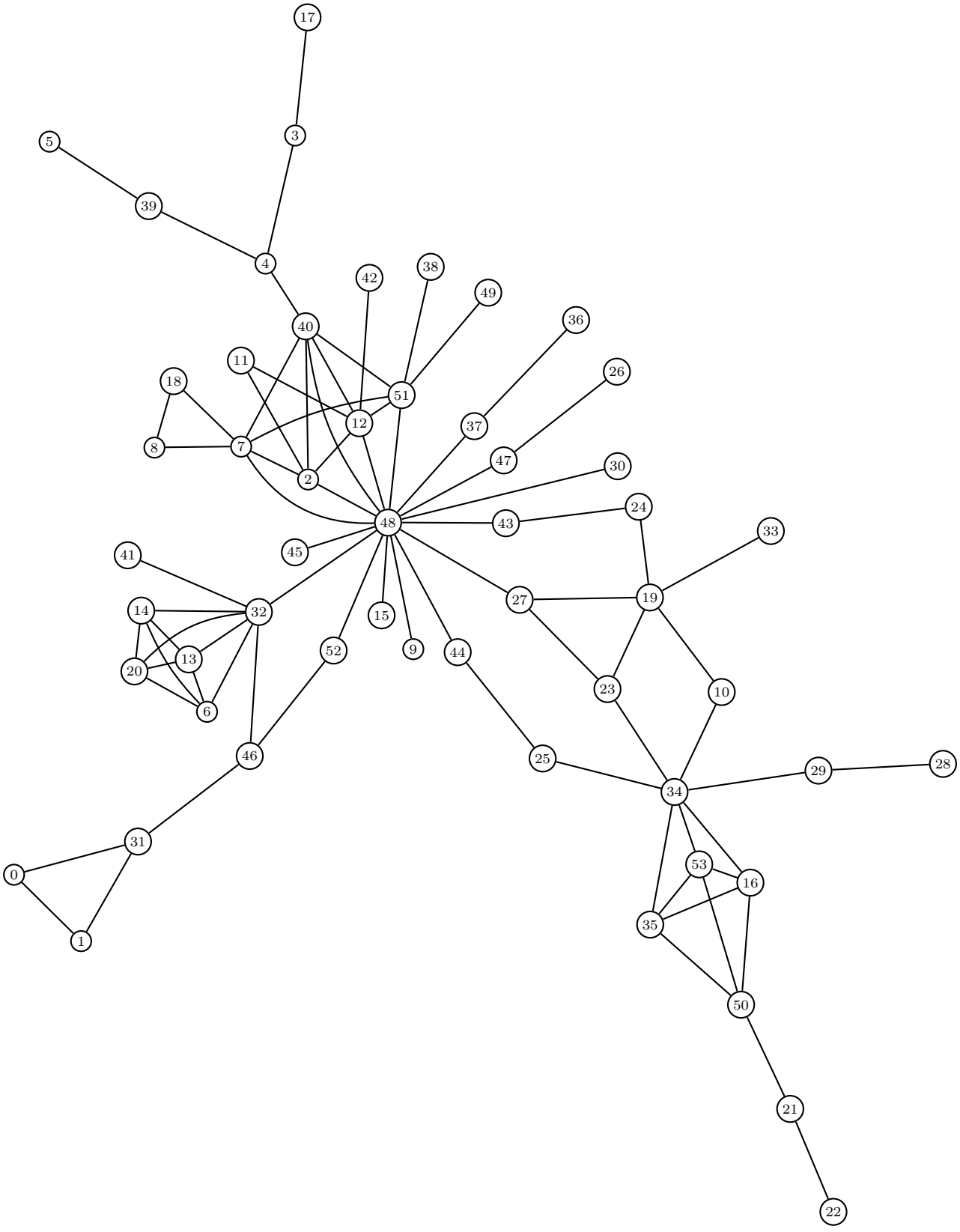


Figure 5.11: Network of common grape cultivars.

code	name	code	name	code	name
0	Alicante Bouschet	1	Aramon	2	Bequignol
3	Cabernet Franc	4	Cabernet Sauvignon	5	Carignan
6	Chardonnay	7	Chenin Blanc	8	Colombard
9	Donzillinho	10	Ehrenfelser	11	Fer Servadou
12	Flora	13	Gamay	14	Gelber Ortlieber
15	Grüner Veltliner	16	Kemer	17	Merlot
18	Meslier-Saint-Francois	19	Müller-Thurgau	20	Muscat Blanc
21	Muscat Hamburg	22	Muscat of Alexandria	23	Optima
24	Ortega	25	Osteiner	26	Péagudo
27	Perle	28	Perle de Csaba	29	Perlriesling
30	Petit Manseng	31	Petite Bouschet	32	Pinot Noir
33	Reichensteiner	34	Riesling	35	Rotberger
36	Roter Veltliner	37	Rotgipfler	38	Royalty
39	Ruby Cabernet	40	Sauvignon Blanc	41	Schönburger
42	Semillon	43	Siegerrebe	44	Sylvaner
45	Taminga	46	Teinturier du Cher	47	Tinta Madeira
48	Traminer	49	Trincadeiro	50	Trollinger
51	Trousseau	52	Verdelho	53	Wittberger

Table 5.2: Numeric code and actual name of common grape cultivars.

5.14. Figure 5.11 depicts how common grape cultivars are related to one another; the graph is adapted from Myles et al. [**?**]. The numeric code of each vertex can be interpreted according to Table 5.2. Compute various distance and connectivity measures for the graph in Figure 5.11.

5.15. Prove the characterizations of 2-connected graphs as stated in Theorem 5.13.

5.16. Let $G = (V, E)$ be an undirected connected graph of order n and suppose that $\deg(v) \geq (n + k - 2)/2$ for all $v \in V$ and some fixed positive integer k. Show that G is k-connected.

5.17. A vertex (or edge) separating set S of a connected graph G is *minimum* if S has the smallest cardinality among all vertex (respectively edge) separating sets in G. Similarly S is said to be *maximum* if it has the greatest cardinality among all vertex (respectively edge) separating sets in G. For the graph in Figure 5.7(a), determine the following:

(a) A minimum vertex separating set.

(b) A minimum edge separating set.

(c) A maximum vertex separating set.

(d) A maximum edge separating set.

(e) The number of minimum vertex separating sets.

(f) The number of minimum edge separating sets.

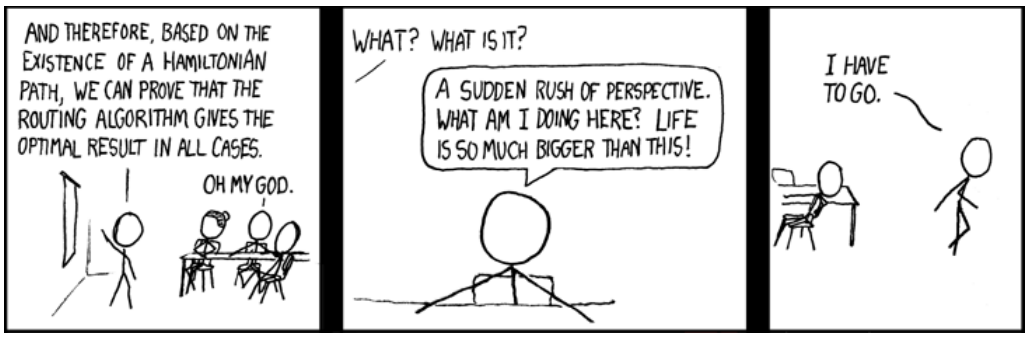
Chapter 6

Optimal graph traversals

6.1 Eulerian graphs

- Motivation: tracing out all the edges of a graph without lifting your pencil.
- multigraphs and simple graphs
- Eulerian tours
- Eulerian trails

6.2 Hamiltonian graphs



— Randall Munroe, xkcd, http://xkcd.com/230/

- Motivation: the eager tourist problem: visiting all major sites of a city in the least time/distance.
- Hamiltonian paths (or cycles)
- Hamiltonian graphs

Theorem 6.1. ***Ore 1960.*** *Let G be a simple graph with $n \geq 3$ vertices. If $\deg(u) + \deg(v) \geq n$ for each pair of non-adjacent vertices $u, v \in V(G)$, then G is Hamiltonian.*

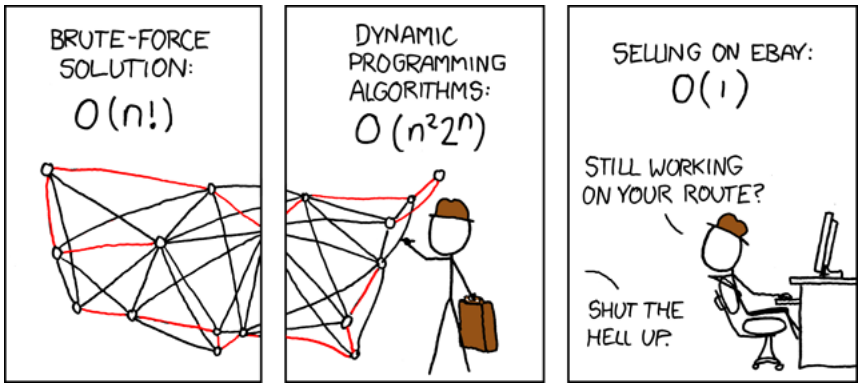
Corollary 6.2. ***Dirac 1952.*** *Let G be a simple graph with $n \geq 3$ vertices. If $\deg(v) \geq n/2$ for all $v \in V(G)$, then G is Hamiltonian.*

6.3 The Chinese Postman Problem

See section 6.2 of Gross and Yellen [**?**].

- de Bruijn sequences
- de Bruijn digraphs
- constructing a $(2, n)$-de Bruijn sequence
- postman tours and optimal postman tours
- constructing an optimal postman tour

6.4 The Traveling Salesman Problem



— Randall Munroe, xkcd, http://xkcd.com/399/

See section 6.4 of Gross and Yellen [**?**], and section 35.2 of Cormen et al. [**?**].

- Gray codes and n-dimensional hypercubes
- the Traveling Salesman Problem (TSP)
- nearest neighbor heuristic for TSP
- some other heuristics for solving TSP

Chapter 7

Planar graphs

A *planar graph* is a graph that can be drawn on a sheet of paper without any overlapping between its edges.

It is a property of many "natural" graphs drawn on the earth's surface, like for instance the graph of roads, or the graph of internet fibers. It is also a necessary property of graphs we want to build, like VLSI layouts.

Of course, the property of being planar does not prevent one from finding a drawing with many overlapping between edges, as this property only asserts that there exists a drawing (or *embedding*) of the graph avoiding it. Planarity can be characterized in many different ways, one of the most satiating being Kuratowski's theorem.

See chapter 9 of Gross and Yellen [**?**].

7.1 Planarity and Euler's Formula

- planarity, non-planarity, planar and plane graphs
- crossing numbers

Theorem 7.1. *The complete bipartite graph $K_{3,n}$ is non-planar for $n \geq 3$.*

Theorem 7.2. ***Euler's Formula.*** *Let G be a connected plane graph having n vertices, e edges and f faces. Then $n - e + f = 2$.*

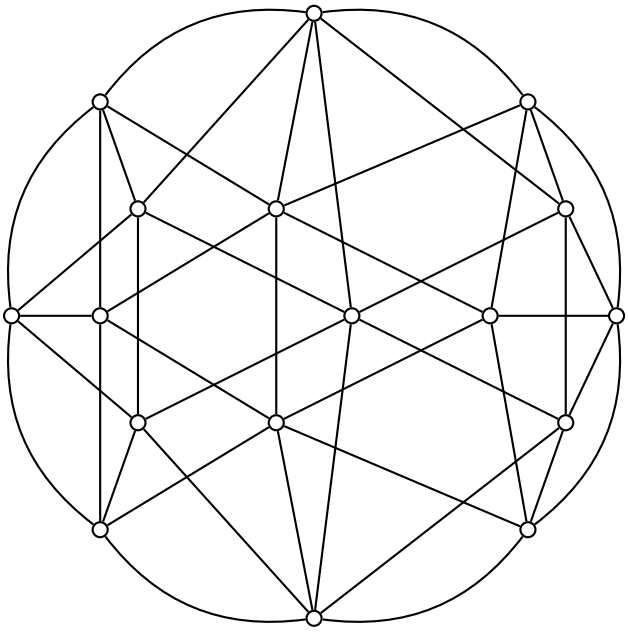
7.2 Kuratowski's Theorem

- Kuratowski graphs

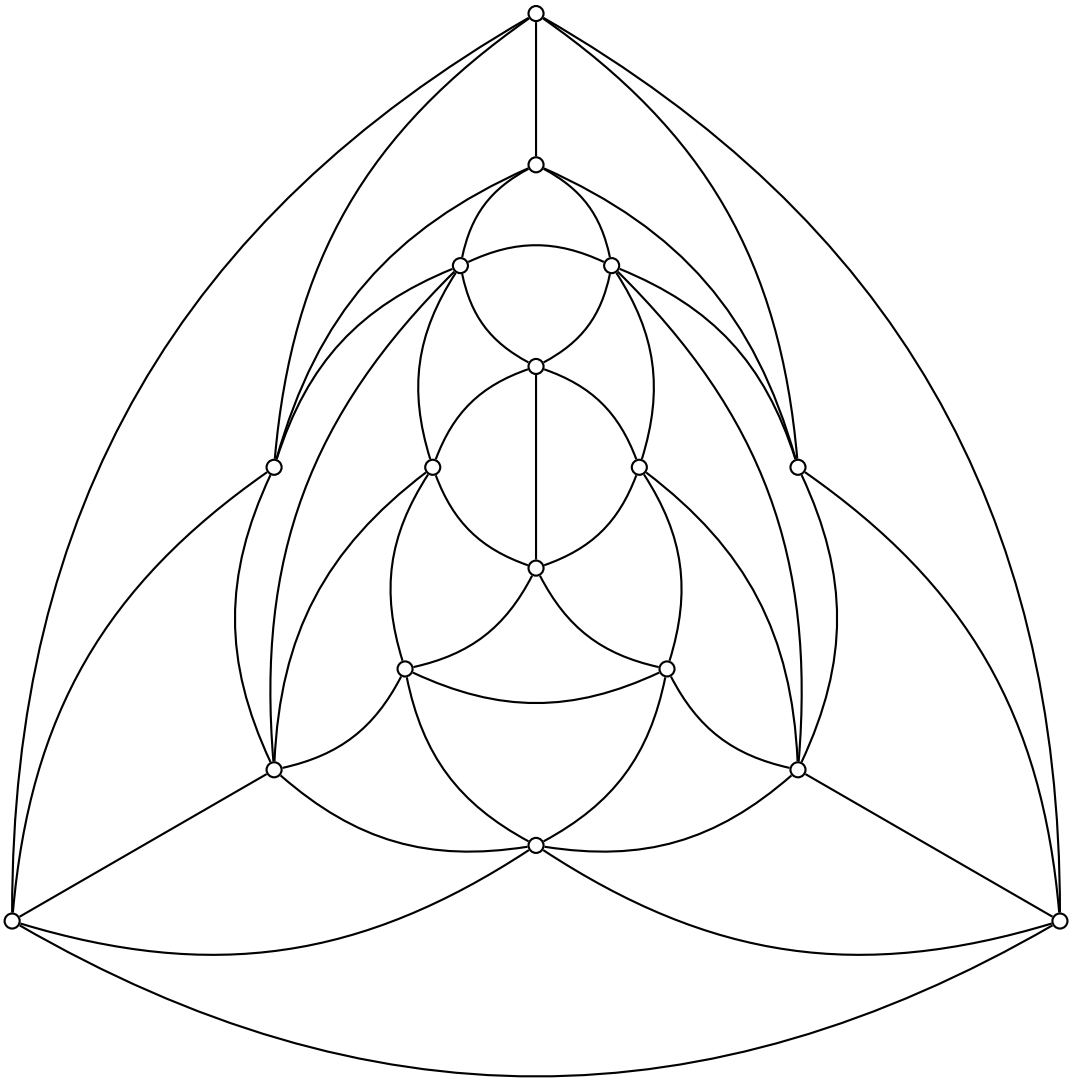
The objective of this section is to prove the following theorem.

Theorem 7.3. *[?]* **Kuratowski's Theorem.** *A graph is planar if and only if it contains no subgraph homeomorphic to a subdivision of K_5 or $K_{3,3}$.*

Graph Minors : The reader may find interesting to notice that the previous result, first proved in 1930 as purely topological (there is no mention of graphs in Kuratowski's original paper), can be seen as a very special case of the Graph Minor Theorem (Thm1.37).



(a) Errera graph.



(b) Planar representation.

Figure 7.1: The Errera graph is planar.

It can easily be seen that if a graph G is planar, any of its subgraph is also planar. Besides, planarity is still preserved under edge contraction. These two facts mean together that any minor of a planar graph is still planar graph, which makes of planarity a *minor-closed* property. If we let $\bar{\mathcal{P}}$ denote the poset of all non-planar graph, ordered with the minor partial order, we can now consider the set $\bar{\mathcal{P}}_{min}$ of its minimal elements which, by the Graph Minor Theorem, is a finite set.

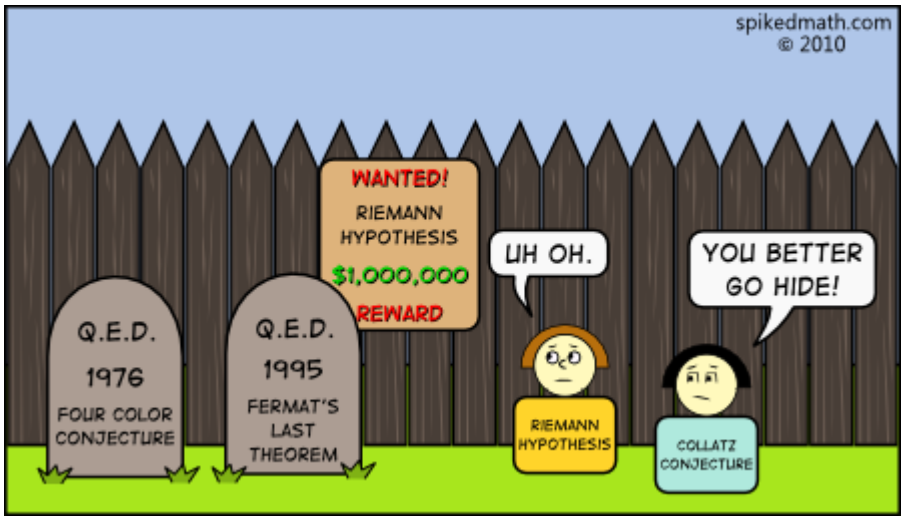
Actually, Kuratowski's theorem asserts that $\bar{\mathcal{P}}_{min} = \{K_5, K_{3,3}\}$.

7.3 Planarity algorithms

- planarity testing for 2-connected graphs
- planarity testing algorithm of Hopcroft and Tarjan [**?**]
- planarity testing algorithm of Boyer and Myrvold [**?**]

Chapter 8

Graph coloring



— Spiked Math, http://spikedmath.com/210.html

- See Jensen and Toft [**?**] for a survey of graph coloring problems.
- See Dyer and Frieze [**?**] for an algorithm on randomly colouring random graphs.

Graph coloring problems originated with the coloring of maps. For example, regard each state in the United States as a vertex, and connect two vertices by an edge if and only if they share a boundary, i.e., are neighbors. If you can color the United States map using k colors in such a way that no two neighboring states have the same color then we say the map has a k-coloring. While a student in London in the mid-1800's, the South African mathematician Francis Guthrie conjectured to his mathematics professor Augustus de Morgan that four colors suffice to color any map. It was an open problem for over 100 years (only proven by Appel and Haken in 1976).

8.1 Vertex coloring

When used without any qualification, a *coloring* of an undirected graph $G = (V, E)$ is intended to mean a vertex coloring, namely a labelling of the graph's vertices with colors such that no two vertices sharing an edge have the same color.

A coloring using at most k colors is called a (proper) *k-coloring*. For example, the 2-colorable graphs are exactly the bipartite graphs.

Remark 8.1. *For each $k = 3, 4, \ldots$, the corresponding decision problem of deciding if a given graph can be k-colored is NP-hard (see [?]).*

The smallest number of colors needed to color a graph G is called its (vertex) *chromatic number*, and is here denoted $\chi_v(G)$. A subset of V assigned to the same color is called a *color-class*. A subset S of V is called an *independent set* if no two vertices in S are adjacent in G. By definition, every color-class forms an independent set. The set of t-colorings is in one-to-one correspondence with the partitions of V into t independent sets.

Example 8.2. The Dyck graph, shown in Figure 8.1, is named after Walther von Dyck. It is a 3-regular graph with 32 vertices and 48 edges. The graph is Hamiltonian with 120 distinct Hamiltonian cycles. It has chromatic number 2 (in other words, is bipartite), chromatic index 3, radius 5, diameter 5 and girth 6. It is also a 3-vertex-connected and a 3-edge-connected graph. ■

```
sage: G = graphs.LCFGraph(32, [5,-5,13,-13], 8)
sage: G.is_bipartite()
True
sage: G.coloring()
[[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31],
 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]]
sage: G.is_vertex_transitive()
True
```

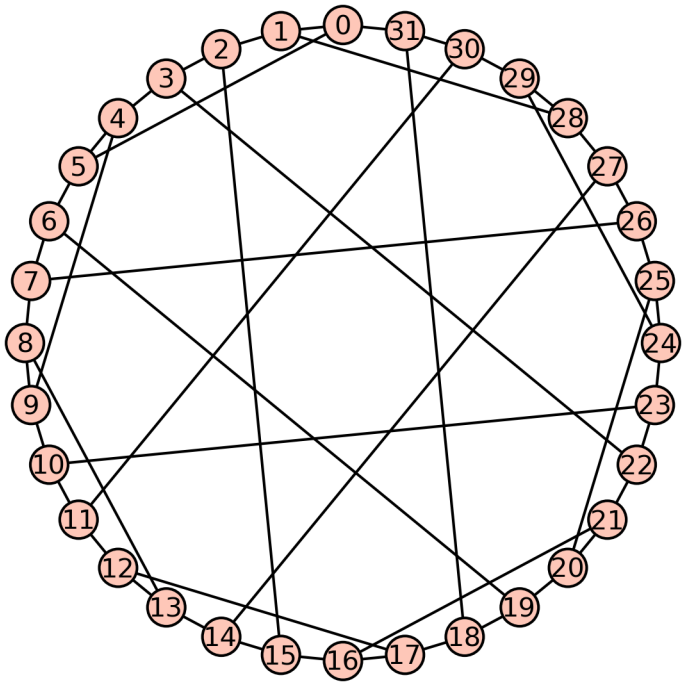


Figure 8.1: A Dyck graph example.

A *clique* in G is a subset of the vertex set $S \subset V$, such that for every two vertices in S, there exists an edge connecting the two. The *clique number* $\omega(G)$ of a graph G is the number of vertices in a maximal clique (a clique which cannot be extended to a clique of larger size by adding a vertex to it) in G. From the definitions, we see that the chromatic number is at least the clique number:

$$\chi_v(G) \geq \omega(G).$$

It is also not hard to give a non-trivial upper bound.

Theorem 8.3. *Every simple graph can be colored with one more color than the maximum vertex degree,*

$$\chi_v(G) \leq \Delta(G) + 1.$$

We give two proofs.

Proof. (proof 1) We prove this by induction on the number n of vertices.

It is obvious in the case $n = 1$.

Assume the result is true for all graphs with $k - 1$ vertices and let $G = (V, E)$ be a graph with k vertices. We want to chow that G has a coloring with $1 + \Delta(G)$ colors. Let $v \in V$ and consider the graph $G - v$. By hypothesis, this graph has a coloring with $1 + \Delta(G - v)$ colors. Since there are at most $\Delta(G)$ neighbors of v, no more than $\Delta(G)$ colors could be used for these adjacent vertices. There are two cases.

Case $1 + \Delta(G - v) < 1 + \Delta(G)$. In this case, we create a new color for v to obtain a coloring for G with $2 + \Delta(G - v) \leq 1 + \Delta(G)$ colors.

Case $1 + \Delta(G - v) = 1 + \Delta(G)$. In this case, the vertices adjacent to v have been colored with at most $\Delta(G)$ colors, leaving us with at least one unused color. We can use that color for v. ■

Proof. (proof 2) The fact that the graph coloring (decision) problem is NP-complete must not prevent one from trying to color it greedily. One such method would be to iteratively pick, in a graph G, an uncolored vertex v, and to color it with the smallest color available which is not yet used by one of its neighbors. Such a coloring algorithm will never use more than $\Delta(G) + 1$ different colors (where $\Delta(G)$ is the maximal degree of G), as no vertex in the procedure will ever exclude more than $\Delta(G)$ colors. ■

Such a greedy algorithm can be written in Sage in a few lines:

```
sage: g = graphs.RandomGNP(100,5/100)
sage: C = Set(xrange(100))
sage: color = {}
sage: for u in g:
...       interdits = Set([color[v] for v in g.neighbors(u) if color.has_key(v)])
...       color[u] = min(C-interdits)
```

Example 8.4. A Frucht graph, shown in Figure 8.2, has 12 nodes and 18 edges. It is 3-regular, planar and Hamiltonian. It is named after Robert Frucht. The Frucht graph has no nontrivial symmetries. It has chromatic number 3, chromatic index 3, radius 3, diameter 4 and girth 3. It is 3-vertex-connected and 3-edge-connected graph. ■

```
sage: G = graphs.FruchtGraph()
sage: G.show(dpi=300)
sage: vc = G.coloring()
sage: G.chromatic_number()
3
sage: d = {'blue':vc[0], 'red':vc[1], 'green':vc[2]}
sage: G.show(vertex_colors=d)
sage: G.automorphism_group().order()
1
```

- Brook's Theorem

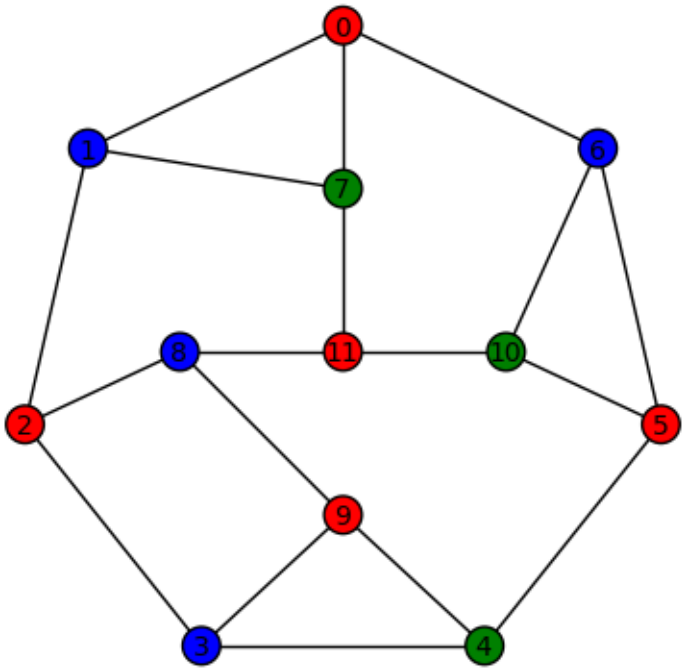


Figure 8.2: Frucht graph vertex-coloring example.

- heuristics for vertex coloring

Theorem 8.5. *(Brooks' inequality) If G is not a complete graph and is not an odd cycle graph then*

$$\chi_v(G) \leq \Delta(G).$$

8.2 Edge coloring

Edge coloring is the direct application of vertex coloring to the line graph of a graph G. (Recall, $L(G)$ is the graph whose vertices are the edges of G, two vertices being adjacent if and only if their corresponding edges share an endpoint). We write $\chi_v(L(G)) = \chi_e(G)$ for the *chromatic index* of G. (This is also called the *edge chromatic number*. An *edge coloring* of a graph is an assignment of colors to edges so that no vertex is incident to two edges of the same color. An edge coloring with k colors is called a *k-edge-coloring*.

Example 8.6. The Heawood graph, shown in Figure 8.4, is named after Percy John Heawood. It is an undirected graph with 14 vertices and 21 edges. It is a 3-regular, distance-transitive, distance-regular graph. It has chromatic number 2 and chromatic index 3. An edge-coloring is shown in Figure 8.3. A vertex-coloring is shown in Figure 8.4.

Recall that a graph is vertex-transitive if its automorphism group acts transitively upon its vertices. The automorphism group of the Heawood graph is isomorphic to the projective linear group PGL 2(7), a group of order 336. It acts transitively on the vertices and on the edges of the graph. Therefore, this graph is vertex-transitive. ■

```
sage: G = graphs.HeawoodGraph()
sage: ec = edge_coloring(G)
sage: d = {'blue':ec[0], 'green':ec[1], 'red':ec[2]}
sage: G.show(edge_colors = d)
sage: G.line_graph().chromatic_number()  # chromatic index
3
```

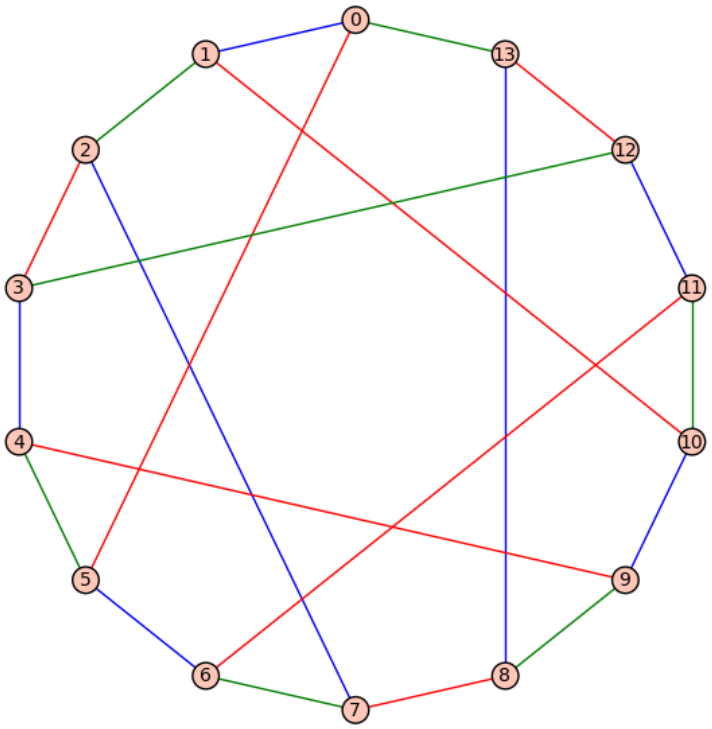


Figure 8.3: The Heawood graph and an edge-coloring example.

```
sage: G = graphs.HeawoodGraph()
sage: vc = G.coloring()
sage: vc
[[1, 3, 5, 7, 9, 11, 13], [0, 2, 4, 6, 8, 10, 12]]
sage: d = {'blue':vc[0], 'red':vc[1]}
sage: G.show(vertex_colors = d)
sage: G.chromatic_number()
2
```

Example 8.7. The Icosahedral graph, shown in Figure 8.5, is a particular projection of the edges of the solid icosahedron onto the plane. It is a 4-regular graph with 30 edges and 12 vertices. It has chromatic number 4 and chromatic index 5. An edge-coloring is shown in Figure 8.6. A vertex-coloring is shown in Figure 8.5. ■

```
sage: G = graphs.IcosahedralGraph()
sage: G.chromatic_number()
4
sage: vc = G.coloring()
sage: d = {'blue':vc[0], 'red':vc[1], 'green':vc[2],'orange':vc[3]}
sage: G.show(vertex_colors = d)

sage: G.is_hamiltonian()
True
sage: G.is_regular()
True
sage: G.is_vertex_transitive()
True
```

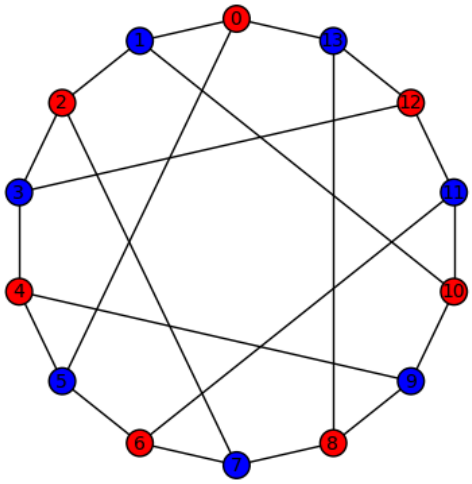


Figure 8.4: The Heawood graph and an vertex-coloring example.

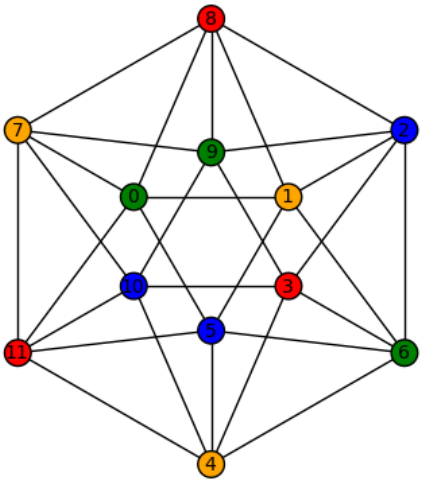


Figure 8.5: An icosahedral graph vertex-coloring example.

```
sage: G.is_perfect()
False
sage: G.is_planar()
True
sage: G.is_clique()
False
sage: G.is_bipartite()
False

sage: G.line_graph().chromatic_number()
5
sage: ec = edge_coloring(G)
sage: d = {'blue':ec[0], 'red':ec[1], 'green':ec[2],'orange':ec[3],'yellow':ec[4]}
sage: G.show(edge_colors = d)
```

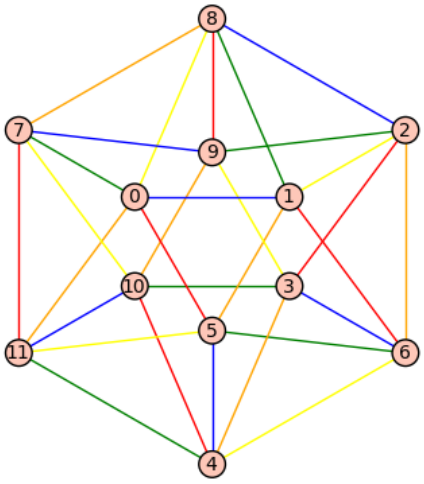


Figure 8.6: An icosahedral graph edge-coloring example.

As in the case of vertex-coloring, the edge-coloring decision problem is still NP-complete. However, it is much better understood through Vizing's theorem.

Theorem 8.8. *(Vizing's theorem) The edges of a graph G can be properly colored using at least $\Delta(G)$ colors and at most $\Delta(G)+1$,*

$$\Delta(G) \leq \chi_e(G) \leq \Delta(G)+1.$$

Notice that the lower bound can be easily proved : if a vertex v has a degree $d(v)$, then at least $d(v)$ colors are required to color G as all the edges incident to v must receive different colors.

Note the upper bound of $\Delta(G)+1$ cannot be deduced from the greedy algorithm given in the previous section, as the maximal degree of the line graph $L(G)$ is not equal to $\Delta(G)$ but to $\max_{u\sim v} d(u)+d(v)-2$, which can reach $2\Delta(G)-2$ in regular graphs.

Example 8.9. The Pappus graph, shown in Figure 8.7, is named after Pappus of Alexandria. It is 3-regular, symmetric, and distance-regular with 18 vertices and 27 edges. The Pappus graph has girth 6, diameter 4, radius 4, chromatic number 2 (i.e. is bipartite), chromatic index 3 and is both 3-vertex-connected and 3-edge-connected. ■

```
sage: G = graphs.PappusGraph()
sage: G.coloring()
[[1, 3, 5, 6, 8, 10, 12, 14, 16], [0, 2, 4, 7, 9, 11, 13, 15, 17]]
sage: G.is_regular()
```

```
True
sage: G.is_planar()
False
sage: G.is_vertex_transitive()
True
sage: G.is_hamiltonian()
True
sage: G.girth()
6
sage: G.is_bipartite()
True
sage: G.show(dpi=300)
sage: G.line_graph().chromatic_number()
3
sage: ec = edge_coloring(G)
sage: ec
[[(0, 1), (2, 3), (4, 5), (6, 17), (7, 14), (8, 13), (9, 16), (10, 15), (11, 12)],
 [(0, 5), (1, 2), (3, 4), (6, 13), (7, 12), (8, 15), (9, 14), (10, 17), (11, 16)],
 [(0, 6), (1, 7), (2, 8), (3, 9), (4, 10), (5, 11), (12, 15), (13, 16), (14, 17)]]
sage: d = {'blue':ec[0], 'red':ec[1], 'green':ec[2]}
sage: G.plot(edge_colors = d).show()
```

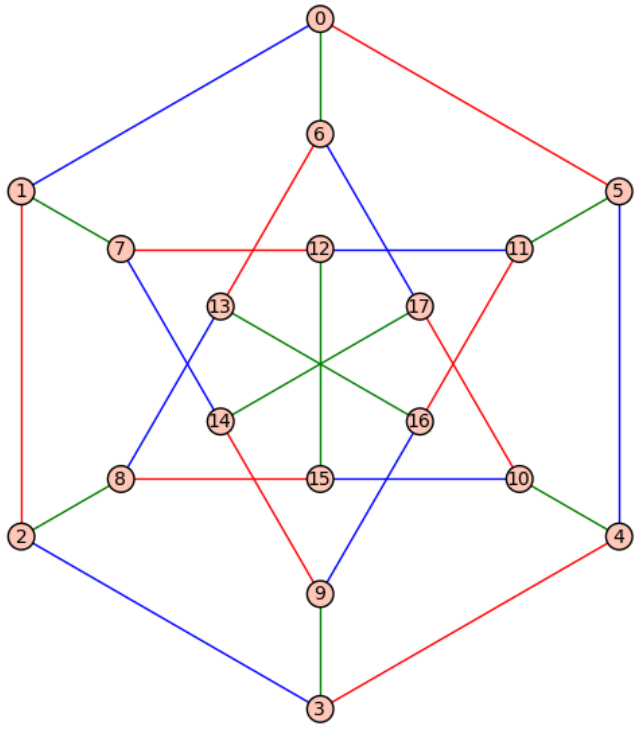


Figure 8.7: A Pappas graph edge-coloring example.

- algorithm for edge coloring by maximum matching
- algorithm for sequential edge coloring

8.3 The chromatic polynomial

George David Birkhoff introduced the chromatic polynomial in 1912. The chromatic polynomial is defined as the unique interpolating polynomial of degree n through the points $(k, P_G(k))$ for $k = 0, 1, \ldots n$, where n is the number of vertices in G. For natural

numbers t, the chromatic polynomial is the function that counts the number of t-colorings of G.

As the name indicates, for a given G the function is indeed a polynomial in t.

For a complex number x, let $x_{(k)} = \prod_{i=0}^{k-1}(x-i)$.

Lemma 8.10. *If $m_k(G)$ denotes the number of distinct partitions of V into k different color-classes then*

$$P_G(t) = \sum_{k=1}^{n} m_k(G) t_{(k)}.$$

Proof. Given a partition of V into k color-classes, we can assign t colors to the color-classes in $t(t-1)\dots(t-k+1) = t_{(k)}$ ways. For each k with $1 \le k \le n$, there are $m_k(G)$ distinct partitions of V into k such color-classes. Therefore, the number of colorings with t colors is $\sum_{k=1}^{n} m_k(G) t_{(k)}$, as desired. ■

Theorem 8.11. *A graph G with n vertices is a tree if and only if $P(G,t) = t(t-1)^{n-1}$.*

Proof. We prove this by induction on n. The statement is clearly true when $n = 1$.

Assume the statement holds for any tree of $n-1$ vertices and let G be a tree with n vertices. There is a vertex v of G having degree 1. The graph $G-v$ is a tree having $n-1$ vertices, so has $t(t-1)^{n-2}$ t-colorings. The number of t-colorings of G is the number of t-colorings of $G-v$ times the number of ways to color v. Since we may color v using any color not used for its one neighbor, there are $t-1$ colorings of v. Therefore, G has $t(t-1)^{n-1}$ t-colorings. ■

The theorem above gives examples of non-isomorphic graphs which have the same chromatic polynomial.

Two graphs are said to be *chromatically equivalent* if they have the same chromatic polynomial. For example, two trees having the same number of vertices are chromatically equivalent.

It is also an open problem to find necessary and sufficient conditions for two arbitrary graphs to be chromatically equivalent.

Theorem 8.12. *If G has n vertices, m edges, and k components G_1, G_2,...,G_k, then*

$$P(G,t) = P(G_1,t)P(G_2,t)\cdots P(G_k,t).$$

This is proven by induction on k and the proof is left to the interested reader.

Fix a pair of vertices $u, v \in V$. Recall that the (edge contraction) graph G/uv is obtained by merging the two vertices and removing any edges between them. Recall that the (edge deletion) graph $G-uv$ is obtained by merging the the edge uv but not the two vertices u, v.

Theorem 8.13. *(Fundamental Reduction Theorem) The chromatic polynomial satisfies the recurrence relation*

$$P(G,t) = P(G-uv,t) - P(G/uv,t).$$

For example, if G is a tree then G/uv is another tree but $G - uv$ is a non-connected graph.

A root (or zero) of a chromatic polynomial, called a *chromatic root* is a value x where $P_G(x) = 0$.

Theorem 8.14. *$\chi_v(C)$ is the smallest positive integer that is not a chromatic root:*

$$\chi_v(G) = \min\{k \mid P(G,k) > 0\}.$$

Birkhoff noted that one could establish the four color theorem by showing $P(G,4) > 0$ for all planar graphs G. This lead to the following statement, which is still an open problem today.

Conjecture 8.15. *(Birkhoff-Lewis Conjecture) As a function of a real variable t, $P(g,t)$ is zero-free in the interval $t \geq 4$.*

8.4 Applications of graph coloring

- assignment problems
- scheduling problems
- matching problems
- map coloring and the Four Color Problem

8.4.1 Application to scheduling

Assume that we have a finite set of different people (for example, students) and a finite set of distinct one-hour meetings they could attend (for example, classes). Construct the graph G as follows. The vertices V of G are given by the set of meetings. We connect two meetings with an edge if some person needs to attend both meetings.

Example 8.16. Suppose the $\times$ in the table below indicated that the student in that column is attending the class in that row.

	Bob	Carol	Ted	Zoe
Calculus		$\times$	$\times$	
Chemistry	$\times$	$\times$	$\times$	$\times$
English	$\times$	$\times$	$\times$	$\times$
History	$\times$			$\times$
Physics		$\times$	$\times$	

Let 0 = Calculus, 1 = Chemistry, 2 = English, 3 = History, and 4 = Physics. Then the corresponding graph can be constructed and drawn in Sage using the following commands (see Figure 8.8). ■

```
sage: G = Graph({0:[1,2,4], 1:[2,3,4], 2:[3,4]})
sage: G.show()
sage: G.coloring()
[[2], [4], [0, 3], [1]]
```

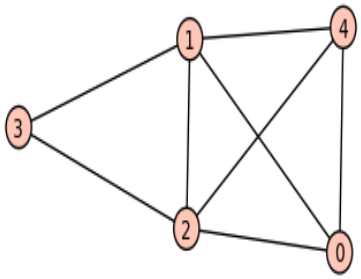


Figure 8.8: An application of graph coloring to scheduling.

Theorem 8.17. *The minimum number of hours required for the schedule of meetings in our scheduling problem is* $\chi_v(G)$.

Proof. Suppose we can schedule the meetings in k hours. In other words, each person attends the meetings they need to and they can do so in at most k kours. Order the meeting times 1, 2, ..., k. Each meeting must occur in one and only one of these time slots (although a given time slot may have several meetings). We color the graph G as follows: if a meeting occurs in hour i then use color i for all the meetings that meet at that hour. Consider two adjacent vertices. These vertices correspond to two meetings which share one or more people. Since a person cannot be in two places at the same time, these two meetings by have different meeting times, hence the vertices must have different colors. This implies $\chi_v(G) \leq k$.

Conversely, suppose that G has a k-coloring. The meetings with color i (where $1 \leq i \leq k$) can be held at the same time since any two such meetings correspond to non-adjacnt vertices, hence have no person in common. Therefore, the minimum number of hours requires for the meeting schedule is less than or equal to $\chi_v(G)$. ■

8.1. What is the number of hours required for a schedule in Example 8.16?

8.2. Draw the Dyck graph in Example 8.2 as a bipartite graph.

8.3. Draw the Pappas graph in Example 8.9 as a bipartite graph.

8.4.2 Map coloring

Theorem 8.18. *(Four Color Theorem) Every planar graph can be* 4*-colored.*

Chapter 9

Network flows

See Jungnickel [**?**], and chapter 12 of Gross and Yellen [**?**].

9.1 Flows and cuts

- single source-single sink networks
- feasible networks
- maximum flow and minimum cut

Let $G = (V, E, i, h)$ be an unweighted multidigraph, as in Definition 1.6.
If F is a field such as $\mathbb{R}$ or $GF(q)$ or a ring such as $\mathbb{Z}$, let

$$C^0(G, F) = \{f : V \to F\}, \quad C^1(G, F) = \{f : E \to F\},$$

be the sets of F-valued functions defined on V and E, respectively. If F is a field then these are F-inner product spaces with inner product

$$(f, g) = \sum_{x \in X} f(x)g(x), \qquad (X = V, \text{ resp. } X = E), \tag{9.1}$$

and

$$\dim C^0(G, F) = |V|, \quad \dim C^1(G, F) = |E|.$$

If you index the sets V and E in some arbitrary but fixed way and define, for $1 \le i \le |V|$ and $1 \le j \le |E|$,

$$f_i(v) = \begin{cases} 1, & v = v_i, \\ 0, & \text{otherwise}, \end{cases} \qquad g_j(e) = \begin{cases} 1, & e = e_j, \\ 0, & \text{otherwise}, \end{cases}$$

then $\mathcal{F} = \{f_i\} \subset C^0(G, F)$ is a basis of $C^0(G, F)$ and $\mathcal{G} = \{g_j\} \subset C^1(G, F)$ is a basis of $C^1(G, F)$.

We order the edges

$$E = \{e_1, e_2, \ldots, e_{|E|}\},$$

in some arbitrary but fixed way. A *vector representation* (or *characteristic vector* or *incidence vector*) of a subgraph $G' = (V, E')$ of $G = (V, E)$, $E' \subset E$, is a binary $|E|$-tuple

$$\mathrm{vec}(G') = (a_1, a_2, \ldots, a_{|E|}) \in GF(2)^{|E|},$$

where

$$a_i = a_i(E') = \begin{cases} 1, & \text{if } e_i \in E', \\ 0, & \text{if } e_i \notin E'. \end{cases}$$

In particular, this defines a mapping

$$\mathrm{vec} : \{\text{subgraphs of } G = (V, E)\} \to GF(2)^{|E|}.$$

For any non-trivial partition

$$V = V_1 \cup V_2, \qquad V_i \neq \emptyset, \quad V_1 \cap V_2 = \emptyset,$$

the set of all edges $e = (v_1, v_2) \in E$, with $v_i \in V_i$ $(i = 1, 2)$, is called a *cocycle*[1] of G. A cocycle with a minimal set of edges is a *bond* (or *cut set*) of G. An *Euler subgraph* is either a cycle or a union of edge-disjoint cycles.

The set of cycles of G is denoted $Z(G)$ and the set of cocycles is denoted $Z^*(G)$.

The F-vector space spanned by the vector representations of all the cycles is called the *cycle space* of G, denoted $\mathcal{Z}(G) = \mathcal{Z}(G, F)$. This is the kernel of the incidence matrix of G (§14.2 in Godsil and Royle [**?**]). Define

$$D : C^1(G, F) \to C^0(G, F),$$
$$(Df)(v) = \textstyle\sum_{h(e)=v} f(e) - \sum_{t(e)=v} f(e).$$

With respect to these bases $\mathcal{F}$ and $\mathcal{G}$, the matrix representing the linear transformation $D : C^1(G, F) \to C^0(G, F)$ is the incidence matrix. An element of the kernel of D is sometimes called a *flow* (see Biggs [**?**]) or *circulation* (see below). Therefore, this is sometimes also referred to as the *space of flows* or the *circulation space*.

It may be regarded as a subspace of $C^1(G, F)$ of dimension $n(G)$. When F is a finite field, sometimes[2] the cycle space is called the *cycle code of* G.

Let F be a field such as $\mathbb{R}$ or $GF(q)$, for some prime power q. Let G be a digraph. Some define a *circulation* (or *flow*) on $G = (V, E)$ to be a function

$$f : E \to F,$$

satisfying[3]

- $\sum_{u \in V,\ (u,v) \in E} f(u, v) = \sum_{w \in V,\ (v,w) \in E} f(v, w)$.

(Note: this is simply the condition that f belongs to the kernel of D.)

Suppose G has a subgraph H and f is a circulation of G such that f is a constant function on H and 0 elsewhere. We call such a circulation a *characteristic function* of H. For example, if G has a cycle C and if f is the characteristic function on C, then f is a circulation.

The *circulation space* $\mathcal{C}$ is the F-vector space of circulation functions. The cycle space "clearly" may be identified with a subspace of the circulation space, since the F-vector

[1] Also called an *edge cut subgraph* or *disconnecting set* or *seg* or *edge cutset*.

[2] Jungnickel and Vanstone in [**?**] call this the *even graphical code* of G.

[3] Note: In addition, some authors add the condition $f(e) \geq 0$ - see e.g., Chung [**?**].

space spanned by the characteristic functions of cycles may be identified with the cycle space of G. In fact, these spaces are isomorphic. Under the inner product (9.1), i.e.,

$$(f, g) = \sum_{e \in E} f(e)g(e), \tag{9.2}$$

this vector space is an inner product space.

Example 9.1. This example is not needed but is presented for its independent interest. Assume $G = (V, E)$ is a strongly connected directed graph. Define the *transition probability matrix* P for a digraph G by

$$P(x, y) = \begin{cases} d_x, & \text{if } (x, y) \in E, \\ 0, & \text{otherwise,} \end{cases}$$

where d_x denotes the out-degree. The Perron-Frobenius Theorem states that there exists a unique left eigenvector ϕ such that (when regarded as a function $\phi : V \to \mathbb{R}$) $\phi(v) > 0$, for all $v \in V$ and $\phi P = \rho\phi$, where ρ is the spectral radius of G. We scale ϕ so that $\sum_{v \in V} \phi(v) = 1$. (This vector is sometimes called the *Perron vector*.) Let $F_\phi(u, v) = \phi(v)P(u, v)$. *Fact:* F_ϕ is a circulation. For a proof, see F. Chung [**?**].

If the edges of E are indexed in some arbitrary but fixed way then a circulation function restricted to a subgraph H of G may be identified with a vector representation of H, as described above. Thefore, the circulation functions gives a coordinate-free version of the cycle space.

The F-vector space spanned by the vector representations of all the segs is called the *cocycle space* (or the *cut space*) of G, denoted $\mathcal{Z}^*(G) = \mathcal{Z}^*(G, F)$. This is the column space of the transpose of the incidence matrix of G (§14.1 in Godsil and Royle [**?**]). It may be regarded as a subspace of $C^1(G, F)$ of dimension the rank of G, $r(G)$. When F is a finite field, sometimes the cocycle space is called the *cocycle code of* G.

Lemma 9.2. Under the inner product (9.1) on $C^1(G, F)$, the cycle space is orthogonal to the cocycle space.

Solution. One proof follows from Theorem 8.3.1 in Godsil and Royle [**?**].

Here is another proof. By Theorem 2.3 in Bondy and Murty [**?**, p.27], an edge of G is an edge cut if and only if it is contained in no cycle. Therefore, the vector representation of any cocycle is supported on a set of indices which is disjoint from the support of the vector representation of any cycle. Therefore, there is a basis of the cycle space which is orthogonal to a basis of the cocycle space. ■

Proposition 9.3. Let $F = GF(2)$. The cycle code of a graph $G = (V, E)$ is a linear binary block code of length $|E|$, dimension equal to the nullity of the graph, $n(G)$, and minimum distance equal to the girth of G. If $C \subset GF(2)^{|E|}$ is the cycle code associated to G and C^* is the cocycle code associated to G then C^* is the dual code of C. In particular, the cocycle code of G is a linear binary block code of length $|E|$, and dimension $r(G) = |E| - n(G)$.

This follows from Hakimi-Bredeson [**?**] (see also Jungnickel-Vanstone [**?**]) in the binary case[4].

[4]It is likely true in the non-binary case as well, but no proof seems to be in the literature.

Solution. Let d denote the minimum distance of the code C. Let γ denote the girth of G, i.e., the smallest cardinality of a cycle in G. If K is a cycle in G then the vector $\mathrm{vec}(K) \in GF(2)^{|E|}$ is an element of the cycle code $C \subset GF(2)^{|E|}$. This implies $d \leq \gamma$.

In the other direction, suppose K_1 and K_2 are cycles in G with associated support vectors $v_1 = \mathrm{vec}(K_1)$, $v_2 = \mathrm{vec}(K_2)$. Assume that at least one of these cycles is a cycle of minimum length, say K_1, so the weight of its corresponding support vector is equal to the girth γ. The only way that $\mathrm{wt}(v_1 + v_2) < \min\{\mathrm{wt}(v_1), \mathrm{wt}(v_2)\}$ can occur is if K_1 and K_2 have some edges in common. In this case, the vector $v_1 + v_2$ represents a subgraph which is either a cycle or it is a union of disjoint cycles. In either case, by minimality of K_1, these new cycles must be at least as long. Therefore, $d \geq \gamma$, as desired.

■

Consider a spanning tree T of a graph G and its complementary subgraph $\overline{T}$. For each edge e of $\overline{T}$ the graph $T \cup e$ contains a unique cycle. The cycles which arise in this way are called the *fundamental cycles* of G, denoted $cyc(T, e)$.

Example 9.4. Consider the graph below, with edges labeled as indicated, together with a spanning tree, depicted to its right, in Figure 9.4.

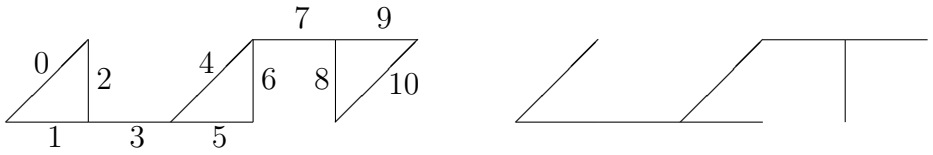


Figure 9.1: A graph and a spanning tree for it.

You can see from Figure 9.4 that:

- by adding edge 2 to the tree, you get a cycle $0, 1, 2$ with vector representation

$$g_1 = (1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0),$$

- by adding edge 6 to the tree, you get a cycle $4, 5, 6$ with vector representation

$$g_2 = (0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0),$$

- by adding edge 10 to the tree, you get a cycle $8, 9, 10$ with vector representation

$$g_3 = (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1).$$

The vectors $\{g_1, g_2, g_3\}$ form a basis of the cycle space of G over $GF(2)$.

The *cocycle space* of a graph G (also known as the *bond space of* G or the *cut-set space* of G) is the F-vector space spanned by the characteristic functions of bonds.

Example 9.5. Consider the graph below, with edges labeled as indicated, together with an example of a bond, depicted to its right, in Figure 9.5.

You can see from Figure 9.5 that:

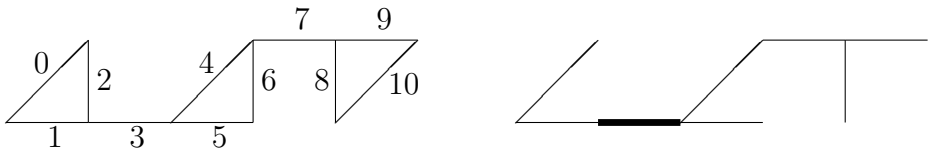


Figure 9.2: A graph and a bond of it.

- by removing edge 3 from the graph, you get a bond with vector representation

$$b_1 = (0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0),$$

- by removing edge 7 from the graph, you get a bond with vector representation

$$b_2 = (0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0),$$

- by removing edges $0, 1$ from the graph, you get a bond with vector representation

$$b_3 = (1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0),$$

- by removing edges $1, 2$ from the graph, you get a bond with vector representation

$$b_4 = (0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0),$$

- by removing edges $4, 5$ from the graph, you get a bond with vector representation

$$b_5 = (0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0),$$

- by removing edges $4, 6$ from the graph, you get a bond with vector representation

$$b_6 = (0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0),$$

- by removing edges $8, 9$ from the graph, you get a bond with vector representation

$$b_7 = (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0),$$

- by removing edges $9, 10$ from the graph, you get a bond with vector representation

$$b_8 = (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1).$$

The vectors $\{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8\}$ form a basis of the cocycle space of G over $GF(2)$.

Note that these vectors are orthogonal to the basis vectors of the cycle space in Example 9.4. Note also that the xor sum of two cuts is not a cut.For example, if you xor the bond $4, 5$ with the bond $4, 6$ then you get the subgraph foormed by the edges $5, 6$ and that is not a disconnecting cut of G.

9.1.1 Electrical networks

We present, as an application of circuit matrices and incidence matrices, and abstract definition of an electrical network.

We define an electrical network abstractly as follows. Let $G = (V, E)$ be a simple connected directed graph having n vertices and m edges. We have a *current* function

$$i : E \to \mathbb{R},$$

and a *voltage* function

$$v : E \to \mathbb{R},$$

subject to three conditions. If we index the edges, say as $E = \{e_1, \ldots, e_m\}$, then these functions may (and sometimes will) be regarded as column vectors in $\mathbb{R}^m$.

The conditions satisfied by the current function are the following.

- *Kirchhoff's current law*,

$$Ai = \vec{0}, \tag{9.3}$$

  where A is the incidence matrix and i is regarded as a column vector. This equation comes from the fact that the algebraic sum of the currents going into a node is zero.

- *Kirchhoff's voltage law*,

$$Cv = \vec{0}, \tag{9.4}$$

  where C is the circuit matrix and v is regarded as a column vector. This equation comes from the fact that the algebraic sum of the voltage drops around a closed loop is zero.

- In a network with resistors but no inductors or capacitors, there is a relationship between i and v given by

$$Ri = v + b,$$

  where b is a vector of external "battery" sources and R is a "resistor matrix."

9.2 Chip firing games

Chip firing games on graphs (which are just pure fun) relate to "abelian sandpile models" from physics to "rotor-routing models" from theoretical computer scientists (designing efficient computer multiprocessor circuits) to "self-organized criticality" (a subdiscipline of dynamical systems) to "algebraic potential theory" on a graph [**?**] to cryptography (via the Biggs cryptosystem). Moreover, it relates the concepts of the Laplacian of the graph to the tree number to the circulation space of the graph to the incidence matrix, as well as many other ideas. Some good references are [**?**], [**?**], [**?**], [**?**] and [**?**].

Basic set-up

A *chip firing game* always starts with a directed multigraph G having no loops. A *configuration* is a vertex-weighting, i.e., a function $s : V \to \mathbb{R}$. The players are represented by the vertices of G and the vertex-weights represent the number of chips each player (represented by that vertex) has. The initial vertex-weighting is called the *starting configuration* of G. Let vertex v have outgoing degree $d_+(v)$. If the weight of vertex v is $\geq d_+(v)$ (so that player can afford to give away all their chips) then that vertex is called *active*.

Here is some SAGE/Python code for determining the active vertices.

SAGE

```
def active_vertices(G, s):
    """
    Returns the list of active vertices.

    INPUT:
     G - a graph
     s - a configuration (implemented as a list
                          or a dictionary keyed on
                          the vertices of the graph)

    EXAMPLES:
        sage: A = matrix([[0,1,1,0,0],[1,0,1,0,0],[1,1,0,1,0],[0,0,0,0,1],[0,0,0,0,0]])
        sage: G = Graph(A, format = "adjacency_matrix", weighted = True)
        sage: s = {0: 3, 1: 1, 2: 0, 3: 1, 4: 1}
        sage: active_vertices(G, s)
        [0, 4]

    """
    V = G.vertices()
    degs = [G.degree(v) for v in V]
    active = [v for v in V if degs[V.index(v)]<=s[v]]
    return active
```

If v is active then when you fire v you must also change the configuration. The new configuration s' will satisfy $s'(v) = s(v) - d_+(v)$ and $s'(v') = s(v') + 1$ for each neighbor v' of v. In other words, v will give away one chip to each of its $d_+(v)$ neighbors. If $x : V \to \{0, 1\}^{|V|} \subset \mathbb{R}^{|V|}$ is the representation vector ("characteristic function") of a vertex then this change can be expressed more compactly as

$$s' = s - L \circ x, \tag{9.5}$$

where L is the vertex Laplacian. It turns out that the column sums of L are all 0, so this operation does not change the total number of chips. We use the notation

$$s \xrightarrow{v} s',$$

to indicate that the configuration s' is the result of firing vertex v in configuration s.

Example 9.6. Consider the graph

This graph has incidence matrix

$$D = \begin{pmatrix} -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 \\ 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

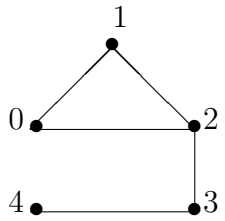


Figure 9.3: A graph with 5 vertices.

and Laplacian

$$L = D \cdot {}^tD = \begin{pmatrix} 2 & -1 & 0-1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix}.$$

Suppose the initial configuration is $s = (3, 1, 0, 1, 1)$, i.e.,

- player 0 has 3 dollars,
- player 1 has 1 dollar,
- player 2 has nothing,
- player 3 has 1 dollar,
- player 4 has 1 dollar.

Notice player 0 is active. If we fire 0 then we get the new configuration $s' = (1, 2, 1, 1, 1)$. Indeed, if we compute $s' = s - Lx(0)$, we get:

$$s' = \begin{pmatrix} 3 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 2 \\ -1 \\ -1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

This can be written more concisely as

$$(3, 1, 0, 1, 1) \xrightarrow{0} (1, 2, 1, 1, 1).$$

We have the cycle

$$(1, 2, 1, 1, 1) \xrightarrow{1} (2, 0, 2, 1, 1) \xrightarrow{0} (0, 1, 3, 1, 1) \xrightarrow{2} (1, 2, 0, 2, 1)$$
$$\xrightarrow{3} (1, 2, 1, 0, 2) \xrightarrow{4} (1, 2, 1, 1, 1).$$

Chip-firing game variants

For simplicity, let $G = (V, E)$ be an undirected graph with an indexed set of vertices $V = \{v_1, \ldots, v_m\}$ and an indexed set of vertices $E = \{e_1, \ldots, e_n\}$.

One variant (the "sandpile model") has a special vertex, called "the sink," which has special firing properties. In the sandpile variant, the sink is never fired. Another variant (the "dollar game") has a special vertex, called "the source," which has special firing properties. In the dollar game variant, the source is only fired when not other vertex is active. We shall consider the dollar game variant here, following Biggs [**?**].

We select a distinguished vertex $q \in V$, called the "source[5]," which has a special property to be described below. For the dollar game, a *configuration* is a function $s : V \to \mathbb{R}$ for which

$$\sum_{v \in V} s(v) = 0,$$

and $s(v) \geq 0$ for all $v \in V$ with $v \neq q$. A vertex $v \neq q$ can be fired if and only if $\deg(v) \leq s(v)$ (i.e., it "has enough chips"). The equation (9.5) describes the new configuration after firing a vertex.

Here is some SAGE/Python code for determining the configuration after firing an active vertex.

SAGE

```
def fire(G, s, v0):
    """
    Returns the configuration after firing the active vertex v.

    INPUT:
     G - a graph
     s - a configuration (implemented as a list
                          or a dictionary keyed on
                          the vertices of the graph)
     v - a vertex of the graph

    EXAMPLES:
        sage: A = matrix([[0,1,1,0,0],[1,0,1,0,0],[1,1,0,1,0],[0,0,0,0,1],[0,0,0,0,0]])
        sage: G = Graph(A, format = "adjacency_matrix", weighted = True)
        sage: s = {0: 3, 1: 1, 2: 0, 3: 1, 4: 1}
        sage: fire(G, s, 0)
        {0: 1, 1: 2, 2: 1, 3: 1, 4: 1}

    """
    V = G.vertices()
    j = V.index(v0)
    s1 = copy(s)
    if not(v0 in V):
        raise ValueError, "the last argument must be a vertex of the graph."
    if not(v0 in active_vertices(G, s)):
        raise ValueError, "the last argument must be an active vertex of the graph."
    degs = [G.degree(w) for w in V]
    for w in V:
         if w == v0:
             s1[v0] = s[v0] - degs[j]
         if w in G.neighbors(v0):
             s1[w] = s[w]+1
    return s1
```

We say $s : V \to \mathbb{R}$ is a *stable* configuration if $0 \leq s(v) < \deg(v)$, for all $v \neq q$. The source vertex q can only be fired when no other vertex can be fired, that is only in the case when a stable configuration has been reached.

[5]Biggs humorously calls q "the government."

Here is some SAGE/Python code for determining the stable vertices.

SAGE

```
def stable_vertices(G, s, source = None):
    """
    Returns the list of stable vertices.

    INPUT:
     G - a graph
     s - a configuration (implemented as a list
                          or a dictionary keyed on
                          the vertices of the graph)

    EXAMPLES:
        sage: A = matrix([[0,1,1,0,0],[1,0,1,0,0],[1,1,0,1,0],[0,0,0,0,1],[0,0,0,0,0]])
        sage: G = Graph(A, format = "adjacency_matrix", weighted = True)
        sage: s = {0: 3, 1: 1, 2: 0, 3: 1, 4: 1}
        sage: stable_vertices(G, s)

    """
    V = G.vertices()
    degs = [G.degree(v) for v in V]
    if source==None:
        stable = [v for v in V if degs[V.index(v)]>s[v]]
    else:
        stable = [v for v in V if degs[V.index(v)]>s[v] and v!=source]
    return stable
```

Suppose we are in a configuration s_1. We say a sequence vertices $S = (w_1, w_2, \ldots, w_k)$, $w_i \in V$ not necessarily distinct, is *legal* if,

- w_1 is active in configuration s_1,
- for each i with $1 \leq i < k$, s_{i+1} is obtained from s_i by firing w_i in configuration s_i,
- for each i with $1 \leq i < k$, w_{i+1} is active in the configuration s_{i+1} defined in the previous step,
- the source vertex q occurs in S only if it immediately follows a stable configuration.

We call s_1 or w_1 the *start* of S. A configuration s is *recurrent* if there is a legal sequence starting at s which leads back to s. A configuration is *critical* if it recurrent and stable.

Here is some SAGE/Python code for determining a stable vertex resulting from a legal sequence of firings of a given configuration s. I think it returns the unique critical configuration associated to s but have not proven this.

SAGE

```
def stabilize(G, s, source, legal_sequence = False):
    """
    Returns the stable configuration of the graph originating from
    the given configuration s. If legal_sequence = True then the
    sequence of firings is also returned. By van den Heuvel [1],
    the number of firings needed to compute a critical configuration
    is < 3(S+2|E|)|V|^2, where S is the sum of the positive
    weights in the configuration.

    EXAMPLES:
        sage: A = matrix([[0,1,1,0,0],[1,0,1,0,0],[1,1,0,1,0],[0,0,1,0,1],[0,0,0,1,0]])
        sage: G = Graph(A, format="weighted_adjacency_matrix")
        sage: s = {0: 3, 1: 1, 2: 0, 3: 1, 4: -5}
        sage: stabilize(G, s, 4)
        {0: 0, 1: 1, 2: 2, 3: 1, 4: -4}

    REFERENCES:
```

```
        [1] J. van den Heuvel, "Algorithmic aspects of a chip-firing
            game," preprint.
    """
    V = G.vertices()
    E = G.edges()
    fire_number = 3*len(V)^2*(sum([s[v] for v in V if s[v]>0])+2*len(E))+len(V)
    if legal_sequence:
        seq = []
    stab = []
    ac = active_vertices(G,s)
    for i in range(fire_number):
        if len(ac)>0:
            s = fire(G,s,ac[0])
            if legal_sequence:
                seq.append(ac[0])
        else:
            stab.append(s)
            break
        ac = active_vertices(G,s)
    if len(stab)==0:
        raise ValueError, "No stable configuration found."
    if legal_sequence:
        return stab[0], seq
    else:
        return stab[0]
```

The incidence matrix D and its transpose ${}^tD$ can be regarded as homomorphisms

$$D : C^1(G,\mathbb{Z}) \to C^0(G,\mathbb{Z}) \quad \text{and} \quad {}^tD : C^0(G,\mathbb{Z}) \to C^1(G,\mathbb{Z}).$$

We can also regard the Laplacian $L = D \cdot {}^tD$ as a homomorphism $C^0(G,\mathbb{Z}) \to C^0(G,\mathbb{Z})$. Denote by $\sigma : C^0(G,\mathbb{Z}) \to \mathbb{Z}$ the homomorphism defined by

$$\sigma(f) = \sum_{v \in V} f(v).$$

Denote by $K(G)$ the set of *critical configurations* on a graph G.

Lemma 9.7. *(Biggs [?])* The set $K(G)$ of critical configurations on a connected graph G is in bijective correspondence with the abelian group $\mathrm{Ker}(\sigma)/\mathrm{Im}(Q)$.

If you accept this lemma (which we do not prove here) then you must believe that there is a bijection $f : K(G) \to \mathrm{Ker}(\sigma)/\mathrm{Im}(Q)$. Now, a group operation $\bullet$ on $K(G)$ an be defined by

$$a \bullet b = f^{-1}(f(a) + f(b)),$$

for all $a, b \in \mathrm{Ker}(\sigma)/\mathrm{Im}(Q)$.

Example 9.8. Consider again the graph

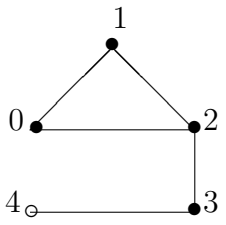


Figure 9.4: A graph with 5 vertices.

This graph has incidence matrix

$$D = \begin{pmatrix} -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 \\ 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

and Laplacian

$$L = D \cdot {}^tD = \begin{pmatrix} 2 & -1 & 0-1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ -1 & -1 & 3 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix}.$$

Suppose the initial configuration is $s = (3, 1, 0, 1, -5)$, i.e.,

- player 0 has 3 dollars,
- player 1 has 1 dollar,
- player 2 has nothing,
- player 3 has 1 dollar,
- player 4 is the source vertex q.

The legal sequence $(0, 1, 0, 2, 1, 0, 3, 2, 1, 0)$ leads to the stable configuration $(0, 1, 2, 1, -4)$. If q is fired then the configuration $(0, 1, 2, 2, -5)$ is achieved. This is recurrent since it is contained in the cyclic legal sequence

$$\begin{gathered}(0, 1, 2, 2, -5) \xrightarrow{3} (0, 1, 3, 0, -4) \xrightarrow{2} (1, 2, 0, 1, -4) \\ \xrightarrow{1} (2, 0, 1, 1, -4) \xrightarrow{0} (0, 1, 2, 1, -4) \xrightarrow{q} (0, 1, 2, 2, -5).\end{gathered}$$

In particular, the configuration $(0, 1, 2, 1, -4)$ is also recurrent. Since it is both stable and recurrent, it is critical.

The following result is of basic importance but I'm not sure who proved it first. It is quoted in many of the papers on this topic in one form or another.

Theorem 9.9. *(Biggs [?], Theorem 3.8)* If s is an configuration and G is connected then there is a unique critical configuration s' which can be obtained by a sequence of legal firings for starting at s.

The map defined by the above theorem is denoted

$$\gamma : C^0(G, \mathbb{R}) \to K(G).$$

Another way to define multiplication $\bullet$ on on $K(G)$ is

$$\gamma(s_1) \bullet \gamma(s_2) = \gamma(s_1 + s_2),$$

where $s_1 + s_2$ is computed using addition on $C^0(G, \mathbb{R})$. According to Perkinson [**?**], Theorem 2.16, the critical group satisfies the following isomorphism:

$$K(G) \cong \mathbb{Z}^{m-1}/\mathcal{L},$$

where $\mathcal{L}$ is the integer lattice generated by the columns of the reduced Laplacian matrix[6].

If s is a configuration then we define

$$\mathrm{wt}(s) = \sum_{v \in V, v \neq q} s(q)$$

to be the *weight* of the configuration. The *level* of the configuration is defined by

$$\mathrm{level}(s) = \mathrm{wt}(s) - |E| + \deg(q).$$

Lemma 9.10. *(Merino [**?**])* If s is a critical configuration then

$$0 \leq \mathrm{level}(s) \leq |E| - |V| + 1.$$

This is proven in Theorem 3.4.5 in [**?**]. What is also proven in [**?**] is a statement whcih computes the number of critical configurations of a given level in terms of the Tutte polynomial of the associated graph.

9.3 Ford-Fulkerson theorem

The Ford-Fulkerson Theorem, or "Max-flow/Min-cut Theorem," was proven by P. Elias, A. Feinstein, and C.E. Shannon in 1956, and, independently, by L.R. Ford, Jr. and D.R. Fulkerson in the same year. So it should be called the "Elias-Feinstein-Ford-Fulkerson-Shannon Theorem," to be precise about the authorship.

To explain the meaning of this theorem, we need to introduce some notation and terminology.

Consider an edge-weighted simple digraph $G = (V, E, i, h)$ without negative weight cycles. Here $E \subset V^{(2)}$, i is an incidence function as in (**??**), which we regard as the identity function, and h is an orientation function as in (**??**). Let G be a *network*, with two distinguished vertices, the "source" and the "sink." Let s and t denote the source and the sink of G, respectively. The *capacity* (or *edge capacity*)) is a mapping $c : E \to \mathbf{R}$, denoted by c_{uv} or $c(u, v)$, for $(u, v) \in E$ and $h(e) = u$. If $(u, v) \in E$ and $h(e) = v$ then we set, by convention, $c(v, u) = -c(u, v)$. Thinking of a graph as a network of pipes (representing the edges) transporting water with various junctions (representing vertices), the capacity function represents the maximum amount of "flow" that can pass through an edge.

A *flow* is a mapping $f : E \to \mathbf{R}$, denoted by f_{uv} or $f(u, v)$, subject to the following two constraints:

- $f(u, v) \leq c(u, v)$, for each $(u, v) \in V$ (the "capacity constraint"),
- $\sum_{u \in V,\ (u,v) \in E} f(u, v) = \sum_{u \in V,\ (v,u) \in E} f(v, u)$, for each $v \in V$ (conservation of flows).

[6]The *reduced Laplacian* matrix is obtained from the Laplacian matrix by removing the row and column associated to the source vertex.

An edge $(u, v) \in E$ is *f-saturated* if $f(u, v) = c(u, v)$. An edge $(u, v) \in E$ is *f-zero* if $f(u, v) = 0$. A path with available capacity is called an "augmenting path." More precisely, a directed path form s to t is *f-augmenting*, or f-unsaturated, if no forward edge is f-saturated and no backward edge is f-zero.

The *value of the flow* is defined by

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s),$$

where s is the source. It represents the amount of flow passing from the source to the sink. The *maximum flow problem* is to maximize $|f|$, that is, to route as much flow as possible from s to t.

Example 9.11. Consider the digraph having adjacency matrix

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

depicted in Figure 9.5.

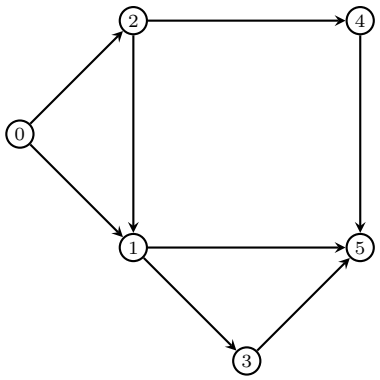


Figure 9.5: A digraph with 6 vertices.

Suppose that each edge has capacity 1. A maximum flow f is obtained by taking a flow value of 1 along each edge of the path

$$p_1 : (0, 1), (1, 5),$$

and a flow value of 1 along each edge of the path

$$p_2 : (0, 2), (2, 4), (4, 5).$$

The maximum value of the flow in this case is $|f| = 2$.

This graph can be created in Sage using the commands

```
sage: B = matrix([[0,1,1,0,0,0],[0,0,0,1,0,1],[0,1,0,0,1,0],[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,
sage: H = DiGraph(B, format = "adjacency_matrix", weighted=True)
```

Type `H.show(edge_labels=True)` if you want to see the graph with the capacities labeling the edges.

Given a capacitated digraph with capacity c and flow f, we define the *residual digraph* $G_f = (V, E)$ to be the digraph with capacity $c_f(u, v) = c(u, v) - f(u, v)$ and no flow. In other words, G_f is the same graph but it has a different capacity c_f and flow 0. This is also called a *residual network*.

Define an $s - t$ *cut* in our capacitated digraph G to be a partition $C = (S, T)$ of V such that $s \in S$ and $t \in T$. Recall the cut-set of C is the set

$$\{(u, v) \in E \mid u \in S, v \in T\}.$$

Lemma 9.12. Let $G = (V, E)$ be a capacitated digraph with capacity $c : E \to \mathbf{R}$, and let s and t denote the source and the sink of G, respectively. If C is an $s - t$ cut and if the edges in the cut-set of C are removed, then $|f| = 0$.

Exercise 9.13. *Prove Lemma 9.12.*

The *capacity of an* $s - t$ *cut* $C = (S, T)$ is defined by

$$c(S, T) = \sum_{(s,t)\in(S,T)} c(u, v).$$

The *minimum cut problem* is to minimize the amount of capacity of an $s - t$ cut.

The following theorem is due to P. Elias, A. Feinstein, L.R. Ford, Jr., D.R. Fulkerson, C.E. Shannon.

Theorem 9.14. *(max-flow min-cut theorem)* The maximum value of an s-t flow is equal to the minimum capacity of an s-t cut.

The intuitive explanation of this result is as follows.

Suppose that $G = (V, E)$ is a graph where each edge has capacity 1. Let $s \in V$ be the source and $t \in V$ be the sink. The maximum flow from s to t is the maximum number of independent paths from s to t. Denote this maximum flow by m. Each s-t cut must intersect each s-t path at least once. In fact, if S is a minimal s-t cut then for each edge e in S there is an s-t path containing e. Therefore, $|S| \leq e$.

On the other hand, since each edge has unit capacity, the maximum flow value can't exceed the number of edges separating s from t, so $m \leq |S|$.

Remark 9.15. *Although the notion of an independent path is important for the network-theoretic proof of Menger's theorem (which we view as a corollary to the Ford-Fulkerson theorem on network flows on networks having capacity* 1 *on all edges), its significance is less important for networks having arbitrary capacities. One must use caution in generalizing the above intuitive argument to establish a rigorous proof of the general version of the MFMC theorem.*

Remark 9.16. This theorem can be generalized as follows. In addition to edge capacity, suppose there is capacity at each *vertex*, that is, a mapping $c : V \to \mathbf{R}$, denoted by $v \mapsto c(v)$, such that the flow f has to satisfy not only the capacity constraint and the conservation of flows, but also the vertex capacity constraint

$$\sum_{w\in V} f(w, v) \leq c(v),$$

for each $v \in V - \{s, t\}$. Define an $s - t$ *cut* to be the set of vertices and edges such that for any path from s to t, the path contains a member of the cut. In this case, the

capacity of the cut is the sum the capacity of each edge *and* vertex in it. In this new definition, the *generalized max-flow min-cut theorem* states that the maximum value of an $s-t$ flow is equal to the minimum capacity of an $s-t$ cut..

The idea behind the Ford-Fulkerson algorithm is very simple: As long as there is a path from the source to the sink, with available capacity on all edges in the path, we send as much flow as we can alone along each of these paths. This is done inductively, one path at a time.

Algorithm 9.1: Ford-Fulkerson algorithm.

Input: Graph $G = (V, E)$ with flow capacity c, source s, and sink t.
Output: A flow f from s to t which is a maximum for all edges in E.

```
1 f(u,v) ← 0 for each edge uv ∈ E
2 while there is an s-t path p in G_f such that c_f(e) > 0 for each edge e ∈ E do
3     find c_f(p) = min{c_f(u,v) | (u,v) ∈ p}
4     for each edge uv ∈ do
5         f(u,v) = f(u,v) + c_f(p)
6         f(v,u) = f(v,u) − c_f(p)
```

To prove the max-flow/min-cut theorem we will use the following lemma.

Lemma 9.17. Let $G = (V, E)$ be a directed graph with edge capacity $c : E \to \mathbf{Z}$, a source $s \in V$, and a sink $t \in V$. A flow $f : E \to \mathbf{Z}$ is a maximum flow if and only if there is no f-augmenting path in the graph.

In other words, a flow f in a capacitated network is a maximum flow if and only if there is no f-augmenting path in the network.

Solution. One direction is easy. Suppose that the flow is a maximum. If there is an f-augmenting path then the current flow can be increased using that path, so the flow would not be a maximum. This contradiction proves the "only if" direction.

Now, suppose there is no f-augmenting path in the network. Let S be the set of vertices v such that there is an f-unsaturated path from the source s to v. We know $s \in S$ and (by hypothesis) $t \notin S$. Thus there is a cut of the form (S, T) in the network. Let $e = (v, w)$ be any edge in this cut, $v \in S$ and $w \in T$. Since there is no f-unsaturated path from s to w, e is f-saturated. Likewise, any edge in the cut (T, S) is f-zero. Therefore, the current flow value is equal to the capacity of the cut (S, T). Therefore, the current flow is a maximum. ■

We can now prove the max-flow/min-cut theorem.

Solution. Let f be a maximum flow. If

$$S = \{v \in V \mid \text{there exists an } f - \text{saturated path from } s \text{ to } v\},$$

then by the previous lemma, $S \neq V$. Since $T = V - S$ is non-empty, there is a cut $C = (S, T)$. Each edge of this cut C in the capacitated network G is f-saturated.

■

Here is some Python code[7] which implements this. The class `FlowNetwork` is basically a Sage Graph class with edge weights and an extra data structure representing the flow on the graph.

```
class Edge:
    def __init__(self,U,V,w):
        self.source = U
        self.to = V
        self.capacity = w
    def __repr__(self):
        return str(self.source) + "->" + str(self.to) + " : " + str(self.capacity)

class FlowNetwork(object):
    """
    This is a graph structure with edge capacities.

    EXAMPLES:
        g=FlowNetwork()
        map(g.add_vertex, ['s','o','p','q','r','t'])
        g.add_edge('s','o',3)
        g.add_edge('s','p',3)
        g.add_edge('o','p',2)
        g.add_edge('o','q',3)
        g.add_edge('p','r',2)
        g.add_edge('r','t',3)
        g.add_edge('q','r',4)
        g.add_edge('q','t',2)
        print g.max_flow('s','t')
    """
    def __init__(self):
        self.adj, self.flow, = {},{}

    def add_vertex(self, vertex):
        self.adj[vertex] = []

    def get_edges(self, v):
        return self.adj[v]

    def add_edge(self, u,v,w=0):
        assert(u != v)
        edge = Edge(u,v,w)
        redge = Edge(v,u,0)
        edge.redge = redge
        redge.redge = edge
        self.adj[u].append(edge)
        self.adj[v].append(redge)
        self.flow[edge] = self.flow[redge] = 0

    def find_path(self, source, sink, path):
        if source == sink:
            return path
        for edge in self.get_edges(source):
            residual = edge.capacity - self.flow[edge]
            if residual > 0 and not (edge,residual) in path:
                result = self.find_path(edge.to, sink, path + [ (edge,residual) ])
                if result != None:
                    return result

    def max_flow(self, source, sink):
        path = self.find_path(source, sink, [])
        while path != None:
            flow = min(res for edge,res in path)
            for edge,res in path:
                self.flow[edge] += flow
                self.flow[edge.redge] -= flow
            path = self.find_path(source, sink, [])
        return sum(self.flow[edge] for edge in self.get_edges(source))
```

[7] Please see http://en.wikipedia.org/wiki/Ford-Fulkerson_algorithm.

9.4 Edmonds and Karp's algorithm

The objective of this section is to prove Edmond and Karp's algorithm for the maximum flow-minimum cut problem with polynomial complexity.

9.5 Goldberg and Tarjan's algorithm

The objective of this section is to prove Goldberg and Tarjan's algorithm for finding maximal flows with polynomial complexity.

Chapter 10

Random graphs

A random graph can be thought of as being a member from a collection of graphs having some common properties. Recall that Algorithm 3.5 allows for generating a random binary tree having at least one vertex. Fix a positive integer n and let $\mathcal{T}$ be a collection of all binary trees on n vertices. It can be infeasible to generate all members of $\mathcal{T}$, so for most purposes we are only interested in randomly generating a member of $\mathcal{T}$. A binary tree of order n generated in this manner is said to be a random graph.

This chapter is a digression into the world of random graphs and various models for generating different types of random graphs. Unlike other chapters in this book, our approach is rather informal and not as rigorous as in other chapters. We will discuss some common models of random graphs and a number of their properties without being bogged down in details of proofs. Along the way, we will demonstrate that random graphs can be used to model diverse real-world networks such as social, biological, technological, and information networks. The edited volume [**?**] provides some historical context for the "new" science of networks. Bollobás [**?**] and Kolchin [**?**] provide standard references on the theory of random graphs with rigorous proofs. For comprehensive surveys of random graphs and networks that do not go into too much technical details, see [**?**, **?**, **?**, **?**]. On the other hand, surveys that cover diverse applications of random graphs and networks and are geared toward the technical aspects of the subject include [**?**, **?**, **?**, **?**, **?**, **?**, **?**].

10.1 Network statistics

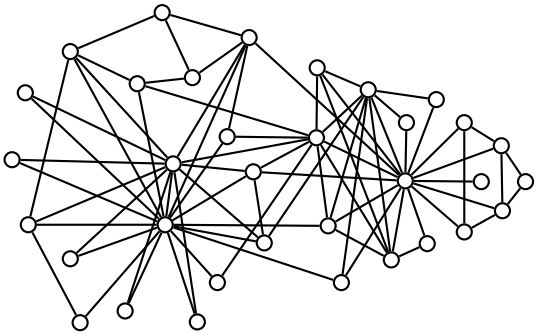
Numerous real-world networks are large, having from thousands up to millions of vertices and edges. Network statistics provide a way to describe properties of networks without concerning ourselves with individual vertices and edges. A network statistic should describe essential properties of the network under consideration, provide a means to differentiate between different classes of networks, and be useful in network algorithms and applications [**?**]. In this section, we discuss various common network statistics that can be used to describe graphs underlying large networks.

10.1.1 Degree distribution

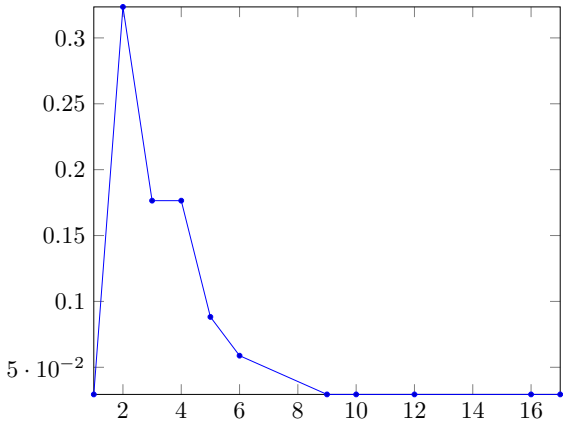
The degree distribution of a graph $G = (V, E)$ quantifies the fraction of vertices in G having a specific degree k. If v is any vertex of G, we denote this fraction by

$$p = \Pr[\deg(v) = k] \tag{10.1}$$

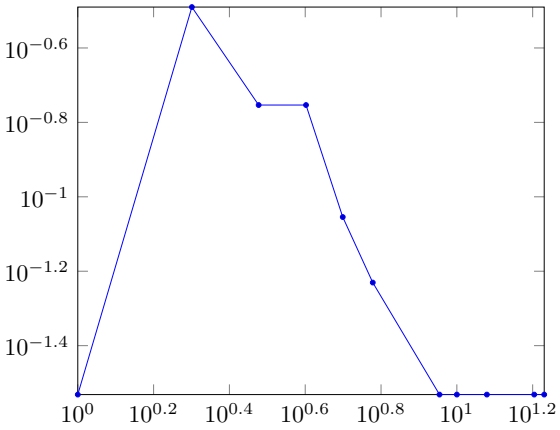
As indicated by the notation, we can think of (10.1) as the probability that a vertex $v \in V$ chosen uniformly at random has degree k. The *degree distribution* of G is consequently a histogram of the degrees of vertices in G. Figure 10.1 illustrates the degree distribution of the Zachary [**?**] karate club network. The degree distributions of many real-world networks have the same general curve as depicted in Figure 10.1(b), i.e. a peak at low degrees followed by a tail at higher degrees. See for example the degree distribution of the neural network in Figure 10.2, that of a power grid network in Figure 10.3, and the degree distribution of a scientific co-authorship network in Figure 10.4.



(a) Zachary karate club network.



(b) Linear scaling.

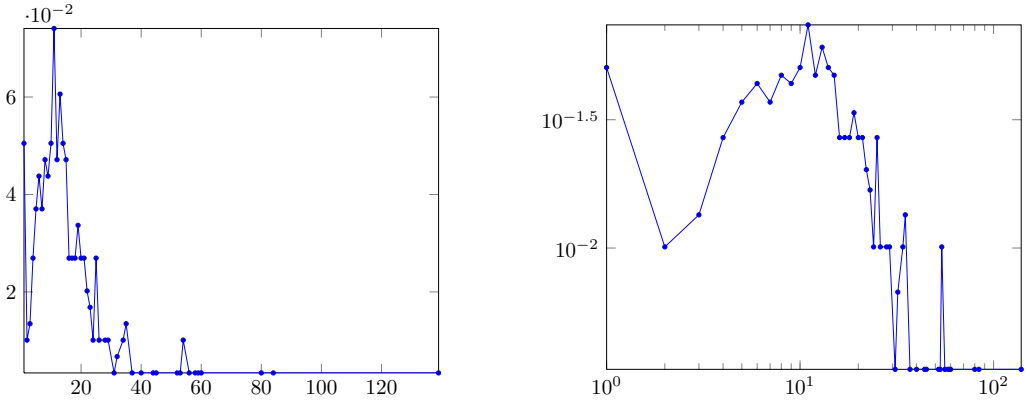


(c) Log-log scaling.

Figure 10.1: The friendship network within a 34-person karate club. This is more commonly known as the Zachary [**?**] karate club network. The network is an undirected, connected, unweighted graph having 34 vertices and 78 edges. The horizontal axis represents degree; the vertical axis represents the probability that a vertex from the network has the corresponding degree.

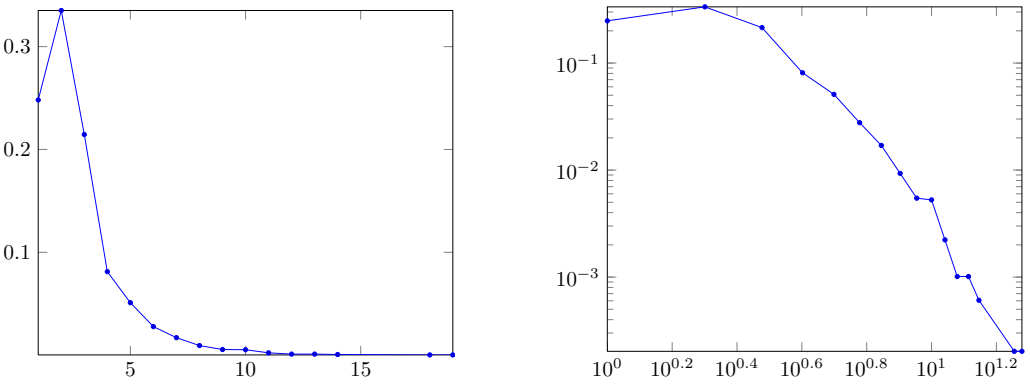
10.1.2 Distance statistics

In chapter 5 we discussed various distance metrics such as radius, diameter, and eccentricity. To that distance statistics collection we add the average or characteristic distance $\overline{d}$, defined as the arithmetic mean of all distances in a graph. Let $G = (V, E)$ be a simple graph with $n = |V|$ and $m = |E|$, where G can be either directed or undirected. Then



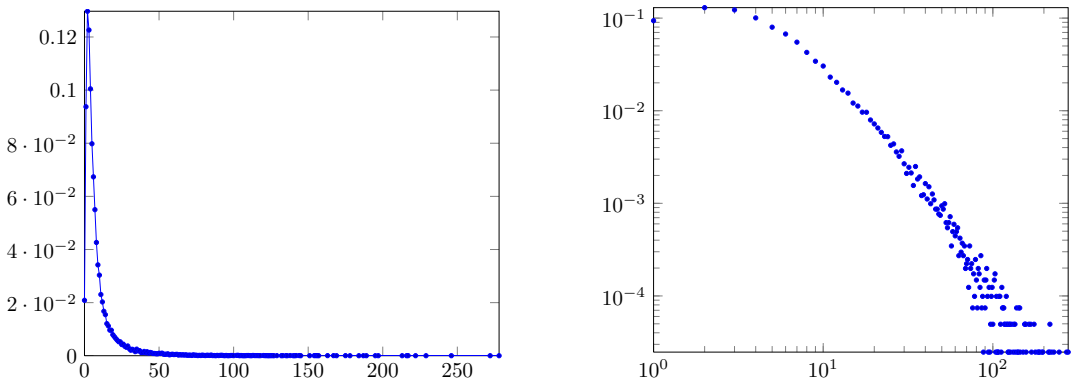
(a) Linear scaling. (b) Log-log scaling.

Figure 10.2: Degree distribution of the neural network of the Caenorhabditis elegans. The network is a directed, not strongly connected, weighted graph with 297 vertices and 2,359 edges. The horizontal axis represents degree; the vertical axis represents the probability that a vertex from the network has the corresponding degree. The degree distribution is derived from dataset by Watts and Strogatz [**?**] and White et al. [**?**].



(a) Linear scaling. (b) Log-log scaling.

Figure 10.3: Degree distribution of the Western States Power Grid of the United States. The network is an undirected, connected, unweighted graph with 4,941 vertices and 6,594 edges. The horizontal axis represents degree; the vertical axis represents the probability that a vertex from the network has the corresponding degree. The degree distribution is derived from dataset by Watts and Strogatz [**?**].



(a) Linear scaling. (b) Log-log scaling.

Figure 10.4: Degree distribution of the network of co-authorships between scientists posting preprints on the condensed matter eprint archive at http://arxiv.org/archive/cond-mat. The network is a weighted, disconnected, undirected graph having 40,421 vertices and 175,693 edges. The horizontal axis represents degree; the vertical axis represents the probability that a vertex from the co-authorship network has the corresponding degree. The degree distribution is derived from the 2005 update of the dataset by Newman [**?**].

G has size at most $n(n-1)$ because for any distinct vertex pair $u, v \in V$ we count the edge from u to v and the edge from v to u. The *characteristic distance* of G is defined by

$$\overline{d}(G) = \frac{1}{n(n-1)} \sum_{u \neq v \in V} d(u, v)$$

where the distance function d is given by

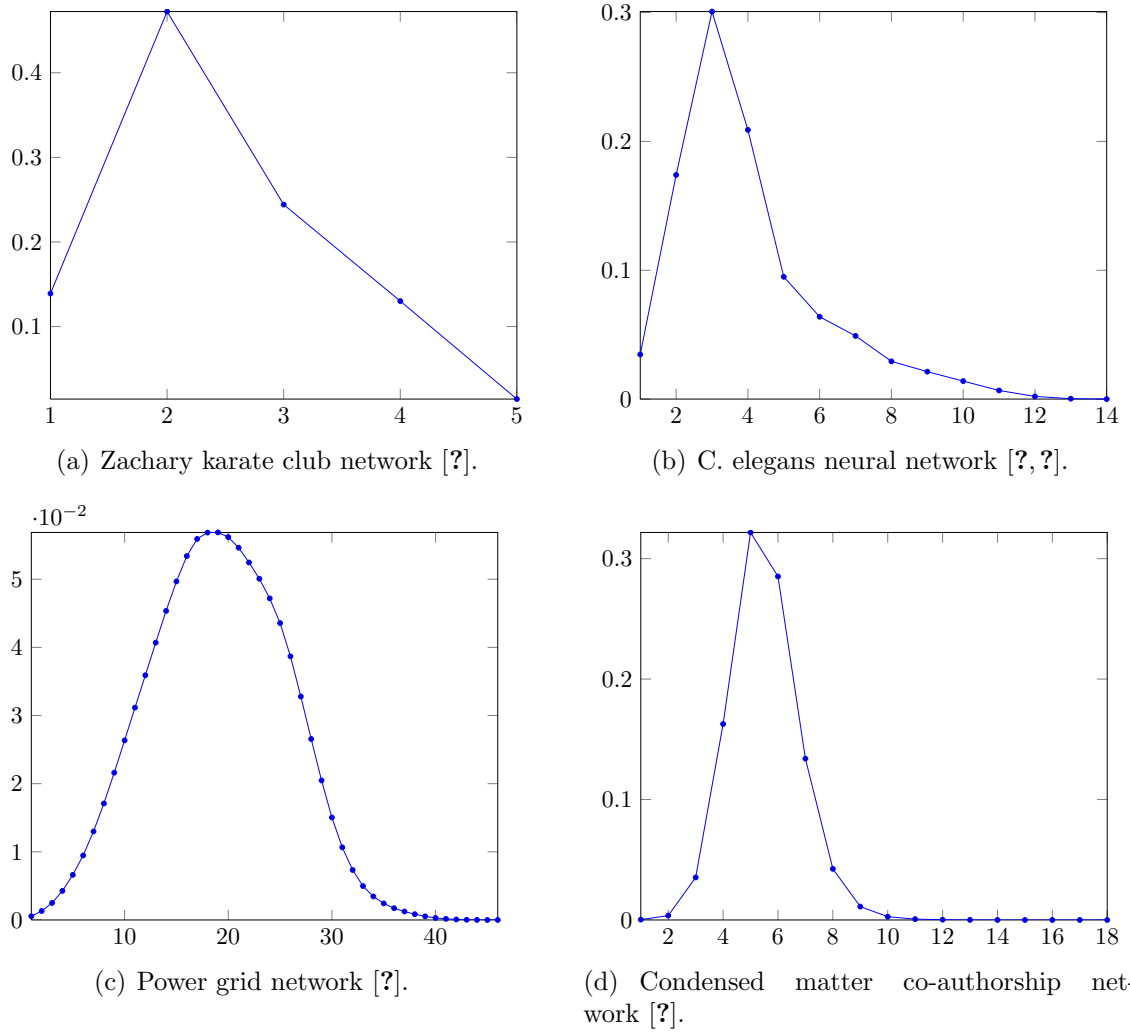
$$d(u,v) = \begin{cases} \infty, & \text{if there is no path from } u \text{ to } v, \\ 0, & \text{if } u = v, \\ k, & \text{where } k \text{ is the length of a shortest } u\text{-}v \text{ path.} \end{cases}$$

If G is strongly connected (respectively, connected for the undirected case) then our distance function is of the form $d : V \times V \to \mathbf{Z}_+ \cup \{0\}$, where the codomain is the set of nonnegative integers. The case where G is not strongly connected (respectively, disconnected for the undirected version) requires special care. One way is to compute the characteristic distance for each component and then find the average of all such characteristic distances. Call the resulting characteristic distance $\overline{d}_c$, where c means component. Another way is to assign a large number as the distance of non-existing shortest paths. If there is no u-v path, we let $d(u, v) = n$ because $n = |V|$ is larger than the length of any shortest path between connected vertices. The resulting characteristic distance is denoted $\overline{d}_b$, where b means big number. Furthermore denote by d_κ the number of pairs (u, v) such that v is not reachable from u. For example, the Zachary [**?**] karate club network has $\overline{d} = 2.4082$ and $d_\kappa = 0$; the C. elegans neural network [**?**, **?**] has $\overline{d}_b = 71.544533$, $\overline{d}_c = 3.991884$, and $d_\kappa = 20,268$; the Western States Power Grid network [**?**] has $\overline{d} = 18.989185$ and $d_\kappa = 0$; and the condensed matter co-authorship network [**?**] has $\overline{d}_b = 7541.74656$, $\overline{d}_c = 5.499329$, and $d_\kappa = 152,328,281$.

We can also define the concept of distance distribution similar to how the degree distribution was defined in section 10.1.1. If ℓ is a positive integer with u and v being connected vertices in a graph $G = (V, E)$, denote by

$$p = \Pr[d(u, v) = \ell] \tag{10.2}$$

the fraction of ordered pairs of connected vertices in $V \times V$ having distance ℓ between them. As is evident from the above notation, we can think of (10.2) as the probability that a uniformly chosen connected pair (u, v) of vertices in G has distance ℓ. The *distance distribution* of G is hence a histogram of the distances between pairs of vertices in G. Figure 10.5 illustrates distance distributions of various real-world networks.



(a) Zachary karate club network [**?**].

(b) C. elegans neural network [**?**, **?**].

(c) Power grid network [**?**].

(d) Condensed matter co-authorship network [**?**].

Figure 10.5: Distance distributions for various real-world networks. The horizontal axis represents distance and the vertical axis represents the probability that a uniformly chosen pair of distinct vertices from the network has the corresponding distance between them.

10.2 Binomial random graph model

In 1959, Gilbert [**?**] introduced a random graph model that now bears the name *binomial* (or *Bernoulli*) random graph model. First, we fix a positive integer n, a probability

Algorithm 10.1: Generate a random graph in $\mathcal{G}(n,p)$.

Input: Positive integer n and a probability $0 < p < 1$.
Output: A random graph from $G(n,p)$.

```
G ← complement of K_n
V ← {0, 1, ..., n − 1}
E ← {2-combinations of V}
for each e ∈ E do
    r ← draw uniformly at random from interval (0, 1)
    if r < p then
        add edge e to G
return G
```

p, and a vertex set $V = \{0, 1, \ldots, n-1\}$. By $\mathcal{G}(n,p)$ we mean a probability space over the set of undirected simple graphs on n vertices. If G is any element of the probability space $\mathcal{G}(n,p)$ and ij is any edge for distinct $i, j \in V$, then ij occurs as an edge of G independently with probability p. In symbols, for any distinct pair $i, j \in V$ we have

$$\Pr[ij \in E(G)] = p$$

where all such events are mutually independent. Any graph G drawn uniformly at random from $\mathcal{G}(n,p)$ is a subgraph of the complete graph K_n and it follows from (1.6) that G has at most $\binom{n}{2}$ edges. Then the probability that G has m edges is given by

$$p^m (1-p)^{\binom{n}{2}-m}. \tag{10.3}$$

Notice the resemblance of (10.3) to the binomial distribution. By $G \in \mathcal{G}(n,p)$ we mean that G is a random graph of the space $\mathcal{G}(n,p)$ and having size distributed as (10.3).

To generate a random graph in $\mathcal{G}(n,p)$, start with G being a graph on n vertices but no edges. That is, initially G is $\overline{K_n}$, the complement of the complete graph on n vertices. Consider each of the $\binom{n}{2}$ possible edges in some order and add it independently to G with probability p. See Algorithm 10.1 for pseudocode of the procedure. The runtime of Algorithm 10.1 depends on an efficient algorithm for generating all 2-combinations of a set of n objects. We could adapt Algorithm 4.22 to our needs or search for a more efficient algorithm; see problem 10.3 for discussion of an algorithm to generate a graph in $\mathcal{G}(n,p)$ in quadratic time. Figure 10.6 illustrates some random graphs from $\mathcal{G}(25,p)$ with $p = i/6$ for $i = 0, 1, \ldots, 5$. See Figure 10.7 for results for graphs in $\mathcal{G}(2 \cdot 10^4,\, p)$.

The expected number of edges of any $G \in \mathcal{G}(n,p)$ is

$$\alpha = \mathrm{E}[|E|] = p \cdot \binom{n}{2} = \frac{pn(n-1)}{2}$$

and the expected total degree is

$$\beta = \mathrm{E}[\#\deg] = 2p \cdot \binom{n}{2} = pn(n-1).$$

Then the expected degree of each edge is $p(n-1)$. From problem 1.7 we know that the number of undirected simple graphs on n vertices is given by

$$2^{n(n-1)/2}$$

where (10.3) is the probability of any of these graphs being the output of the above procedure. Let $\kappa(n,m)$ be the number of graphs from $\mathcal{G}(n,p)$ that are connected and have size m, and by $\Pr[G_\kappa]$ is meant the probability that $G \in \mathcal{G}(n,p)$ is connected. Apply expression (10.3) to see that

$$\Pr[G_\kappa] = \sum_{i=n-1}^{\binom{n}{2}} \kappa(n,i) \cdot p^i (1-p)^{\binom{n}{2}-i}$$

where $n-1$ is the least number of edges of any undirected connected graph on n vertices, i.e. the size of any spanning tree of a connected graph in $\mathcal{G}(n,p)$. Similarly define $\Pr[\kappa_{ij}]$ to be the probability that two distinct vertices i,j of $G \in \mathcal{G}(n,p)$ are connected. Gilbert [**?**] showed that as $n \to \infty$, then we have

$$\Pr[G_\kappa] \sim 1 - n(1-p)^{n-1}$$

and

$$\Pr[\kappa_{ij}] \sim 1 - 2(1-p)^{n-1}.$$

Algorithm 10.2: Random oriented graph via $\mathcal{G}(n,p)$.

Input: Positive integer n and probability $0 < p < 1$.
Output: A random oriented graph on n vertices.

```
G ← random graph in G(n,p) as per Algorithm 10.3
E ← edge set of G
G ← directed version of G
cutoff ← draw uniformly at random from interval (0,1)
for each edge uv ∈ E do
    r ← draw uniformly at random from interval (0,1)
    if r < cutoff then
        remove uv from G
    else
        remove vu from G
return G
```

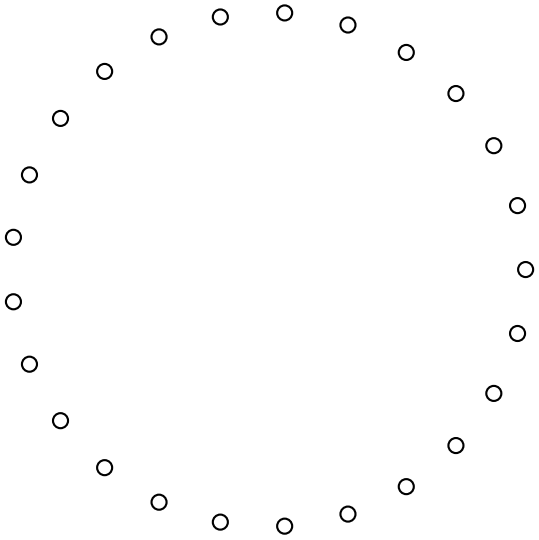
Example 10.1. *Consider a digraph $D = (V,E)$ without self-loops or multiple edges. Then D is said to be* oriented *if for any distinct pair $u,v \in V$ at most one of uv, vu is an edge of D. Provide specific examples of oriented graphs.*

Solution. If $u,v \in V$ is any pair of distinct vertices of an oriented graph $D = (V,E)$, we have various possibilities:

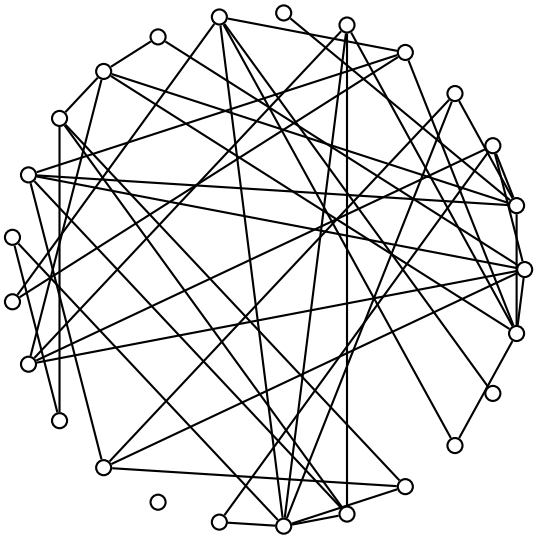
1. $uv \notin E$ and $vu \notin E$.

2. $uv \in E$ and $vu \notin E$.

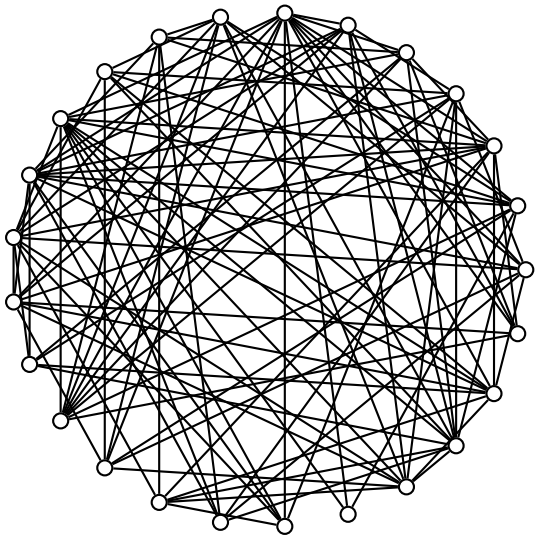
3. $uv \notin E$ and $vu \in E$.



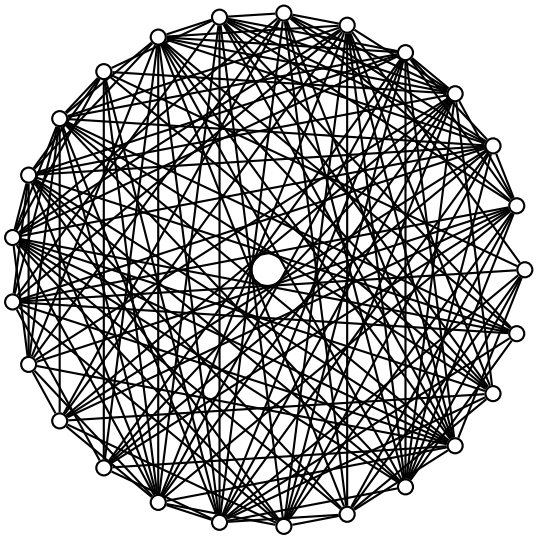
(a) $p = 0$; $\alpha = 0$, $|E| = 0$; $\beta = 0$, $\#\deg = 0$



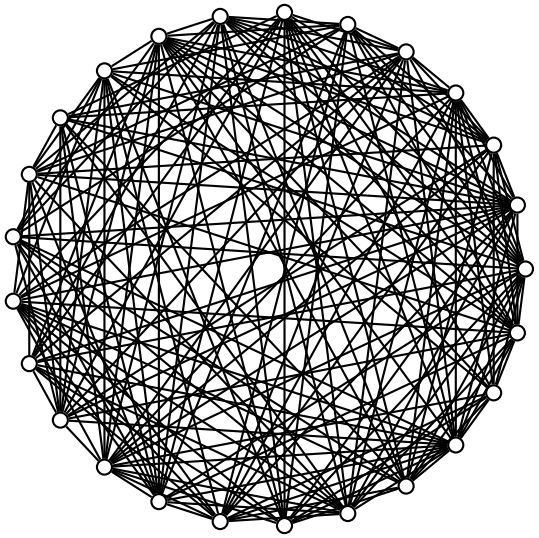
(b) $p = 1/6$; $\alpha = 50$, $|E| = 44$; $\beta = 100$, $\#\deg = 88$



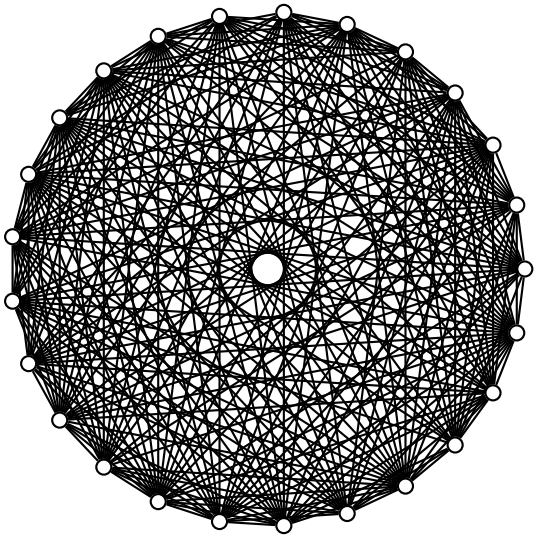
(c) $p = 1/3$; $\alpha = 100$, $|E| = 108$; $\beta = 200$, $\#\deg = 212$



(d) $p = 1/2$; $\alpha = 150$, $|E| = 156$; $\beta = 300$, $\#\deg = 312$



(e) $p = 2/3$; $\alpha = 200$, $|E| = 185$; $\beta = 400$, $\#\deg = 370$



(f) $p = 5/6$; $\alpha = 250$, $|E| = 255$; $\beta = 500$, $\#\deg = 510$

Figure 10.6: Binomial random graphs $\mathcal{G}(25, p)$ for various values of p.

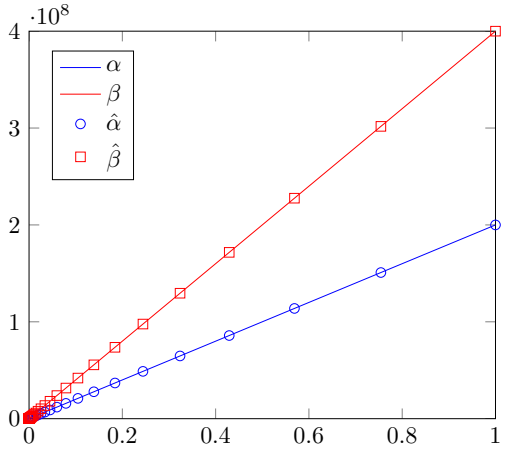


Figure 10.7: Comparison of expected and experimental values of the number of edges and total degree of random simple undirected graphs in $\mathcal{G}(n,p)$. The horizontal axis represents probability points; the vertical axis represents the size and total degree (expected or experimental). Fix $n = 20,000$ and consider $r = 50$ probability points chosen as follows. Let $p_{\min} = 0.000001$, $p_{\max} = 0.999999$, and $F = (p_{\max}/p_{\min})^{1/(r-1)}$. For $i = 1, 2, \ldots, r = 50$ the i-th probability point p_i is defined by $p_i = p_{\min}F^{i-1}$. Each experiment consists in generating $M = 500$ random graphs from $\mathcal{G}(n,p_i)$. For each $G_i \in \mathcal{G}(n,p_i)$, where $i = 1, 2, \ldots, 500$, compute its actual size α_i and actual total degree β_i. Then take the mean $\hat{\alpha}$ of the α_i and the mean $\hat{\beta}$ of the β_i.

Let $n > 0$ be the number of vertices in D and let $0 < p < 1$. Generate a random oriented graph as follows. First we generate a binomial random graph $G \in \mathcal{G}(n,p)$ where G is simple and undirected. Then we consider the digraph version of G and proceed to randomly prune either uv or vu from G, for each distinct pair of vertices u, v. Refer to Algorithm 10.2 for pseudocode of our discussion. A Sage implementation follows:

```
sage: G = graphs.RandomGNP(20, 0.1)
sage: E = G.edges(labels=False)
sage: G = G.to_directed()
sage: cutoff = 0.5
sage: for u, v in E:
...       r = random()
...       if r < cutoff:
...           G.delete_edge(u, v)
...       else:
...           G.delete_edge(v, u)
```

which produced the random oriented graph in Figure 10.8. ■

Efficient generation of sparse $G \in \mathcal{G}(n,p)$

The techniques discussed so far (Algorithms 10.1 and 10.9) for generating a random graph from $\mathcal{G}(n,p)$ can be unsuitable when the number of vertices n is in the hundreds of thousands or millions. In many applications of $\mathcal{G}(n,p)$ we are only interested in sparse random graphs. A linear time algorithm to generate a random sparse graph from $\mathcal{G}(n,p)$ is presented by Batagelj and Brandes [**?**].

The Batagelj-Brandes algorithm for generating a random sparse graph $G \in \mathcal{G}(n,p)$ uses what is known as a geometric method to skip over certain edges. Fix a probability $0 < p < 1$ that an edge will be in the resulting random sparse graph G. If e is an edge

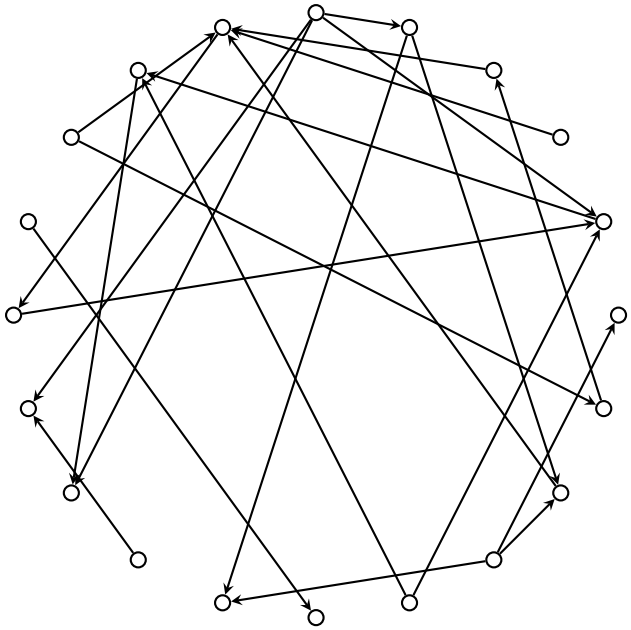


Figure 10.8: A random oriented graph generated using a graph in $G(20, 0.1)$ and cutoff probability 0.5.

of G, we can consider the events leading up to the choice of e as

$$e_1, e_2, \ldots, e_k$$

where in the i-th trial the event e_i is a failure, for $1 \leq i < k$, but the event e_k is the first success after $k-1$ successive failures. In probabilistic terms, we perform a series of independent trials each having success probability p and stop when the first success occurs. Letting X be the number of trials required until the first success occurs, then X is a geometric random variable with parameter p and probability mass function

$$\Pr[X = k] = p(1-p)^{k-1} \tag{10.4}$$

for integers $k \geq 1$, where

$$\sum_{k=1}^{\infty} p(1-p)^{k-1} = 1.$$

In other words, waiting times are geometrically distributed.

Suppose we want to generate a random number from a geometric distribution, i.e. we want to simulate X such that

$$\Pr[X = k] = p(1-p)^{k-1}, \qquad k = 1, 2, 3, \ldots$$

Note that

$$\sum_{k=1}^{\ell} \Pr[X = k] = 1 - \Pr[X > \ell - 1] = 1 - (1-p)^{\ell-1}.$$

In other words, we can simulate a geometric random variable by generating r uniformly at random from the interval $(0, 1)$ and set X to that value of k for which

$$1 - (1-p)^{k-1} < r < 1 - (1-p)^{k}$$

or equivalently for which

$$(1-p)^k < 1-r < (1-p)^{k-1}$$

where $1-r$ and r are both uniformly distributed. Thus we can define X by

$$\begin{aligned} X &= \min\{k \mid (1-p)^k < 1-r\} \\ &= \min\left\{k \;\middle|\; k > \frac{\ln(1-r)}{\ln(1-p)}\right\} \\ &= 1 + \left\lfloor \frac{\ln(1-r)}{\ln(1-p)} \right\rfloor. \end{aligned}$$

That is, we can choose k to be

$$k = 1 + \left\lfloor \frac{\ln(1-r)}{\ln(1-p)} \right\rfloor$$

which is used as a basis of Algorithm 10.3. In the latter algorithm, note that the vertex set is $V = \{0, 1, \ldots, n-1\}$ and candidate edges are generated in lexicographic order. The Batagelj-Brandes Algorithm 10.3 has worst-case runtime $O(n+m)$, where n and m are the order and size, respectively, of the resulting graph.

Algorithm 10.3: Linear generation of a random sparse graph in $\mathcal{G}(n,p)$.

Input: Positive integer n and a probability $0 < p < 1$.
Output: A random sparse graph from $G(n,p)$.

```
1  G ← complement(K_n)
2  u ← 1
3  v ← −1
4  while u < n do
5      r ← draw uniformly at random from interval (0, 1)
6      v ← v + 1 + ⌊ln(1 − r)/ ln(1 − p)⌋
7      while v ≥ u and u < n do
8          v ← v − u
9          u ← u + 1
10     if u < n then
11         add edge uv to G
12 return G
```

Degree distribution

Consider a random graph $G \in \mathcal{G}(n,p)$ and let v be a vertex of G. With probability p, the vertex v is incident with each of the remaining $n-1$ vertices in G. Then the probability that v has degree k is given by the binomial distribution

$$\Pr[\deg(v) = k] = \binom{n-1}{k} p^k (1-p)^{n-1-k} \tag{10.5}$$

and the expected degree of v is $\mathrm{E}[\deg(v)] = p(n-1)$. Setting $z = p(n-1)$, we can express (10.5) as

$$\Pr[\deg(v) = k] = \binom{n-1}{k} \left(\frac{z}{n-1-z}\right)^k \left(1 - \frac{z}{n-1}\right)^{n-1}$$

and thus

$$\Pr[\deg(v) = k] \to \frac{z^k}{k!} \exp(-z)$$

as $n \to \infty$. In the limit of large n, the probability that vertex v has degree k approaches the Poisson distribution. That is, as n gets larger and larger any random graph in $\mathcal{G}(n,p)$ has a Poisson degree distribution.

10.3 Erdős-Rényi model

Let N be a fixed nonnegative integer. The *Erdős-Rényi* [**?**,**?**] (or *uniform*) random graph model, denoted $\mathcal{G}(n,N)$, is a probability space over the set of undirected simple graphs on n vertices and exactly N edges. Hence $\mathcal{G}(n,N)$ can be considered as a collection of $\binom{\binom{n}{2}}{N}$ undirected simple graphs on exactly N edges, each such graph being selected with equal probability. A note of caution is in order here. Numerous papers on random graphs refer to $\mathcal{G}(n,p)$ as the Erdős-Rényi random graph model, where in fact this binomial random graph model should be called the Gilbert model in honor of E. N. Gilbert who introduced [**?**] it in 1959. Whenever a paper makes a reference to the Erdős-Rényi model, one should question whether the paper is referring to $\mathcal{G}(n,p)$ or $\mathcal{G}(n,N)$.

To generate a graph in $\mathcal{G}(n,N)$, start with G being a graph on n vertices but no edges. Then choose N of the possible $\binom{n}{2}$ edges independently and uniformly at random and let the chosen edges be the edge set of G. Each graph $G \in \mathcal{G}(n,N)$ is associated with a probability

$$1 \Big/ \binom{\binom{n}{2}}{N}$$

of being the graph resulting from the above procedure. Furthermore each of the $\binom{n}{2}$ edges has a probability

$$1 \Big/ \binom{n}{2}$$

of being chosen. Algorithm 10.4 presents a straightforward translation of the above procedure into pseudocode.

The runtime of Algorithm 10.4 is probabilistic and can be analyzed via the geometric distribution. If i is the number of edges chosen so far, then the probability of choosing a new edge in the next step is

$$\frac{\binom{n}{2} - i}{\binom{n}{2}}.$$

We repeatedly choose an edge uniformly at random from the collection of all possible edges, until we come across the first edge that is not already in the graph. The number of trials required until the first new edge is chosen can be modeled using the geometric distribution with probability mass function (10.4). Given a geometric random variable

Algorithm 10.4: Generation of random graph in $\mathcal{G}(n, N)$.

Input: Positive integer n and integer N with $0 \leq N \leq \binom{n}{2}$.
Output: A random graph from $G(n, N)$.

```
1 G ← K_n (complement)
2 E ← {e_0, e_1, ..., e_{C(n,2)-1}}
3 for i ← 0, 1, ..., N − 1 do
4     r ← draw uniformly at random from {0, 1, ..., C(n,2) − 1}
5     while e_r is an edge of G do
6         r ← draw uniformly at random from {0, 1, ..., C(n,2) − 1}
7     add edge e_r to G
8 return G
```

X, we have the expectation

$$\mathrm{E}[X] = \sum_{n=1}^{\infty} n \cdot p(1-p)^{n-1} = \frac{1}{p}.$$

Therefore the expected number of trials until a new edge be chosen is

$$\frac{\binom{n}{2}}{\binom{n}{2} - i}$$

from which the expected total runtime is

$$\begin{aligned}\sum_{i=1}^{N} \frac{\binom{n}{2}}{\binom{n}{2} - i} &\approx \int_0^N \frac{\binom{n}{2}}{\binom{n}{2} - x}\, dx \\ &= \binom{n}{2} \cdot \ln \frac{\binom{n}{2}}{\binom{n}{2} - N} + C\end{aligned}$$

for some constant C. The denominator in the latter fraction becomes zero when $\binom{n}{2} = N$, which can be prevented by adding one to the denominator. Then we have the expected total runtime

$$\sum_{i=1}^{N} \frac{\binom{n}{2}}{\binom{n}{2} - i} \in \Theta\left(\binom{n}{2} \cdot \ln \frac{\binom{n}{2}}{\binom{n}{2} - N + 1}\right)$$

which is $O(N)$ when $N \leq \binom{n}{2}/2$, and $O(N \ln N)$ when $N = \binom{n}{2}$. In other words, Algorithm 10.4 has expected linear runtime when the number N of required edges satisfies $N \leq \binom{n}{2}/2$. But for $N > \binom{n}{2}/2$, we obtain expected linear runtime by generating the complete graph K_n and randomly delete $\binom{n}{2} - N$ edges from the latter graph. Our discussion is summarized in Algorithm 10.5.

Algorithm 10.5: Generation of random graph in $\mathcal{G}(n, N)$ in expected linear time.

Input: Positive integer n and integer N with $0 \leq N \leq \binom{n}{2}$.
Output: A random graph from $G(n, N)$.

```
1 if N ≤ (n choose 2)/2 then
2     return result of Algorithm 10.4
3 G ← K_n
4 for i ← 1, 2, ..., (n choose 2) − N do
5     e ← draw uniformly at random from E(G)
6     remove edge e from G
7 return G
```

10.4 Small-world networks

Vicky: Hi, Janice.

Janice: Hi, Vicky.

Vicky: How are you?

Janice: Good.

Harry: You two know each other?

Janice: Yeah, I met Vicky at the mall today.

Harry: Well, what a small world! You know, I wonder who else I know knows someone I know that I don't know knows that person I know.

— from the TV series *Third Rock from the Sun*, season 5, episode 22, 2000.

Many real-world networks exhibit the *small-world effect*: that most pairs of distinct vertices in the network are connected by relatively short path lengths. The small-world effect was empirically demonstrated [**?**] in a famous 1960s experiment by Stanley Milgram, who distributed a number of letters to a random selection of people. Recipients were instructed to deliver the letters to the addressees on condition that letters must be passed to people whom the recipients knew on a first-name basis. Milgram found that on average six steps were required for a letter to reach its target recipient, a number now immortalized in the phrase "six degrees of separation" [**?**]. Figure 10.9 plots results of an experimental study of the small-world problem as reported in [**?**]. The small-world effect has been studied and verified for many real-world networks including

- social: collaboration network of actors in feature films [**?**, **?**], scientific publication authorship [**?**, **?**, **?**, **?**];
- information: citation network [**?**], Roget's Thesaurus [**?**], word co-occurrence [**?**, **?**];
- technological: internet [**?**, **?**], power grid [**?**], train routes [**?**], software [**?**, **?**];
- biological: metabolic network [**?**], protein interactions [**?**], food web [**?**, **?**], neural network [**?**, **?**].

Watts and Strogatz [**?**, **?**, **?**] proposed a network model that produces graphs exhibiting the small-world effect. We will use the notation "$\gg$" to mean "much greater than". Let n and k be positive integers such that $n \gg k \gg \ln n \gg 1$ (in particular, $0 < k < n/2$) with k being even. Consider a probability $0 < p < 1$. Starting from an undirected

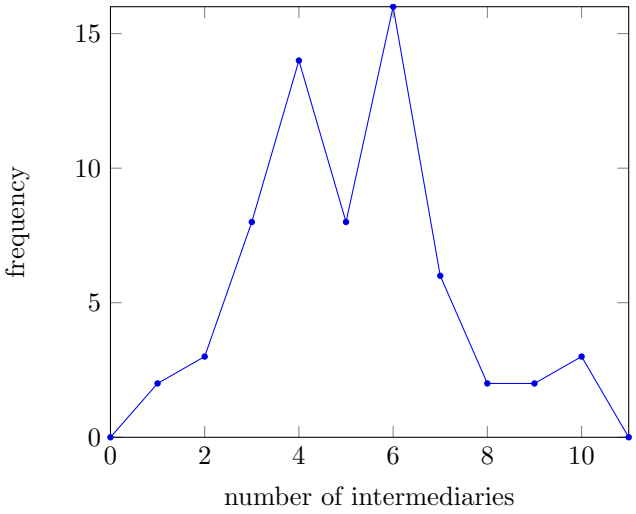
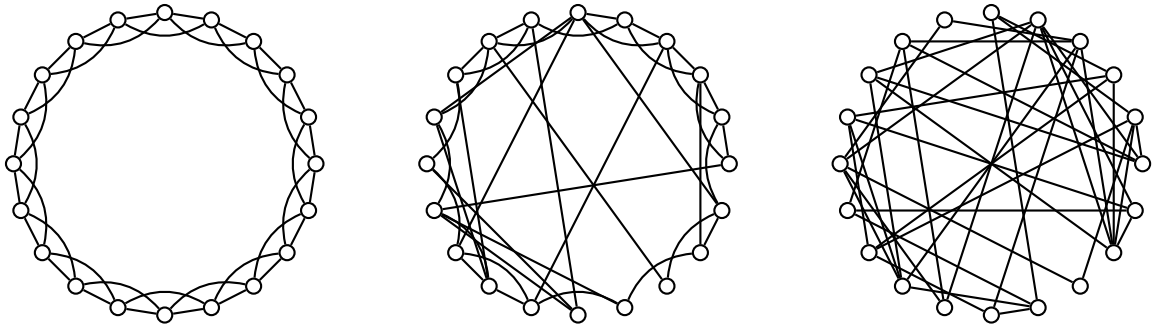


Figure 10.9: Frequency distribution of the number of intermediaries required for letters to reach their intended addressees. The distribution has a mean of 5.3, interpreted as the average number of intermediaries required for a letter to reach its intended destination. The plot is derived from data reported in [**?**].

k-circulant graph $G = (V, E)$ on n vertices, the Watts-Strogatz model proceeds to rewire each edge with probability p. The rewiring procedure, also called edge swapping, works as follows. Let V be uniformly distributed. For each $v \in V$, let $e \in E$ be an edge having v as an endpoint. Choose another $u \in V$ different from v. With probability p, delete the edge e and add the edge vu. The rewiring must produce a simple graph with the same order and size as G. As $p \to 1$, the graph G goes from k-circulant to exhibiting properties of graphs drawn uniformly from $\mathcal{G}(n, p)$. Small-world networks are intermediate between k-circulant and binomial random graphs (see Figure 10.10). The Watts-Strogatz model is said to provide a procedure for interpolating between the latter two types of graphs.



(a) $p = 0$, k-circulant (b) $p = 0.3$, small-world (c) $p = 1$, random

Figure 10.10: With increasing randomness, k-circulant graphs evolve to exhibit properties of random graphs in $\mathcal{G}(n, p)$. Small-world networks are intermediate between k-circulant graphs and random graphs in $\mathcal{G}(n, p)$.

The last paragraph contains an algorithm for rewiring edges of a graph. While the algorithm is simple, in practice it potentially skips over a number of vertices to be considered for rewiring. If $G = (V, E)$ is a k-circulant graph on n vertices and p is the rewiring probability, the candidate vertices to be rewired follow a geometric distribution

with parameter p. This geometric trick, essentially the same speed-up technique used by the Batagelj-Brandes Algorithm 10.3, can be used to speed up the rewiring algorithm. To elaborate, suppose G has vertex set $V = \{0, 1, \ldots, n-1\}$. If r is chosen uniformly at random from the interval $(0, 1)$, the index of the vertex to be rewired can be obtained from

$$1 + \left\lfloor \frac{\ln(1-r)}{\ln(1-p)} \right\rfloor .$$

The above geometric method is incorporated into Algorithm 10.6 to generate a Watts-Strogatz network in worst-case runtime $O(nk+m)$, where n and k are as per the input of the algorithm and m is the size of the k-circulant graph on n vertices. Note that lines 7 to 12 are where we avoid self-loops and multiple edges.

Algorithm 10.6: Watts-Strogatz network model.

Input: Positive integer n denoting the number of vertices. Positive even integer k for the degree of each vertex, where $n \gg k \gg \ln n \gg 1$. In particular, k should satisfy $0 < k < n/2$. Rewiring probability $0 < p \leq 1$.

Output: A Watts-Strogatz network on n vertices.

```
1  M ← nk    /* sum of all vertex degrees = twice number of edges */
2  r ← draw uniformly at random from interval (0, 1)
3  v ← 1 + ⌊ln(1 − r)/ ln(1 − p)⌋
4  E ← contiguous edge list of k-circulant graph on n vertices
5  while v ≤ M do
6      u ← draw uniformly at random from [0, 1, ..., n − 1]
7      if v − 1 is even then
8          while E[v] = u or (u, E[v]) ∈ E do
9              u ← draw uniformly at random from [0, 1, ..., n − 1]
10     else
11         while E[v − 2] = u or (E[v − 2], u) ∈ E do
12             u ← draw uniformly at random from [0, 1, ..., n − 1]
13     E[v − 1] ← u
14     r ← draw uniformly at random from interval (0, 1)
15     v ← v + 1 + ⌊ln(1 − r)/ ln(1 − p)⌋
16 G ← K̄_n
17 add edges in E to G
18 return G
```

Characteristic path length

Watts and Strogatz [**?**] analyzed the structure of networks generated by Algorithm 10.6 via two quantities: the *characteristic path length* ℓ and the *clustering coefficient* C. The characteristic path length quantifies the average distance between any distinct pair of vertices in a Watts-Strogatz network. The quantity $\ell(G)$ is thus said to be a global property of G. Watts and Strogatz characterized as *small-world* those networks that exhibit high clustering coefficients and low characteristic path lengths.

Let $G = (V, E)$ be a Watts-Strogatz network as generated by Algorithm 10.6, where the vertex set is $V = \{0, 1, \ldots, n-1\}$. For each pair of vertices $i, j \in V$, let d_{ij} be the

distance from i to j. If there is no path from i to j or $i = j$, set $d_{ij} = 0$. Thus

$$d_{ij} = \begin{cases} 0, & \text{if there is no path from } i \text{ to } j, \\ 0, & \text{if } i = j, \\ k, & \text{where } k \text{ is the length of a shortest path from } i \text{ to } j. \end{cases}$$

Since G is undirected, we have $d_{ij} = d_{ji}$. Consequently when computing the distance between each distinct pair of vertices, we should avoid double counting by computing d_{ij} for $i < j$. Then the characteristic path length of G is defined by

$$\begin{aligned} \ell(G) &= \frac{1}{n(n-1)/2} \cdot \frac{1}{2} \sum_{i \neq j} d_{ij} \\ &= \frac{1}{n(n-1)} \sum_{i \neq j} d_{ij} \end{aligned} \tag{10.6}$$

which is averaged over all possible pairs of distinct vertices, i.e. the number of edges in the complete graph K_n.

It is inefficient to compute the characteristic path length via equation (10.6) because we would effectively sum $n(n-1)$ distance values. As G is undirected, note that

$$\frac{1}{2} \sum_{i \neq j} d_{ij} = \sum_{i<j} d_{ij} = \sum_{i>j} d_{ij}.$$

The latter equation holds for the following reason. Let $D = [d_{ij}]$ be a matrix of distances for G, where i is the row index, j is the column index, and d_{ij} is the distance from i to j. The required sum of distances can be obtained by summing all entries above (or below) the main diagonal of D. Therefore the characteristic path length can be expressed as

$$\begin{aligned} \ell(G) &= \frac{2}{n(n-1)} \sum_{i<j} d_{ij} \\ &= \frac{2}{n(n-1)} \sum_{i>j} d_{ij} \end{aligned}$$

which requires summing $\frac{n(n-1)}{2}$ distance values.

Let $G = (V, E)$ be a Watts-Strogatz network with $n = |V|$. Set $k' = k/2$, where k is as per Algorithm 10.6. As the rewiring probability $p \to 0$, the average path length tends to

$$\ell \to \frac{n}{4k'} = \frac{n}{2k}.$$

In the special case $p = 0$, we have

$$\ell = \frac{n(n+k-2)}{2k(n-1)}.$$

However as $p \to 1$, we have $\ell \to \frac{\ln n}{\ln k}$.

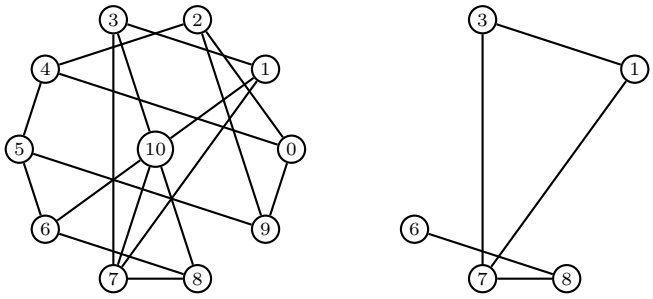
Clustering coefficient

The *clustering coefficient* of a simple graph G quantifies the "cliquishness" of vertices in $G = (V, E)$. This quantity is thus said to be a local property of G. Watts and Strogatz [**?**] defined the clustering coefficient as follows. Suppose $n = |V| > 0$ and let n_i count the number of neighbors of vertex $i \in V$, a quantity that is equivalent to the degree of i, i.e. $\deg(i) = n_i$. The complete graph K_{n_i} on the n_i neighbors of i has $n_i(n_i - 1)/2$ edges. The *neighbor graph* $\mathcal{N}_i$ of i is a subgraph of G, consisting of all vertices ($\neq i$) that are adjacent to i and preserving the adjacency relation among those vertices as found in the supergraph G. For example, given the graph in Figure 10.11(a) the neighbor graph of vertex 10 is shown in Figure 10.11(b). The local clustering coefficient C_i of i is the ratio

$$C_i = \frac{N_i}{n_i(n_i - 1)/2}$$

where N_i counts the number of edges in $\mathcal{N}_i$. In case i has degree $\deg(i) < 2$, we set the local clustering coefficient of i to be zero. Then the clustering coefficient of G is defined by

$$C(G) = \frac{1}{n}\sum_{i \in V} C_i = \frac{1}{n}\sum_{i \in V} \frac{N_i}{n_i(n_i - 1)/2}.$$



(a) Graph on 11 vertices. (b) $\mathcal{N}_{10}$

Figure 10.11: The neighbor graph of a vertex.

Consider the case where we have a k-circulant graph $G = (V, E)$ on n vertices and a rewiring probability $p = 0$. That is, we do not rewire any edge of G. Each vertex of G has degree k. Let $k' = k/2$. Then the k neighbors of each vertex in G has $3k'(k' - 1)/2$ edges between them, i.e. each neighbor graph $\mathcal{N}_i$ has size $3k'(k' - 1)/2$. Then the clustering coefficient of G is

$$\frac{3(k' - 1)}{2(2k' - 1)}.$$

When the rewiring probability is $p > 0$, Barrat and Weigt [**?**] showed that the clustering coefficient of any graph G' in the Watts-Strogatz network model (see Algorithm 10.6) can be approximated by

$$C(G') \approx \frac{3(k' - 1)}{2(2k' - 1)}(1 - p)^3.$$

Degree distribution

For a Watts-Strogatz network without rewiring, each vertex has the same degree k. It easily follows that for each vertex v, we have the degree distribution

$$\Pr[\deg(v) = i] = \begin{cases} 1, & \text{if } i = k, \\ 0, & \text{otherwise.} \end{cases}$$

A rewiring probability $p > 0$ introduces disorder in the network and broadens the degree distribution, while the expected degree is k. A k-circulant graph on n vertices has $nk/2$ edges. With the rewiring probability $p > 0$, a total of $pnk/2$ edges would be rewired. However note that only one endpoint of an edge is rewired, thus after the rewiring process the degree of any vertex v is $\deg(v) \geq k/2$. Therefore with $k > 2$, a Watts-Strogatz network has no isolated vertices.

For $p > 0$, Barrat and Weigt [**?**] showed that the degree of a vertex v can be written as $\deg(v) = k/2 + n_i$ with $n_i \geq 0$, where n_i can be divided into two parts α and β as follows. First $\alpha \leq k/2$ edges are left intact after the rewiring process, the probability of this occurring is $1-p$ for each edge. Second $\beta = n_i - \alpha$ edges have been rewired towards i, each with probability $1/n$. The probability distribution of α is

$$P_1(\alpha) = \binom{k/2}{\alpha}(1-p)^{\alpha}p^{k/2-\alpha}$$

and the probability distribution of β is

$$P_2(\beta) = \binom{pnk/2}{\beta}\left(\frac{1}{n}\right)^{\beta}\left(1-\frac{1}{n}\right)^{pnk/2-\beta}$$

where

$$P_2(\beta) \to \frac{(pk/2)^{\beta}}{\beta!}\exp(-pk/2)$$

for large n. Combine the above two factors to obtain the degree distribution

$$\Pr[\deg(v) = \kappa] = \sum_{i=0}^{\min\{\kappa-k/2,\, k/2\}} \binom{k/2}{i}(1-p)^i p^{k/2-i}\frac{(pk/2)^{\kappa-k/2-i}}{(\kappa-k/2-i)!}\exp(-pk/2)$$

for $\kappa \geq k/2$.

10.5 Scale-free networks

The networks covered so far—Gilbert $\mathcal{G}(n,p)$ model, Erdős-Rényi $\mathcal{G}(n,N)$ model, Watts-Strogatz small-world model—are static. Once a network is generated from any of these models, the corresponding model does not specify any means for the network to evolve over time. Barabási and Albert [**?**] proposed a network model based on two ingredients:

1. Growth: at each time step, a new vertex is added to the network and connected to a pre-determined number of existing vertices.

2. Preferential attachment: the newly added vertex is connected to an existing vertex in proportion to the latter's existing degree.

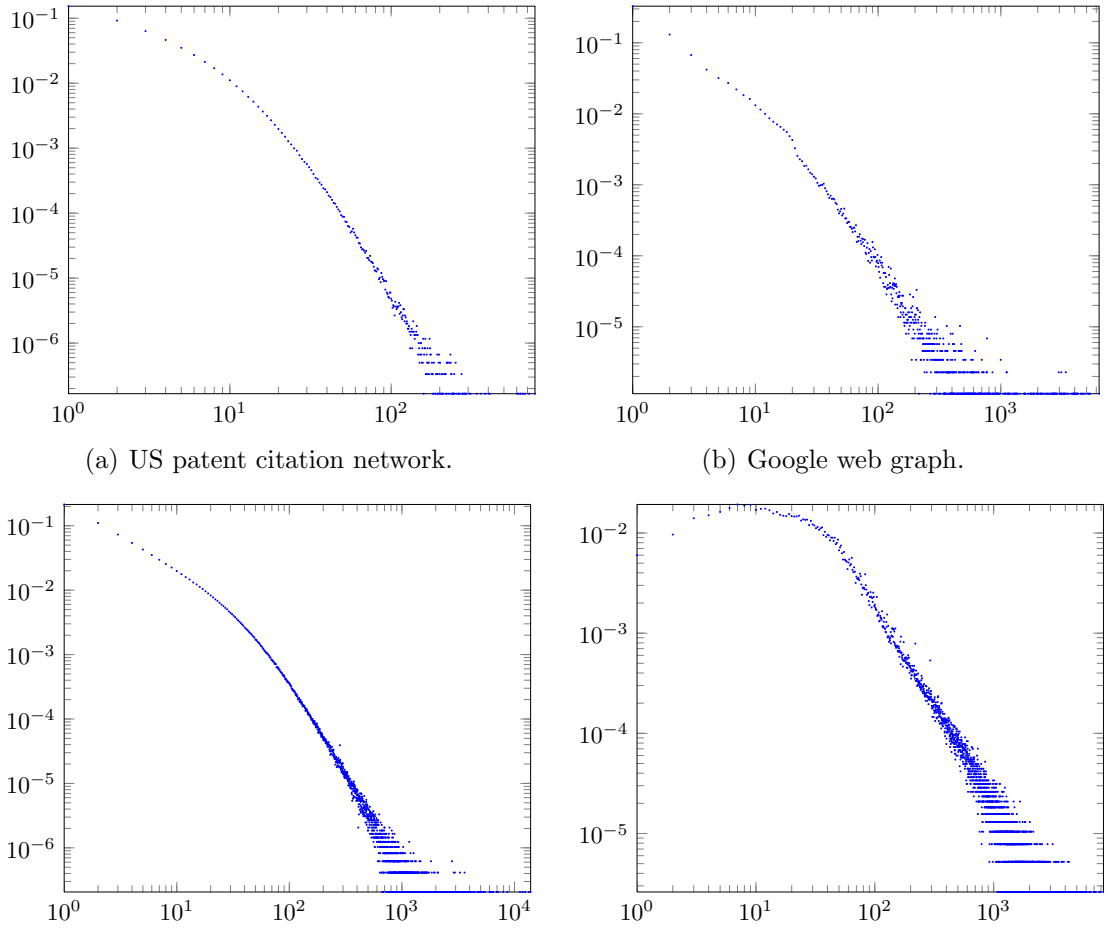
Preferential attachment also goes by the colloquial name of the "rich-get-richer" effect due to the work of Herbert Simon [**?**]. In sociology, preferential attachment is known as the *Matthew effect* due to the following verse from the Book of Matthew, chapter 25 verse 29, in the Bible: "For to every one that hath shall be given but from him that hath not, that also which he seemeth to have shall be taken away." Barabási and Albert observed that many real-world networks exhibit statistical properties of their proposed model. One particularly significant property is that of power-law scaling, hence the Barabási-Albert model is also called a model of scale-free networks. Note that it is only the degree distributions of scale-free networks that are scale-free. In their empirical study of the World Wide Web (WWW) and other real-world networks, Barabási and Albert noted that the probability that a web page increases in popularity is directly proportional to the page's current popularity. Thinking of a web page as a vertex and the degree of a page as the number of other pages that the current page links to, the degree distribution of the WWW follows a power law function. Power-law scaling has been confirmed for many real-world networks:

- actor collaboration network [**?**]
- citation [**?**, **?**, **?**] and co-authorship networks [**?**]
- human sexual contacts network [**?**, **?**]
- the Internet [**?**, **?**, **?**] and the WWW [**?**, **?**, **?**]
- metabolic networks [**?**, **?**]
- telephone call graphs [**?**, **?**]

Figure 10.12 illustrates the degree distributions of various real-world networks, plotted on log-log scales. Corresponding distributions for various simulated Barabási-Albert networks are illustrated in Figure 10.13.

But how do we generate a scale-free graph as per the description in Barabási and Albert [**?**]? The original description of the Barabási-Albert model as contained in [**?**] is rather ambiguous with respect to certain details. First, the whole process is supposed to begin with a small number of vertices. But as the degree of each of these vertices is zero, it is unclear how the network is to grow via preferential attachment from the initial pool of vertices. Second, Barabási and Albert neglected to clearly specify how to select the neighbors for the newly added vertex. The above ambiguities are resolved by Bollobás et al. [**?**], who gave a precise statement of a random graph process that realizes the Barabási-Albert model. Fix a sequence of vertices $v_1, v_2, \ldots$ and consider the case where each newly added vertex is to be connected to $m = 1$ vertex already in a graph. Inductively define a random graph process $(G_1^t)_{t\geq 0}$ as follows, where G_1^t is a digraph on $\{v_i \mid 1 \leq i \leq t\}$. Start with the null graph G_1^0 or the graph G_1^1 with one vertex and one self-loop. Denote by $\deg_G(v)$ the total (in and out) degree of vertex v in the graph G. For $t > 1$ construct G_1^t from G_1^{t-1} by adding the vertex v_t and a directed edge from v_t to v_i, where i is randomly chosen with probability

$$\Pr[i = s] = \begin{cases} \deg_{G_1^{t-1}}(v_s)/(2t-1), & \text{if } 1 \leq s \leq t-1, \\ 1/(2t-1), & \text{if } s = t. \end{cases}$$



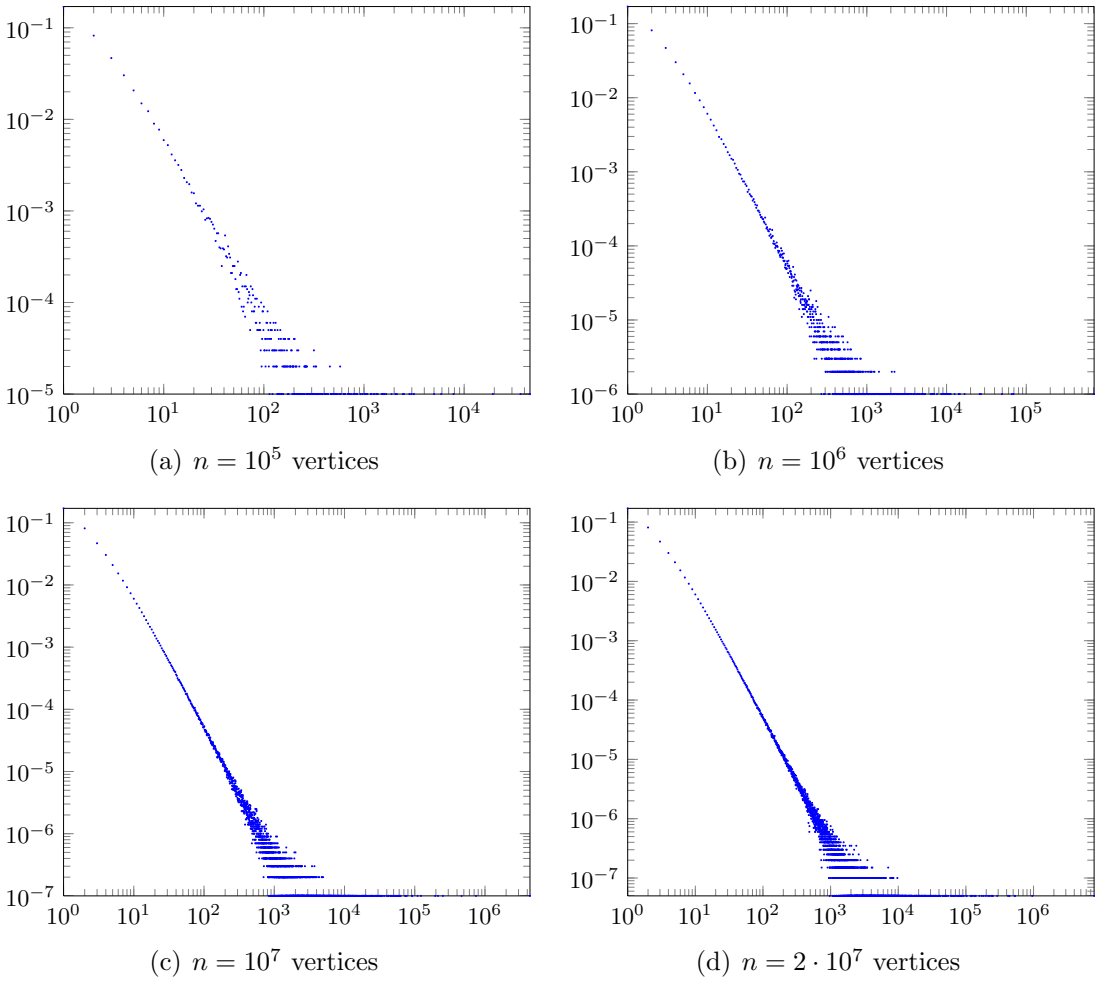
(a) US patent citation network.

(b) Google web graph.

(c) LiveJournal friendship network.

(d) Actor collaboration network.

Figure 10.12: Degree distributions of various real-world networks on log-log scales. The horizontal axis represents degree and the vertical axis is the corresponding probability of a vertex having that degree. The US patent citation network [**?**] is a directed graph on $3,774,768$ vertices and $16,518,948$ edges. It covers all citations made by patents granted between 1975 and 1999. The Google web graph [**?**] is a digraph having $875,713$ vertices and $5,105,039$ edges. This dataset was released in 2002 by Google as part of the Google Programming Contest. The LiveJournal friendship network [**?**, **?**] is a directed graph on $4,847,571$ vertices and $68,993,773$ edges. The actor collaboration network [**?**], based on the Internet Movie Database (IMDb) at http://www.imdb.com, is an undirected graph on $383,640$ vertices and $16,557,920$ edges. Two actors are connected to each other if they have starred in the same movie. In all of the above degree distributions, self-loops are not taken into account and, where a graph is directed, we only consider the in-degree distribution.



(a) $n = 10^5$ vertices

(b) $n = 10^6$ vertices

(c) $n = 10^7$ vertices

(d) $n = 2 \cdot 10^7$ vertices

Figure 10.13: Degree distributions of simulated graphs in the classic Barabási-Albert model. The horizontal axis represents degree; the vertical axis is the corresponding probability of a vertex having a particular degree. Each generated graph is directed and has minimum out-degree $m = 5$. The above degree distributions are only for in-degrees and do not take into account self-loops.

The latter process generates a forest. For $m > 1$ the graph evolves as per the case $m = 1$; i.e. we add m edges from v_t one at a time. This process can result in self-loops and multiple edges. We write $\mathcal{G}_m^n$ for the collection of all graphs on n vertices and minimal degree m in the Barabási-Albert model, where a random graph from $\mathcal{G}_m^n$ is denoted $G_m^n \in \mathcal{G}_m^n$.

Now consider the problem of translating the above procedure into pseudocode. Fix a positive integer $n > 1$ for the number of vertices in the scale-free graph to be generated via preferential attachment. Let $m \geq 1$ be the number of vertices that each newly added vertex is to be connected to; this is equivalent to the minimum degree that any new vertex will end up possessing. At any time step, let M be the contiguous edge list of all edges created thus far in the above random graph process. It is clear that the frequency (or number of occurrences) of a vertex is equivalent to the vertex's degree. We can thus use M as a pool to sample in constant time from the degree-skewed distribution. Batagelj and Brandes [**?**] used the latter observation to construct an algorithm for generating scale-free networks via preferential attachment; pseudocode is presented in Algorithm 10.7. Note that the algorithm has linear runtime $O(n + m)$, where n is the order and m the size of the graph generated by the algorithm.

Algorithm 10.7: Scale-free network via preferential attachment.

Input: Positive integer $n > 1$ and minimum degree $d \geq 1$.
Output: Scale-free network on n vertices.

```
G ← K_n                    /* vertex set is V = {0, 1, ..., n − 1} */
M ← list of length 2nd
for v ← 0, 1, ..., n − 1 do
    for i ← 0, 1, ..., d − 1 do
        M[2(vd + i)] ← v
        r ← draw uniformly at random from {0, 1, ..., 2(vd + i)}
        M[2(vd + i) + 1] ← M[r]
add edge (M[2i], M[2i + 1]) to G for i ← 0, 1, ..., nd − 1
return G
```

On the evidence of computer simulation and various real-world networks, it was suggested [**?**, **?**] that $\Pr[\deg(v) = k] \sim k^{-\gamma}$ with $\gamma = 2.9 \pm 0.1$. Letting n be the number of vertices, Bollobás et al. [**?**] obtained $\Pr[\deg(v) = k]$ asymptotically for all $k \leq n^{1/15}$ and showed as a consequence that $\gamma = 3$. In the process of doing so, Bollobás et al. proved various results concerning the expected degree. Denote by $\#_m^n(k)$ the number of vertices of G_m^n with in-degree k (and consequently with total degree $m+k$). For the case $m = 1$, we have the expectation

$$\mathrm{E}[\deg_{G_1^t}(v_t)] = 1 + \frac{1}{2t-1}$$

and for $s < t$ we have

$$\mathrm{E}[\deg_{G_1^t}(v_s)] = \frac{2t}{2t-1}\mathrm{E}[\deg_{G_1^{t-1}}(v_s)].$$

Taking the above two equations together, for $1 \leq s \leq n$ we have

$$\mathrm{E}[\deg_{G_1^n}(v_s)] = \prod_{i=s}^{n} \frac{2i}{2i-1} = \frac{4^{n-s+1}n!^2(2s-2)!}{(2n)!(s-1)!^2}.$$

Furthermore for $0 \leq k \leq n^{1/15}$ we have

$$\mathrm{E}[\#_m^n(k)] \sim \frac{2m(m+1)n}{(k+m)(k+m+1)(k+m+2)}$$

uniformly in k.

As regards the diameter, with n as per Algorithm 10.7, computer simulation by Barabási, Albert, and Jeong [**?**, **?**] and heuristic arguments by Newman et al. [**?**] suggest that a graph generated by the Barabási-Albert model has diameter approximately $\ln n$. As noted by Bollobás and Riordan [**?**], the approximation $\mathrm{diam}(G_m^n) \approx \ln n$ holds for the case $m = 1$, but for $m \geq 2$ they showed that as $n \to \infty$ then $\mathrm{diam}(G_m^n) \to \ln / \ln \ln n$.

10.6 Problems

> Where should I start? Start from the statement of the problem. What can I do? Visualize the problem as a whole as clearly and as vividly as you can.
> — G. Polya, from page 33 of [**?**]

10.1. Algorithm 10.8 presents a procedure to construct a random graph that is simple and undirected; the procedure is adapted from pages 4–7 of Lau [**?**]. Analyze the time complexity of Algorithm 10.8. Compare and contrast your results with that for Algorithm 10.5.

10.2. Modify Algorithm 10.8 to generate the following random graphs.

(a) Simple weighted, undirected graph.

(b) Simple digraph.

(c) Simple weighted digraph.

10.3. Algorithm 10.1 can be considered as a template for generating random graphs in $\mathcal{G}(n, p)$. The procedure does not specify how to generate all the 2-combinations of a set of $n > 1$ objects. Here we discuss how to construct all such 2-combinations and derive a quadratic time algorithm for generating random graphs in $\mathcal{G}(n, p)$.

(a) Consider a vertex set $V = \{0, 1, \ldots, n-1\}$ with at least two elements and let E be the set of all 2-combinations of V, where each 2-combination is written ij. Show that $ij \in E$ if and only if $i < j$.

(b) From the previous exercise, we know that if $0 \leq i < n-1$ then there are $n - (i+1)$ pairs jk where either $i = j$ or $i = k$. Show that

$$\sum_{i=0}^{n-2}(n-i-1) = \frac{n^2-n}{2}$$

and conclude that Algorithm 10.9 has worst-case runtime $O((n^2-n)/2)$.

10.4. Modify the Batagelj-Brandes Algorithm 10.3 to generate the following types of graphs.

(a) Directed simple graphs.

Algorithm 10.8: Random simple undirected graph.

Input: Positive integers n and m specifying the order and size, respectively, of the output graph.

Output: A random simple undirected graph with n vertices and m edges. If m exceeds the size of K_n, then K_n is returned.

```
1  if n = 1 then
2      return K_1
3  max ← n(n − 1)/2
4  if m > max then
5      return K_n
6  G ← null graph
7  A ← n × n adjacency matrix with entries a_ij
8  a_ij ← False for 0 ≤ i, j < n
9  i ← 0
10 while i < m do
11     u ← draw uniformly at random from {0, 1, ..., n − 1}
12     v ← draw uniformly at random from {0, 1, ..., n − 1}
13     if u = v then
14         continue with next iteration of loop
15     if u > v then
16         swap values of u and v
17     if a_uv = False then
18         add edge uv to G
19         a_uv ← True
20         i ← i + 1
21 return G
```

Algorithm 10.9: Quadratic generation of a random graph in $\mathcal{G}(n, p)$.

Input: Positive integer n and a probability $0 < p < 1$.

Output: A random graph from $G(n, p)$.

```
1 G ← complement of K_n
2 V ← {0, 1, ..., n − 1}
3 for i ← 0, 1, ..., n − 2 do
4     for j ← i + 1, i + 2, ..., n − 1 do
5         r ← draw uniformly at random from interval (0, 1)
6         if r < p then
7             add edge ij to G
8 return G
```

Algorithm 10.10: Briggs' algorithm for random graph in $\mathcal{G}(n, N)$.

Input: Positive integers n and N such that $1 \leq N \leq \binom{n}{2}$.
Output: A random graph from $G(n, N)$.

```
1  max ← (n choose 2)
2  if n = 1 or N = max then
3      return K_n
4  G ← complement(K_n)
5  u ← 0
6  v ← 1
7  t ← 0              /* number of candidates processed so far */
8  k ← 0                 /* number of edges selected so far */
9  while True do
10     r ← draw uniformly at random from {0, 1, ..., max − t}
11     if r < N − k then
12         add edge uv to G
13         k ← k + 1
14         if k = N then
15             return G
16     t ← t + 1
17     v ← v + 1
18     if v = n then
19         u ← u + 1
20         v ← u + 1
```

(b) Directed acyclic graphs.

(c) Bipartite graphs.

10.5. Repeat the previous problem for Algorithm 10.5.

10.6. In 2006, Keith M. Briggs provided [**?**] an algorithm that generates a random graph in $\mathcal{G}(n, N)$, inspired by Knuth's Algorithm S (Selection sampling technique) as found on page 142 of Knuth [**?**]. Pseudocode of Briggs' procedure is presented in Algorithm 10.10. Provide runtime analysis of Algorithm 10.10 and compare your results with those presented in section 10.3. Under which conditions would Briggs' algorithm be more efficient than Algorithm 10.5?

10.7. Briggs' Algorithm 10.10 follows the general template of an algorithm that samples without replacement n items from a pool of N candidates. Here $0 < n \leq N$ and the size N of the candidate pool is known in advance. However there are situations where the value of N is not known beforehand, and we wish to sample without replacement n items from the candidate pool. What we know is that the candidate pool has enough members to allow us to select n items. Vitter's algorithm R [**?**], called reservoir sampling, is suitable for the situation and runs in $O(n(1+\ln(N/n)))$ expected time. Describe and provide pseudocode of Vitter's algorithm, prove its correctness, and provide runtime analysis.

10.8. Repeat Example 10.1 but using each of Algorithms 10.1 and 10.5.

10.9. Diego Garlaschelli introduced [**?**] in 2009 a weighted version of the $\mathcal{G}(n, p)$ model, called the weighted random graph model. Denote by $\mathcal{G}_W(n, p)$ the weighted random graph model. Provide a description and pseudocode of a procedure to generate a graph in $\mathcal{G}_W(n, p)$ and analyze the runtime complexity of the algorithm. Describe various statistical physics properties of $\mathcal{G}_W(n, p)$.

10.10. Latora and Marchiori [**?**] extended the Watts-Strogatz model to take into account weighted edges. A crucial idea in the Latora-Marchiori model is the concept of network efficiency. Describe the Latora-Marchiori model and provide pseudocode of an algorithm to construct Latora-Marchiori networks. Explain the concepts of local and global efficiencies and how these relate to clustering coefficient and characteristic path length. Compare and contrast the Watts-Strogatz and Latora-Marchiori models.

10.11. The following model for "growing" graphs is known as the CHKNS model [**?**],[1] named for its original proponents. Start with the trivial graph G at time step $t = 1$. For each subsequent time step $t > 1$, add a new vertex to G. Furthermore choose two vertices uniformly at random and with probability δ join them by an undirected edge. The newly added edge does not necessarily have the newly added vertex as an endpoint. Denote by $d_k(t)$ the expected number of vertices with degree k at time t. Assuming that no self-loops are allowed, show that

$$d_0(t+1) = d_0(t) + 1 - 2\delta\frac{d_0(t)}{t}$$

[1] Or the "chickens" model, depending on how you pronounce "CHKNS".

and

$$d_k(t+1) = d_k(t) + 2\delta\frac{d_{k-1}(t)}{t} - 2\delta\frac{d_k(t)}{t}.$$

As $t \to \infty$, show that the probability that a vertex be chosen twice decreases as t^{-2}. If v is a vertex chosen uniformly at random, show that

$$\Pr[\deg(v) = k] = \frac{(2\delta)^k}{(1+2\delta)^{k+1}}$$

and conclude that the CHKNS model has an exponential degree distribution. The *size* of a component counts the number of vertices in the component itself. Let $N_k(t)$ be the expected number of components of size k at time t. Show that

$$N_1(t+1) = N_1(t) + 1 - 2\delta\frac{N_1(t)}{t}$$

and for $k > 1$ show that

$$N_k(t+1) = N_k(t) + \delta\left(\sum_{i=1}^{k-1}\frac{iN_i(t)}{t}\cdot\frac{(k-i)N_{k-i}(t)}{t}\right) - 2\delta\frac{kN_k(t)}{t}.$$

10.12. Algorithm 10.7 can easily be modified to generate other types of scale-free networks. Based upon the latter algorithm, Batagelj and Brandes [**?**] presented a procedure for generating bipartite scale-free networks; see Algorithm 10.11 for pseudocode. Analyze the runtime efficiency of Algorithm 10.11. Fix positive integer values for n and d, say $n = 10,000$ and $d = 4$. Use Algorithm 10.11 to generate a bipartite graph with your chosen values for n and d. Plot the degree distribution of the resulting graph using a log-log scale and confirm that the generated graph is scale-free.

10.13. Find the degree and distance distributions, average path lengths, and clustering coefficients of the following network datasets:

(a) actor collaboration [**?**]
(b) co-authorship of condensed matter preprints [**?**]
(c) Google web graph [**?**]
(d) LiveJournal friendship [**?**, **?**]
(e) neural network of the C. elegans [**?**, **?**]
(f) US patent citation [**?**]
(g) Western States Power Grid of the US [**?**]
(h) Zachary karate club [**?**]

10.14. Consider the plots of degree distributions in Figures 10.12 and 10.13. Note the noise in the tail of each plot. To smooth the tail, we can use the cumulative degree distribution

$$P^c(k) = \sum_{i=k}^{\infty}\Pr[\deg(v) = i].$$

Given a graph with scale-free degree distribution $P(k) \sim k^{-\alpha}$ and $\alpha > 1$, the cumulative degree distribution follows $P^c(k) \sim k^{1-\alpha}$. Plot the cumulative degree distribution of each network dataset in Problem 10.13.

Algorithm 10.11: Bipartite scale-free network via preferential attachment.

Input: Positive integer $n > 1$ and minimum degree $d \geq 1$.

Output: Bipartite scale-free multigraph. Each partition has n vertices and each vertex has minimum degree d.

```
1  G ← complement(K_{2n})                    /* vertex set is {0, 1, ..., 2n − 1} */
2  M_1 ← list of length 2nd
3  M_2 ← list of length 2nd
4  for v = 0, 1, ..., n − 1 do
5      for i = 0, 1, ..., d − 1 do
6          M_1[2(vd + i)] ← v
7          M_2[2(vd + i)] ← n + v
8          r ← draw uniformly at random from {0, 1, ..., 2(vd + i)}
9          if r is even then
10             M_1[2(vd + i) + 1] ← M_2[r]
11         else
12             M_1[2(vd + i) + 1] ← M_1[r]
13         r ← draw uniformly at random from {0, 1, ..., 2(vd + i)}
14         if r is even then
15             M_2[2(vd + i) + 1] ← M_1[r]
16         else
17             M_2[2(vd + i) + 1] ← M_2[r]
18 add edges (M_1[2i], M_1[2i + 1]) and (M_2[2i], M_2[2i + 1]) to G for i = 0, 1, ..., nd − 1
19 return G
```

Chapter 11

Graph problems and their LP formulations

This document is meant as an explanation of several graph theoretical functions defined in Sage's Graph Library (http://www.sagemath.org/), which use Linear Programming to solve optimization of existence problems.

11.1 Maximum average degree

The average degree of a graph G is defined as $ad(G) = \frac{2|E(G)|}{|V(G)|}$. The maximum average degree of G is meant to represent its densest part, and is formally defined as :

$$mad(G) = \max_{H \subseteq G} ad(H)$$

Even though such a formulation does not show it, this quantity can be computed in polynomial time through Linear Programming. Indeed, we can think of this as a simple flow problem defined on a bipartite graph. Let D be a directed graph whose vertex set we first define as the disjoint union of $E(G)$ and $V(G)$. We add in D an edge between $(e, v) \in E(G) \times V(G)$ if and only if v is one of e's endpoints. Each edge will then have a flow of 2 (through the addition in D of a source and the necessary edges) to distribute among its two endpoints. We then write in our linear program the constraint that each vertex can absorb a flow of at most z (add to D the necessary sink and the edges with capacity z).

Clearly, if $H \subseteq G$ is the densest subgraph in G, its $|E(H)|$ edges will send a flow of $2|E(H)|$ to their $|V(H)|$ vertices, such a flow being feasible only if $z \geq \frac{2|E(H)|}{|V(H)|}$. An elementary application of the max-flow/min-cut theorem, or of Hall's bipartite matching theorem shows that such a value for z is also sufficient. This LP can thus let us compute the Maximum Average Degree of the graph.

Sage method : `Graph.maximum_average_degree()`

LP Formulation :

- Minimize : z

- Such that :

– a vertex can absorb at most z

$$\forall v \in V(G), \sum_{\substack{e \in E(G) \\ e \sim v}} x_{e,v} \leq z$$

– each edge sends a flow of 2

$$\forall e = uv \in E(G), x_{e,u} + x_{e,u} = 2$$

- $x_{e,v}$ real positive variable

Here is the corresponding Sage code:

```
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram(maximization = False)
sage: x = p.new_variable( dim = 2 )

sage: p.set_objective(p['z'])

sage: for v in g:
...       p.add_constraint( sum([ x[u][v] for u in g.neighbors(v) ]) <= p['z'] )

sage: for u,v in g.edges(labels = False):
...       p.add_constraint( x[u][v] + x[v][u] == 2 )

sage: p.solve()
3.0
```

REMARK : In many if not all the other LP formulations, this Linear Program is used as a constraint. In those problems, we are always at some point looking for a subgraph H of G such that H does not contain any cycle. The edges of G are in this case variables, whose value can be equal to 0 or 1 depending on whether they belong to such a graph H. Based on the observation that the Maximum Average Degree of a tree on n vertices is exactly its average degree ($= 2 - 2/n < 1$), and that any cycles in a graph ensures its average degree is larger than 2, we can then set the constraint that $z \leq 2 - \frac{2}{|V(G)|}$. This is a handy way to write in LP the constraint that "the set of edges belonging to H is acyclic". For this to work, though, we need to ensure that the variables corresponding to our edges are binary variables.

11.2 Traveling Salesman Problem

Given a graph G whose edges are weighted by a function $w : E(G) \to \mathbf{R}$, a solution to the TSP is a Hamiltonian (spanning) cycle whose weight (the sum of the weight of its edges) is minimal. It is easy to define both the objective and the constraint that each vertex must have exactly two neighbors, but this could produce solutions such that the set of edges define the disjoint union of several cycles. One way to formulate this linear program is hence to add the constraint that, given an arbitrary vertex v, the set S of edges in the solution must contain no cycle in $G - v$, which amounts to checking that the set of edges in S no adjacent to v is of maximal average degree strictly less than 2, using the remark from section **??**.

We will then, in this case, define variables representing the edges included in the solution, along with variables representing the weight that each of these edges will send to their endpoints.

LP Formulation :

- Minimize

$$\sum_{e \in E(G)} w(e) b_e$$

- Such that :
  - Each vertex is of degree 2

$$\forall v \in V(G), \sum_{\substack{e \in E(G) \\ e \sim v}} b_e = 2$$

  - No cycle disjoint from a special vertex v^*
    - Each edge sends a flow of 2 if it is taken

$$\forall e = uv \in E(G - v^*), x_{e,u} + x_{e,v} = 2b_e$$

    - Vertices receive strictly less than 2

$$\forall v \in V(G - v^*), \sum_{\substack{e \in E(G) \\ e \sim v}} x_{e,v} \leq 2 - \frac{2}{|V(G)|}$$

- Variables
  - $x_{e,v}$ real positive variable (flow sent by the edge)
  - b_e binary (is the edge in the solution ?)

Sage method : `Graph.traveling_salesman_problem()`

Here is the corresponding Sage corresponding to a simpler case – looking for an Hamiltonian cycle in a graph:

```
sage: g = graphs.GridGraph([4,4])
sage: p = MixedIntegerLinearProgram(maximization = False)

sage: f = p.new_variable()
sage: r = p.new_variable()

sage: eps = 1/(2*Integer(g.order()))
sage: x = g.vertex_iterator().next()

sage: # reorders the edge as they can appear in the two different ways
sage: R = lambda x,y : (x,y) if x < y else (y,x)

sage: # All the vertices have degree 2
sage: for v in g:
...     p.add_constraint( sum([ f[R(u,v)] for u in g.neighbors(v)]) == 2)

sage: # r is greater than f
sage: for u,v in g.edges(labels = None):
...     p.add_constraint( r[(u,v)] + r[(v,u)] - f[R(u,v)] >= 0)

sage: # no cycle which does not contain x
sage: for v in g:
...     if v != x:
...         p.add_constraint( sum([ r[(u,v)] for u in g.neighbors(v)]) <= 1-eps)

sage: p.set_objective(None)
sage: p.set_binary(f)

sage: p.solve()                                    # optional - GLPK,CBC,CPLEX
0.0
```

```
sage: # We can now build the solution
sage: # found as a graph

sage: f = p.get_values(f)                                  # optional - GLPK,CBC,CPLEX
sage: tsp = Graph()                                        # optional - GLPK,CBC,CPLEX
sage: for e in g.edges(labels = False):                    # optional - GLPK,CBC,CPLEX
...       if f[R(e[0],e[1])] == 1:                         # optional - GLPK,CBC,CPLEX
...           tsp.add_edge(e)                              # optional - GLPK,CBC,CPLEX

sage: tsp.is_regular(k=2) and tsp.is_connected()           # optional - GLPK,CBC,CPLEX
True
sage: tsp.order() == g.order()                             # optional - GLPK,CBC,CPLEX
True
```

11.3 Edge-disjoint spanning trees

This problem is polynomial by a result from Edmonds. Obviously, nothing ensures the following formulation is a polynomial algorithm as it contains many integer variables, but it is still a short practical way to solve it.

This problem amounts to finding, given a graph G and an integer k, edge-disjoint spanning trees $T_1, \ldots, T_k$ which are subgraphs of G. In this case, we will chose to define a spanning tree as an acyclic set of $|V(G)| - 1$ edges.

Sage method : `Graph.edge_disjoint_spanning_trees()`

LP Formulation :

- Maximize : nothing
- Such that :
  - An edge belongs to at most one set

$$\forall e \in E(G), \sum_{i \in [1,\ldots,k]} b_{e,k} \leq 1$$

  - Each set contains $|V(G)| - 1$ edges

$$\forall i \in [1, \ldots, k], \sum_{e \in E(G)} b_{e,k} = |V(G)| - 1$$

  - No cycles
    - In each set, each edge sends a flow of 2 if it is taken

$$\forall i \in [1, \ldots, k], \forall e = uv \in E(G), x_{e,k,u} + x_{e,k,u} = 2b_{e,k}$$

    - Vertices receive strictly less than 2

$$\forall i \in [1, \ldots, k], \forall v \in V(G), \sum_{\substack{e \in E(G) \\ e \sim v}} x_{e,k,v} \leq 2 - \frac{2}{|V(G)|}$$

- Variables
  - $b_{e,k}$ binary (is edge e in set k ?)
  - $x_{e,k,u}$ positive real (flow sent by edge e to vertex u in set k)

Here is the corresponding Sage code:

```
sage: g = graphs.RandomGNP(40,.6)
sage: p = MixedIntegerLinearProgram()
sage: colors = range(2)

sage: # Sort an edge
sage: S = lambda (x,y) : (x,y) if x<y else (y,x)

sage: edges = p.new_variable(dim = 2)
sage: r_edges = p.new_variable(dim = 2)

sage: # An edge belongs to at most one tree
sage: for e in g.edges(labels=False):
...       p.add_constraint(sum([edges[j][S(e)] for j in colors]), max = 1)

sage: for j in colors:
...       # each color class has g.order()-1 edges
...       p.add_constraint(
...         sum([edges[j][S(e)] for e in g.edges(labels=None)])
...         >= g.order()-1)
...       # Each vertex is in the tree
...       for v in g.vertices():
...            p.add_constraint(
...              sum([edges[j][S(e)] for e in g.edges_incident(v, labels=None)])
...              >= 1)
...       # r_edges is larger than edges
...       for u,v in g.edges(labels=None):
...            p.add_constraint(
...               r_edges[j][(u,v)] + r_edges[j][(v, u)]
...               == edges[j][S((u,v))] )

sage: # no cycles
sage: epsilon =  (3*Integer(g.order()))**(-1)
sage: for j in colors:
...      for v in g:
...          p.add_constraint(
...             sum([r_edges[j][(u,v)] for u in g.neighbors(v)])
...             <= 1-epsilon)

sage: p.set_binary(edges)
sage: p.set_objective(None)
sage: p.solve()                                    # optional - GLPK,CBC,CPLEX
0.0


sage: # We can now build the solution
sage: # found as a list of trees

sage: edges = p.get_values(edges)                  # optional - GLPK,CBC,CPLEX
sage: trees = [Graph() for c in colors]            # optional - GLPK,CBC,CPLEX

sage: for e in g.edges(labels = False):            # optional - GLPK,CBC,CPLEX
...       for c in colors:                         # optional - GLPK,CBC,CPLEX
...          if round(edges[c][S(e)]) == 1:        # optional - GLPK,CBC,CPLEX
...              trees[c].add_edge(e)              # optional - GLPK,CBC,CPLEX

sage: all([ trees[j].is_tree() for j in colors ])  # optional - GLPK,CBC,CPLEX
True
```

11.4 Steiner tree

See Trietsch [**?**] for a relationship between Steiner trees and Euler's problem of polygon division. Finding a spanning tree in a Graph G can be done in linear time, whereas computing a Steiner Tree is NP-hard. The goal is in this case, given a graph, a weight function $w : E(G) \to \mathbf{R}$ and a set S of vertices, to find the tree of minimum cost connecting them all together. Equivalently, we will be looking for an acyclic subgraph $H$of G containing $|V(H)|$ vertices and $|E(H)| = |V(H)| - 1$ edges, which contains each vertex from S

LP Formulation :

- Minimize :

$$\sum_{e \in E(G)} w(e) b_e$$

- Such that :
  - Each vertex from S is in the tree

$$\forall v \in S, \sum_{\substack{e \in E(G) \\ e \sim v}} b_e \geq 1$$

  - c is equal to 1 when a vertex v is in the tree

$$\forall v \in V(G), \forall e \in E(G), e \sim v, b_e \leq c_v$$

  - The tree contains $|V(H)|$ vertices and $|E(H)| = |V(H)| - 1$ edges

$$\sum_{v \in G} c_v - \sum_{e \in E(G)} b_e = 1$$

  - No Cycles
    - Each edge sends a flow of 2 if it is taken

$$\forall e = uv \in E(G), x_{e,u} + x_{e,u} = 2b_{e,k}$$

    - Vertices receive strictly less than 2

$$\forall v \in V(G), \sum_{\substack{e \in E(G) \\ e \sim v}} x_{e,v} \leq 2 - \frac{2}{|V(G)|}$$

- Variables :
  - b_e binary (is e in the tree ?)
  - c_v binary (does the tree contain v ?)
  - $x_{e,v}$ real positive variable (flow sent by the edge)

Sage method : `Graph.steiner_tree()`

Here is the corresponding Sage code:

```
sage: g = graphs.GridGraph([10,10])
sage: vertices = [(0,2),(5,3)]

sage: from sage.numerical.mip import MixedIntegerLinearProgram
sage: p = MixedIntegerLinearProgram(maximization = False)

sage: # Reorder an edge
sage: R = lambda (x,y) : (x,y) if x<y else (y,x)

sage: # edges used in the Steiner Tree
sage: edges = p.new_variable()
sage: # relaxed edges to test for acyclicity
sage: r_edges = p.new_variable()
```

```
sage: # Whether a vertex is in the Steiner Tree
sage: vertex = p.new_variable()

sage: #  Which vertices are in the tree ?
sage: for v in g:
...           for e in g.edges_incident(v, labels=False):
...               p.add_constraint(vertex[v] - edges[R(e)], min = 0)

sage: # We must have the given vertices in our tree
sage: for v in vertices:
...           p.add_constraint(
...             sum([edges[R(e)] for e in g.edges_incident(v,labels=False)]
...              == 1)

sage: # The number of edges is equal to the number of vertices in our tree minus 1
sage: p.add_constraint(
...     sum([vertex[v] for v in g])
...     - sum([edges[R(e)] for e in g.edges(labels=None)])
...     == 1)

sage: # There are no cycles in our graph
sage: for u,v in g.edges(labels = False):
...           p.add_constraint(
...             r_edges[(u,v)]+ r_edges[(v,u)] - edges[R((u,v))]
...             <= 0 )

sage: eps = 1/(5*Integer(g.order()))

sage: for v in g:
...           p.add_constraint(sum([r_edges[(u,v)] for u in g.neighbors(v)]), max = 1-eps)

sage: p.set_objective(sum([edges[R(e)] for e in g.edges(labels = False)]))
sage: p.set_binary(edges)
sage: p.solve()                                          # optional - GLPK,CBC,CPLEX
6.0

sage: # We can now build the solution
sage: # found as a tree

sage: edges = p.get_values(edges)                        # optional - GLPK,CBC,CPLEX
sage: st = Graph()                                       # optional - GLPK,CBC,CPLEX
sage: st.add_edges(
...     [e for e in g.edges(labels = False)
...     if edges[R(e)] == 1])                            # optional - GLPK,CBC,CPLEX
sage: st.is_tree()                                       # optional - GLPK,CBC,CPLEX
True
sage: all([v in st for v in vertices])                   # optional - GLPK,CBC,CPLEX
True
```

11.5 Linear arboricity

The linear arboricity of a graph G is the least number k such that the edges of G can be partitioned into k classes, each of them being a forest of paths (the disjoints union of paths – trees of maximal degree 2). The corresponding LP is very similar to the one giving edge-disjoint spanning trees

LP Formulation :

- Maximize : nothing
- Such that :
  - An edge belongs to exactly one set

$$\forall e \in E(G), \sum_{i \in [1,\dots,k]} b_{e,k} = 1$$

- Each class has maximal degree 2

$$\forall v \in V(G), \forall i \in [1, \dots, k], \sum_{\substack{e \in E(G) \\ e \sim v}} b_{e,k} \leq 2$$

- No cycles
  - In each set, each edge sends a flow of 2 if it is taken

$$\forall i \in [1, \dots, k], \forall e = uv \in E(G), x_{e,k,u} + x_{e,k,v} = 2b_{e,k}$$

  - Vertices receive strictly less than 2

$$\forall i \in [1, \dots, k], \forall v \in V(G), \sum_{\substack{e \in E(G) \\ e \sim v}} x_{e,k,v} \leq 2 - \frac{2}{|V(G)|}$$

- Variables
  - $b_{e,k}$ binary (is edge e in set k ?)
  - $x_{e,k,u}$ positive real (flow sent by edge e to vertex u in set k)

Sage method : `sage.graphs.graph_coloring.linear_arboricity()`
Here is the corresponding Sage code :

```
sage: g = graphs.GridGraph([4,4])
sage: k = 2
sage: p = MixedIntegerLinearProgram()

sage: # c is a boolean value such that c[i][(u,v)] = 1
sage: # if and only if (u,v) is colored with i
sage: c = p.new_variable(dim=2)

sage: # relaxed value
sage: r = p.new_variable(dim=2)

sage: E = lambda x,y : (x,y) if x<y else (y,x)

sage: MAD = 1-1/(Integer(g.order())*2)

sage: # Partition of the edges
sage: for u,v in g.edges(labels=None):
...     p.add_constraint(sum([c[i][E(u,v)] for i in range(k)]), max=1, min=1)

sage: for i in range(k):
...     # r greater than c
...     for u,v in g.edges(labels=None):
...         p.add_constraint(r[i][(u,v)] + r[i][(v,u)] - c[i][E(u,v)], max=0, min=0)
...         # Maximum degree 2
...         for u in g.vertices():
...             p.add_constraint(sum([c[i][E(u,v)] for v in g.neighbors(u)]),max = 2)
...             # no cycles
...             p.add_constraint(sum([r[i][(u,v)] for v in g.neighbors(u)]),max = MAD)

sage: p.set_objective(None)
sage: p.set_binary(c)

sage: c = p.get_values(c)

sage: gg = g.copy()
sage: gg.delete_edges(g.edges())
sage: answer = [gg.copy() for i in range(k)]
sage: add = lambda (u,v),i : answer[i].add_edge((u,v))

sage: for i in range(k):
...         for u,v in g.edges(labels=None):
...             if c[i][E(u,v)] == 1:
...                 add((u,v),i)
```

11.6 H-minor

For more information on minor theory, please see
http://en.wikipedia.org/wiki/Minor_%28graph_theory%29
It is a wonderful subject, and I do not want to begin talking about it when I know I couldn't freely fill pages with remarks :-)

For our purposes, we will just say that finding a minor H in a graph G, consists in :

1. Associating to each vertex $h \in H$ a set S_h of representants in H, different vertices h having disjoints representative sets
2. Ensuring that each of these sets is connected (can be contracted)
3. If there is an edge between h_1 and h_2 in H, there must be an edge between the corresponding representative sets

Here is how we will address these constraints :

1. Easy
2. For any h, we can find a spanning tree in S_h (an acyclic set of $|S_h| - 1$ edges)
3. This one is very costly.

   To each *directed* edge g_1g_2 (I consider g_1g_2 and g_2g_1 as different) and every edge h_1h_2 is associated a binary variable which can be equal to one only if g_1 represents h_1 and g_2 represents g_2. We then sum all these variables to be sure there is at least one edge from one set to the other.

LP Formulation :

- Maximize : nothing
- Such that :
  - A vertex $g \in V(G)$ can represent at most one vertex $h \in V(H)$

$$\forall g \in V(G), \sum_{h \in V(H)} rs_{h,g} \leq 1$$

  - An edge e can only belong to the tree of h if both its endpoints represent h

$$\forall e = g_1g_2 \in E(G), t_{e,h} \leq rs_{h,g_1} \text{ and } t_{e,h} \leq rs_{h,g_2}$$

  - In each representative set, the number of vertices is one more than the number of edges in the corresponding tree

$$\forall h, \sum_{g \in V(G)} rs_{h,g} - \sum_{e \in E(G)} t_{e,h} = 1$$

  - No cycles in the union of the spanning trees
    - Each edge sends a flow of 2 if it is taken

$$\forall e = uv \in E(G), x_{e,u} + x_{e,v} = 2 \sum_{h \in V(H)} t_{e,h}$$

    - Vertices receive strictly less than 2

$$\forall v \in V(G), \sum_{\substack{e \in E(G) \\ e \sim v}} x_{e,k,v} \leq 2 - \frac{2}{|V(G)|}$$

  - $arc_{(g_1,g_2),(h_1,h_2)}$ can only be equal to 1 if g_1g_2 is leaving the representative set of h_1 to enter the one of h_2. (note that this constraints has to be written both for g_1, g_2, and then for g_2, g_1)

$$\forall g_1, g_2 \in V(G), g_1 \neq g_2, \forall h_1h_2 \in E(H)$$

$$arc_{(g_1,g_2),(h_1,h_2)} \leq rs_{h_1,g_1} \text{ and } arc_{(g_1,g_2),(h_1,h_2)} \leq rs_{h_2,g_2}$$

  - We have the necessary edges between the representative sets

$$\forall h_1h_2 \in E(H)$$

$$\sum_{\forall g_1,g_2 \in V(G), g_1 \neq g_2} arc_{(g_1,g_2),(h_1,h_2)} \geq 1$$

- Variables
  - $rs_{h,g}$ binary (does g represent h ? rs = "representative set")
  - $t_{e,h}$ binary (does e belong to the spanning tree of the set representing h ?)
  - $x_{e,v}$ real positive (flow sent from edge e to vertex v)

– $arc_{(g_1,g_2),(h_1,h_2)}$ binary (is edge g_1g_2 leaving the representative set of h_1 to enter the one of h_2 ?)

Here is the corresponding Sage code:

```
sage: g = graphs.PetersenGraph()
sage: H = graphs.CompleteGraph(4)

sage: p = MixedIntegerLinearProgram()

sage: # sorts an edge
sage: S = lambda (x,y) : (x,y) if x<y else (y,x)

sage: # rs = Representative set of a vertex
sage: # for h in H, v in G is such that rs[h][v] == 1 if and only if v
sage: # is a representant of h in g
sage: rs = p.new_variable(dim=2)

sage: for v in g:
...           p.add_constraint(sum([rs[h][v] for h in H]), max = 1)

sage: # We ensure that the set of representatives of a
sage: # vertex h contains a tree, and thus is connected

sage: # edges represents the edges of the tree
sage: edges = p.new_variable(dim = 2)

sage: # there can be a edge for h between two vertices
sage: # only if those vertices represent h
sage: for u,v in g.edges(labels=None):
...           for h in H:
...               p.add_constraint(edges[h][S((u,v))] - rs[h][u], max = 0 )
...               p.add_constraint(edges[h][S((u,v))] - rs[h][v], max = 0 )

sage: # The number of edges of the tree in h is exactly the cardinal
sage: # of its representative set minus 1

sage: for h in H:
...           p.add_constraint(
...             sum([edges[h][S(e)] for e in g.edges(labels=None)])
...             -sum([rs[h][v] for v in g])
...             ==1 )

sage: # a tree  has no cycle
sage: epsilon = 1/(5*Integer(g.order()))
sage: r_edges = p.new_variable(dim=2)

sage: for h in H:
...           for u,v in g.edges(labels=None):
...               p.add_constraint(
...                r_edges[h][(u,v)] + r_edges[h][(v,u)] >= edges[h][S((u,v))])
...           for v in g:
...               p.add_constraint(
...                  sum([r_edges[h][(u,v)] for u in g.neighbors(v)]) <= 1-epsilon)

sage: # Once the representative sets are described, we must ensure
sage: # there are arcs corresponding to those of H between them
sage: h_edges = p.new_variable(dim=2)

sage: for h1, h2 in H.edges(labels=None):
...           for v1, v2 in g.edges(labels=None):
...               p.add_constraint(h_edges[(h1,h2)][S((v1,v2))] - rs[h2][v2], max = 0)
...               p.add_constraint(h_edges[(h1,h2)][S((v1,v2))] - rs[h1][v1], max = 0)
...               p.add_constraint(h_edges[(h2,h1)][S((v1,v2))] - rs[h1][v2], max = 0)
...               p.add_constraint(h_edges[(h2,h1)][S((v1,v2))] - rs[h2][v1], max = 0)



sage: p.set_binary(rs)
sage: p.set_binary(edges)
sage: p.set_objective(None)
sage: p.solve()                                          # optional - GLPK,CBC,CPLEX
0.0

sage: # We can now build the solution found as a
sage: # dictionary associating to each vertex of H
```

```
sage: # the corresponding set of vertices in G

sage: rs = p.get_values(rs)

sage: from sage.sets.set import Set
sage: rs_dict = {}
sage: for h in H:
...           rs_dict[h] = [v for v in g if rs[h][v]==1]
```

Appendix A

Asymptotic growth

Name	Standard notation	Intuitive notation	Meaning
theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
big oh	$f(n) = O(g(n))$	$f(n) \leq O(g(n))$	$f(n) \leq c \cdot g(n)$
omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$
little oh	$f(n) = o(g(n))$	$f(n) \ll o(g(n))$	$f(n) \ll g(n)$
little omega	$f(n) = \omega(g(n))$	$f(n) \gg \omega(g(n))$	$f(n) \gg g(n)$
tilde	$f(n) = \tilde{\Theta}(g(n))$	$f(n) \in \tilde{\Theta}(g(n))$	$f(n) \approx \log^{\Theta(1)} g(n)$

Table A.1: Meaning of asymptotic notations.

Class	$\lim_{n\to\infty} f(n)/g(n) =$	Equivalent definition
$f(n) = \Theta(g(n))$	a constant	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
$f(n) = o(g(n))$	zero	$f(n) = O(g(n))$ but $f(n) \neq \Omega(g(n))$
$f(n) = \omega(g(n))$	∞	$f(n) \neq O(g(n))$ but $f(n) = \Omega(g(n))$

Table A.2: Asymptotic behavior in the limit of large n.

Appendix B

GNU Free Documentation License

Version 1.3, 3 November 2008
Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

http://www.fsf.org

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ

in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment

to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.