Algorithmic Graph Theory

David Joyner, Minh Van Nguyen, Nathann Cohen

Version 0.3

David Joyner <wdjoyner@gmail.com>
Minh Van Nguyen <nguyenminh2@gmail.com>
Nathann Cohen <nathann.cohen@gmail.com>

Edition
Version 0.3
19 March 2010

Contents

Acknowledgements

- Fidel Barrera-Cruz: reported typos in Chapter 3. See changeset 101.
- Daniel Black: reported a typo in Chapter 1. See changeset 61.

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction to Graph Theory

To paraphrase what Felix Klein said about curves,[1] it is easy to define a graph until you realize the countless number of exceptions. There are directed graphs, weighted graphs, multigraphs, simple graphs, and so on. Where do we begin?

1.1 Graphs and digraphs

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

Definition 1.1. ***Graphs.*** *A* graph $G = (V, E)$ *is an ordered pair of sets. Elements of* V *are called* vertices *or* nodes*, and elements of* $E \subseteq V \times V$ *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*.*

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.

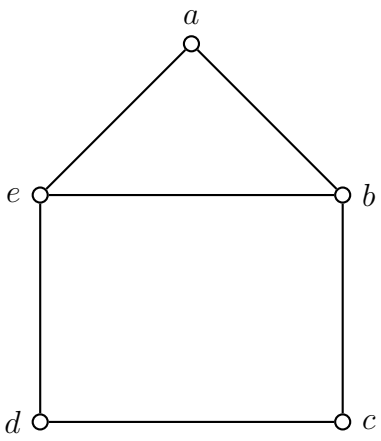


Figure 1.1: A house graph.

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

Example 1.2. *Consider the graph in Figure 1.1.*

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 1.1. 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 neighbours. We can use the function `G.neighbors()` to list all the neighbours 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 $V \times V$. 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 1.1 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 an *empty* or *null* graph. □

Example 1.3. *Consider the illustration in Figure 1.2. Does Figure 1.2 represent a graph? Why or why not?*

Solution. If $V = \{a, b, c\}$ and $E = \{aa, bc\}$, it is clear that $E \subseteq V \times V$. 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. □

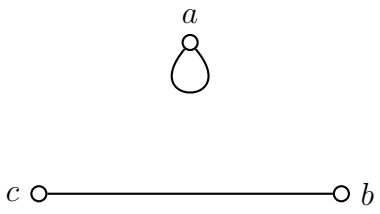


Figure 1.2: A figure with a self-loop.

In Figure 1.1, 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 1.1, 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.

Definition 1.4. ***Directed graphs.*** *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. 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 such that each of whose edges is directed. The* indegree *of a vertex $v \in V(G)$ counts the number of edges such that v is the head of those edges. The* outdegree *of a vertex $v \in V(G)$ is the number of edges such that v is the tail of those 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. Another important class of graphs consist of those graphs having multiple edges between pairs of vertices.

Definition 1.5. ***Multigraphs.*** *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.*

Definition 1.6. ***Simple graphs.*** *A* simple graph *is a graph with no self-loops and no multiple edges.*

The edges of a digraph can be visually represented as directed arrows, similarly to the digraph in Figure 1.3. The graph in Figure 1.3 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 1.3.

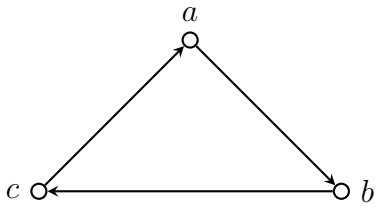


Figure 1.3: A triangle as a directed graph.

For any vertex v in a graph $G = (V, E)$, the cardinality of $\text{adj}(v)$ is called the *degree* of v and written as $\deg(v) = |\text{adj}(v)|$. The degree of v counts the number of vertices in G that are adjacent to v. If $\deg(v) = 0$, we say that v is an *isolated* vertex. For example, in the graph in Figure 1.1, we have $\deg(b) = 3$. For the graph in Figure 1.3, 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 1.1 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 1.1 then we have $\delta(G) = 2$ and $\Delta(G) = 3$. In the following Sage session, we construct the digraph in Figure 1.3 and computes 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 1.3, we have $\delta(G) = \Delta(G) = 2$.

The graph G in Figure 1.3 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 1.7. ***Euler.*** *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 1.7 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, B shaking A's hand, since the sum in the theorem is over all vertices).

As $E \subseteq V \times V$, 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 1.7, the total degree of G is non-negative and even. This result is an immediate consequence of Theorem 1.7 and is captured in the following corollary.

Corollary 1.8. *If G is a graph, then its total number of degrees is non-negative 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 1.7 we have $2m = rn$ and therefore $m = rn/2$. This result is captured in the following corollary.

Corollary 1.9. *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

1.2.1 Walks, trails, and paths

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. For the graph in Figure 1.4, 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.

A *trail* is a walk with no repeating edges. For example, the a-b walk a, b, c, d, f, g, b in Figure 1.4 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 is called a *path*. Without any repeating vertices, a path cannot have repeating edges, hence a path is also a trail.

A path whose start and end vertices are the same is called a *cycle*. For example, the walk a, b, c, e, a in Figure 1.4 is a path and a cycle. A walk which has no repeated edges and the start and end vertices are the same, but otherwise has no repeated vertices, is a *closed path* (with apologies for slightly abusing terminology).[2] Thus the walk a, b, e, a in Figure 1.4 is a closed path. It is easy to see that if you remove any edge from a cycle,

[2] A closed path 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.

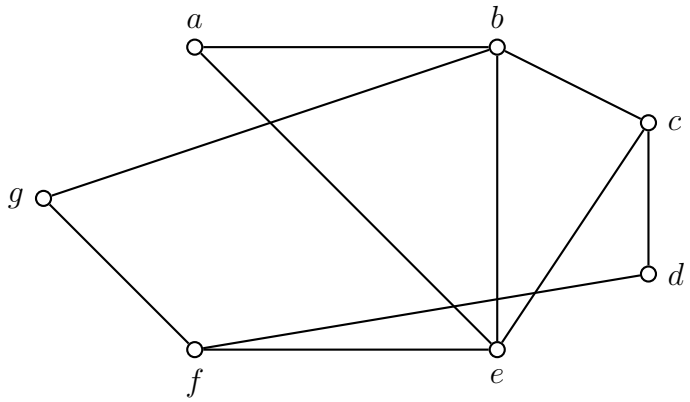


Figure 1.4: Walking along a graph.

then the resulting walk contains no closed paths. An *Euler subgraph* of a graph G is either a cycle or an edge-disjoint union of cycles in G.

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

Example 1.10. *Consider the graph in Figure 1.4.*

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 path that is not a cycle.*

5. *Find a closed path 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: $t_1 : a, b, c, d, f$ and $t_2 : b, e, f, d, c$. The length of trail t_1 is 4 and the length of trail t_2 is also 4.

(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 path that is not a cycle: d, c, e, b, a, e, f, d. □

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 non-empty graph with an isolated vertex. However, the graph in Figure 1.3 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 non-empty graph may have several shortest paths between some distinct pair of vertices. For the graph in Figure 1.4, both a, b, c and a, e, c are geodesic paths between a and c. The number of connected components of a graph G will be denoted $\omega(G)$.

Example 1.11. *Determine whether or not the graph in Figure 1.4 is connected. Find a shortest path from g to d.*

Solution. In the following Sage session, we first construct the graph in Figure 1.4 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']
```

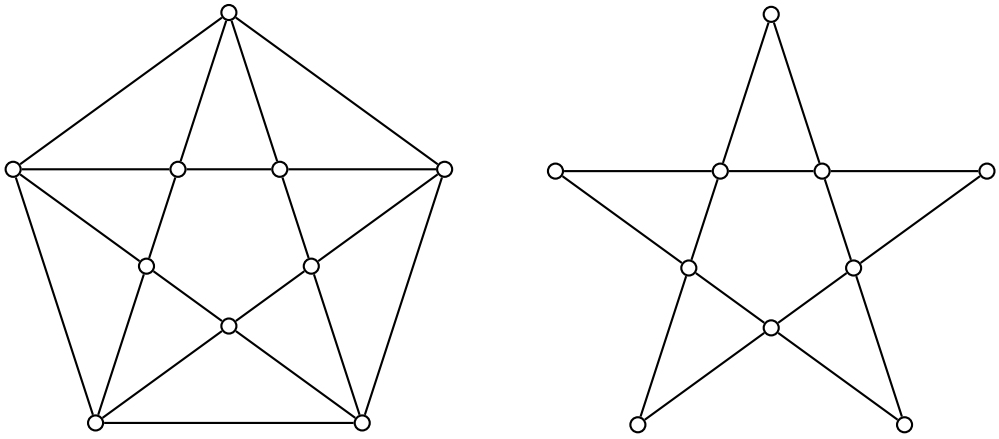
This shows that g, f, d is a shortest path from g to d. In fact, any other g-d path has length greater than 2, so we can say that g, f, d is the shortest path between g and d. □

We will explain Dijkstra's algorithm in Chapter 2, which 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!

1.2.2 Subgraphs, complete and bipartite graphs

Definition 1.12. *Let G be a graph 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 $uv \in E(H)$ then $u, v \in V(H)$. Then H is called a* subgraph *of G and 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 1.5, 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.



(a) (b)

Figure 1.5: A graph and one of its subgraphs.

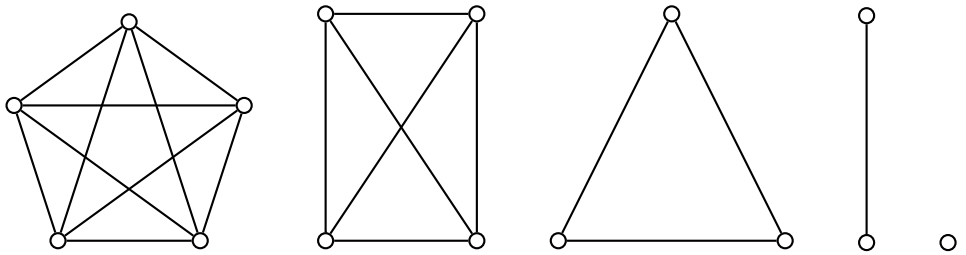
We now consider several standard classes of graphs. The *complete* graph K_n on n vertices is a graph such that any two distinct vertices are adjacent. As $|V(K_n)| = n$, then $|E(K_n)|$ is equivalent to the total number of 2-combinations from a set of n objects:

$$|E(K_n)| = \binom{n}{2} = \frac{n(n-1)}{2}. \tag{1.2}$$

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

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

Figure 1.6 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

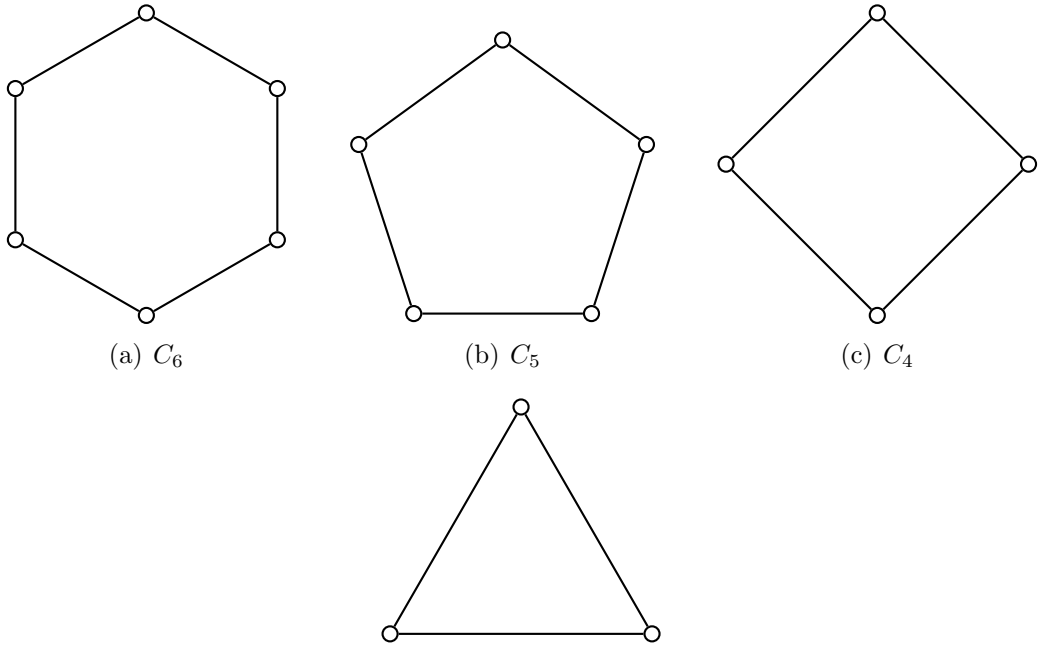
Figure 1.6: Complete graphs K_n for $1 \leq n \leq 5$.

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 1.7 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 non-empty. Every edge $uv \in E(G)$ is such that $u \in V_1$ and $v \in V_2$, or $v \in V_1$ and $u \in V_2$.

The *complete bipartite* graph $K_{m,n}$ is the bipartite graph whose vertex set is partitioned into two non-empty 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 1.8 shows a bipartite graph together with the complete bipartite graphs $K_{4,3}$ and $K_{3,3}$, and the star graph $K_{1,4}$.

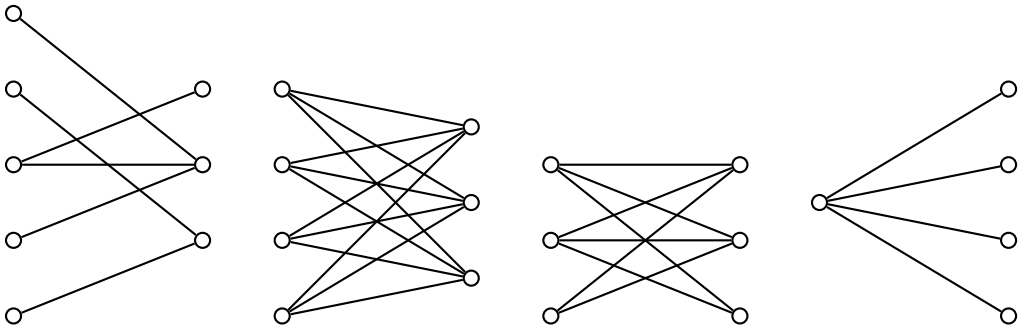
As an example of $K_{3,3}$, suppose that there are 3 boys and 3 girls dancing in a room. The boys and girls naturally partition the set of all people in the room. Construct a graph having 6 vertices, each vertex corresponding to a person in the room, and draw an edge form one vertex to another if the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



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

(d) C_3

Figure 1.7: Cycle graphs C_n for $3 \leq n \leq 6$.



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

Figure 1.8: Bipartite, complete bipartite, and star graphs.

1.3 Representing graphs using matrices

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 1.7. 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. This section requires some concepts and techniques from linear algebra, especially matrix theory. See introductory texts on linear algebra and matrix theory [4] 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

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

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 1.13. *Compute the adjacency matrices of the graphs in Figure 1.9.*

Solution. Define the graphs in Figure 1.9 using `DiGraph` and `Graph`. Then call the method `adjacency_matrix()`.

```
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]
```

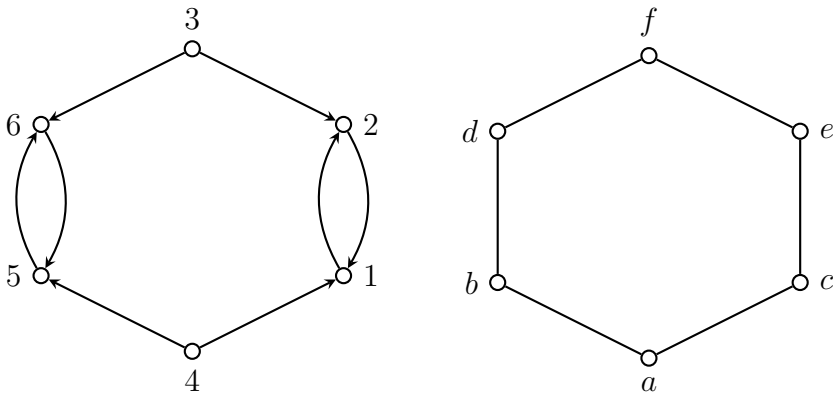


Figure 1.9: Adjacency matrices of directed and undirected graphs.

```
[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 and $V = V_1 \cup V_2$ is the partition of the vertices into n_1 vertices in V_1 and n_2 vertices in V_2, so $|V| = n_1 + n_2$. Then the adjacency matrix A of G can be realized as a block diagonal matrix $A = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}$,

where A_1 is an $n_1 \times n_2$ matrix and A_2 is an $n_2 \times n_1$ matrix. Since G is undirected, $A_2 = A_1^T$. The matrix 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").

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 1.10.

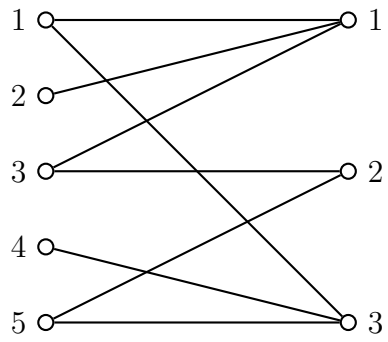


Figure 1.10: Tanner graph for H.

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

$$b_{ij} = \begin{cases} -1, & \text{if } v_i \text{ is the tail of } e_j, \\ 1, & \text{if } v_i \text{ is the head of } e_j, \\ 2, & \text{if } e_j \text{ is a self-loop at } v_i, \\ 0, & \text{otherwise.} \end{cases} \tag{1.3}$$

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 (1.3) 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. The *oriented incidence matrix* 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 B, 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, $b_{ij} = 2$ if e_j is a self-loop at v_i.

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 4.

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 4 and 9.

Problems 1.3

1. 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.

2. 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.

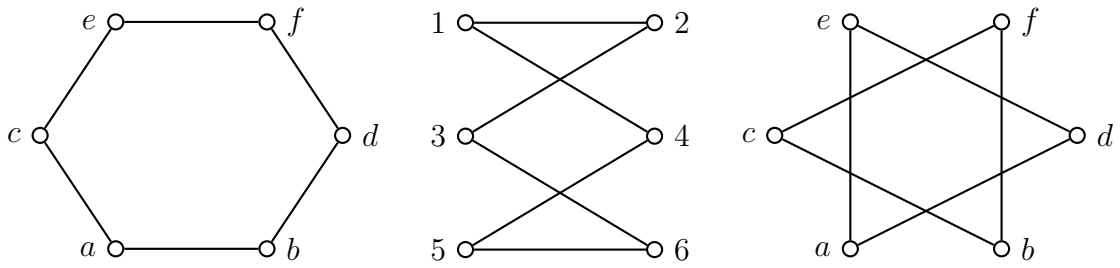
3. Let G be a digraph and let M be its incidence matrix. For any row r of M, let m be the frequency of -1 in r, let p be the frequency of 1 in r, and let t be twice the frequency of 2 in r. If v is the vertex corresponding to r, show that the degree of v is $\deg(v) = m + p + t$.

4. 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.

1.4 Isomorphic graphs

Determining whether or not two graphs are, in some sense, the "same" is a hard but important problem.

Definition 1.14. ***Isomorphic graphs.*** *Two graphs G and H are* isomorphic *if there is a bijection $f : V(G) \longrightarrow V(H)$ 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) C_6 (b) G_1 (c) G_2

Figure 1.11: Isomorphic and non-isomorphic 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 $f : V(G) \longrightarrow V(H)$ 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 $f(v_1), f(v_k) \in V(H)$.

Example 1.15. *Consider the graphs in Figure 1.11. 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 1.11, 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 1.16. *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,

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

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 Compute the adjacency matrix A_i of G_i (i = 1, 2).
2 Compute the lexicographically maximal element A'_i of the permutation
  equivalence class of A_i, for i = 1, 2.
3 if A'_1 = A'_2 then
4 |   return True
5 else
6 |   return False
7 end
```

Algorithm 1.1: Computing graph isomorphism using canonical labels.

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.

1.4.2 Degree sequence

Definition 1.17. ***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 1.1 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

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

Definition 1.18. ***Graphical sequence.*** *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.*

Let $S = (d_i)_{i=1}^n$ 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 1.8 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 non-negative and even.

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 1.11 are isomorphic so they have the same number of vertices and edges. Also, G_1 and G_2 in Figure 1.11 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.

Problems 1.4

1. 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 1.16 for incidence matrices.)

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

$$G_1 \cup G_2 = (V_1 \cup V_2,\ E1 \cup E2).$$

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.

The *intersection* 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

$$G_1 \cap G_2 = (V_1 \cap V_2,\ E1 \cap E2).$$

In case $V_1 = V_2$, then $G_1 \cap G_2$ is simply the graph consisting of all edges in G_1 and in G_2.

The *symmetric difference* (or *ring sum*) 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 symmetric difference is the graph

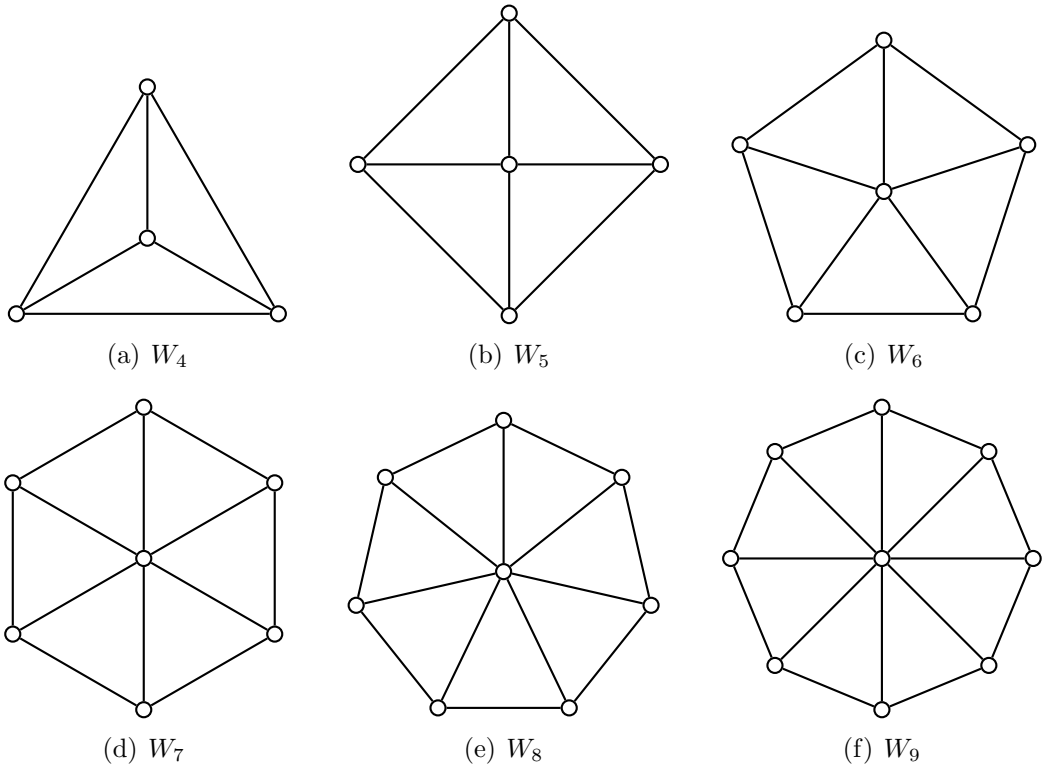
$$G_1 \Delta G_2 = (V_1 \Delta V_2,\ E1 \Delta E2).$$

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 graph consisting of all edges in G_1 or in G_2, but not in both. In this case, sometimes $G_1 \Delta G_2$ is written as $G_1 \oplus G_2$.

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 1.12 shows various wheel graphs.



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

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

Figure 1.12: The wheel graphs W_n for $n = 4, \ldots, 9$.

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
```

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
```

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.

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.

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.

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. 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 1.19. *If $G = (V, E)$ is self-complementary, then the order of G is $|V| = 4k$ or $|V| = 4k + 1$ for some non-negative 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 (1.2), 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 [23, Theorem 1.9] we have $4 \mid n$ and so $n = 4k$ for some non-negative $k \in \mathbf{Z}$. If $n-1$ is even, use a similar argument to conclude that $n = 4k + 1$ for some non-negative $k \in \mathbf{Z}$. □

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 $V(G) \times V(H)$. 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 $V(G \square H) = V(G) \times V(H)$ and the edge set of $G \square H$ is the union

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

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
```

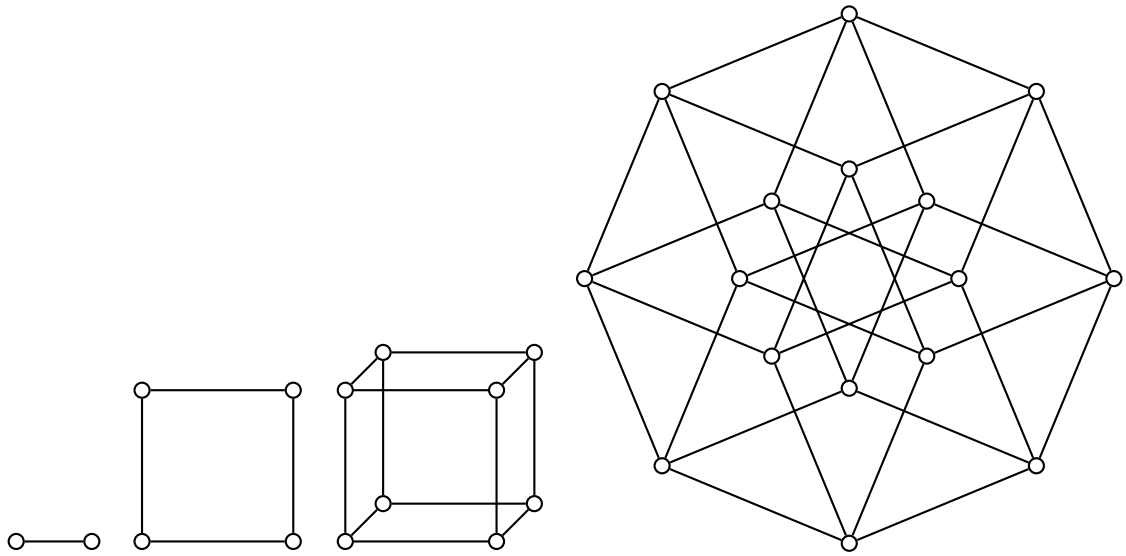
The *hypercube graph* Q_n is the n-regular graph with vertex set

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

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.[3] The Cartesian product of n edge graphs K_2 is a hypercube:

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

Figure 1.13 illustrates the hypercube graphs Q_n for $n = 1, \ldots, 4$.



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

Figure 1.13: Hypercube graphs Q_n for $n = 1, \ldots, 4$.

Example 1.20. *The Cartesian product of two hypercube graphs is another hypercube:* $Q_i \square Q_j = Q_{i+j}$.

The *path graph* P_n is a tree with n vertices $V = \{v_1, \ldots, v_n\}$ and edges $E = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n-1\}$. 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. $L_n = P_n \square P_1$.

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

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.

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 1.21. ***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 6.

Problems 1.5

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

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

1.6 Common applications

A few other common problems arising in applications of weighted graphs are listed below.

- If the edge weights are all non-negative, 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 5.

- 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 4 and 9.

- 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 6.

- 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*. 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 7.

- In some fields, such as operations research, a directed graph with non-negative 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 8.

Chapter 2

Graph Algorithms

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. We shall examine several in this chapter.

2.1 Graph searching

This section discusses algorithms for

- breadth-first searches,
- depth-first searches, and
- we explain how these relate to determining a graph's connectivity.

2.1.1 Breadth-first search

Breadth-first search (BFS) is a strategy for running through the nodes of a graph. Suppose you want to count the number of vertices (or edges) satisfying a property P. Algorithm 2.1 presents a technique for finding the number of vertices satisfying P.

Another version of Algorithm 2.1 is where you are searching the graph for a vertex (or edge) satisfying a certain property P. In that situation, you simply quit at the step where you increment the counter, i.e. line 7 in Algorithm 2.1. Other variations are also possible as well.

For the example of the graph in Figure 1.4, the list of distances from vertex `a` to any other vertex is

```
[['a', 0], ['b', 1], ['c', 2], ['d', 3], ['e', 1], ['f', 2], ['g', 2]]
```

To create this list,

- Start at `a` and compute the distance from `a` to itself.

Input : A connected graph $G = (V, E)$ (and, optionally, a starting or "root" vertex $v_0 \in V$). A property P to be tested.
Output: The number of vertices of G satisfying P.

```
1  Create a queue Q of "unseen" vertices initially containing a starting vertex v_0.
2  Start a list T of "already seen" vertices initially empty.
3  count ← 0
4  for w ∈ Q do
5  |  Test w for P.
6  |  if P(w) = True then
7  |  |  count ← count + 1
8  |  end
9  |  Add all neighbors of w not in T to Q.
10 |  Remove all "seen" vertices w from Q.
11 |  Add such w to T.
12 |  if T = V then
13 |  |  return count
14 |  end
15 end
```

Algorithm 2.1: Breadth-first search.

- Move to each neighbor of `a`, namely `b` and `e`, and compute the distance from `a` to each of them.
- Move to each "unseen" neighbor of `b`, namely just `c`, and compute the distance from `a` to it.
- Move to each "unseen" neighbor of `e`, namely just `f`, and compute the distance from `a` to it.
- Move to each "unseen" neighbor of `c`, namely just `d`, and compute the distance from `a` to it.
- Move to each "unseen" neighbor of `f`, namely just `g`, and compute the distance from `a` to it.

As an example, here is some Sage code which implements BFS to compute the list distances from a given vertex.

```
def graph_distance(G, v0):
    """
    Breadth first search algorithm to find the
    distance from a fixed vertex $v_0$ to any
    other vertex.

    INPUT:
        G - a connected graph
        v0 - a vertex

    OUTPUT:
        D - a list of distances to
            every other vertex

    EXAMPLES:
        sage: G = Graph({1: [2, 4], 2: [1, 4], 3: [2, 6],
```

```
                        4: [1, 3], 5: [4, 2], 6: [3, 1]})
        sage: v0 = 1
        sage: graph_distance(G,v0)
        [[1, 0], [2, 1], [3, 2], [4, 1], [5, 2], [6, 1]]
        sage: G = Graph({"a": ["b", "e"], "b": ["c", "e"], \
         "c": ["d", "e"], "d": ["f"], "e": ["f"], "f": ["g"], "g":["b"]})
        sage: v0 = "a"
        sage: graph_distance(G, v0)
        [['a', 0], ['b', 1], ['c', 2], ['d', 3], ['e', 1],
         ['f', 2], ['g', 2]]
        sage: G = Graph({1: [2,3], 2: [1, 3], 3: [2], 4: [5], 5: [6], 6: [5]})
        sage: v0 = 1
        sage: graph_distance(G, v0) # note G is disconnected
        [[1, 0], [2, 1], [3, 1]]
    """
    V = G.vertices()
    Q = [v0]
    T = []
    D = []
    while Q<>[] and T<>V:
        for v in Q:
            if not(v in T):
                D.append([v,G.distance(v0,v)])
            if v in Q:
                Q.remove(v)
            T.append(v)
            T = list(Set(T))
            Q = Q+[x for x in G.neighbors(v) if not(x in T+Q)]
            if T == V:
                break
    D.sort()
    print Q, T
    return D
```

Exercise 2.1. *Using Sage's* `shortest_path` *method, can you modify the above function to return a list of shortest paths from v_0 to any other vertex?*

2.1.2 Depth-first search

A depth-first search is a type of algorithm that visits each vertex of a graph, proceeding from vertex-to-vertex in this search but moving along a spanning tree of that graph.

Suppose you have a normal 8×8 chess board in front of you, with a single knight piece on the board. If you can find a sequence of knight moves which visits each and every square exactly once, then you will have found a so-called *complete knight tour*. Naively, how do you find a complete knight tour? Intuitively, you would make one knight move after another, recording each move to ensure that you did not step on a square you have already visited, until you could not make any more moves. It is very, very unlikely that if do this you will have visited every square exactly once (if you don't believe me, please try it yourself!). Acknowledging defeat, at this stage, it might make sense to *backtrack* a few moves and try again, hoping you will not get "stuck" so soon. If you fail again, try backtracking a few move moves and traverse yet another path, hoping to make further progress. Repeat this until a compete tour is found. This is an example of *depth-first search*, also sometimes called *backtracking*.

Similar to BFS, *depth-first search* (DFS) is an algorithm for traversing a graph. One starts at a *root vertex* and explores as *far as possible* along each branch before, if necessary, backtracking along a new path. It is easier to see what this means in the case of a rooted tree than for more general graphs, as illustrated below.

Suppose you want to count the number of vertices (or edges) satisfying a property P.

In the case of a graph, you can modify Algorithm 2.2 to a so-called iterative DFS. This modification applies DFS repeatedly with an increasing depth of search at each

Input : A rooted tree $G = (V, E)$ with root vertex $v_0 \in V$.
Output: True if G has a vertex satisfying P; False otherwise.

```
1  Create a queue Q of "child" vertices of the root v_0.
2  Initialize a list S of "seen" vertices.
3  count ← 0
4  for w ∈ Q do
5  |  Test w for P.
6  |  if P(w) = True then
7  |  |  count ← count + 1
8  |  end
9  |  Add w to S.
10 |  if S = V then
11 |  |  return count
12 |  end
13 end
14 Call the DFS algorithm iteratively with the rooted subtree having w and all its
   children as vertices and w as the rooted vertex.
```

Algorithm 2.2: Depth-first search.

step, until the diameter of the graph is reached and all vertices are seen.

2.1.3 Application: connectivity of a graph

A simple algorithm to determine if a graph is connected might be described as follows:

- Begin at any arbitrary vertex of the graph, $\Gamma = (V, E)$.
- Proceed from that vertex using either DFS or BFS, counting all vertices reached.
- Once the connected component of the graph has been entirely traversed, if the number of vertices counted is equal to $|V|$, the graph is connected and otherwise it is disconnected.

2.2 Shortest path algorithms

Let $G = (V, E)$ be a graph with non-negative edge weights, $w(e)$ for $e \in E$. The *length of a path* P from $v \in V$ to $w \in V$ is the sum of the edge weights for each edge in the path P, denoted $\delta(P)$. We write $\delta(v, w)$ for the smallest value of $\delta(P)$ for all paths P from v to w. When we regard these weights w as distances, a path from v to w which realizes $\delta(v, w)$ is sometimes called a *shortest path* from v to w.

There are a number of different algorithms 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 you use, the length of the shortest path cannot exceed the number of vertices in the graph.

Lemma 2.2. Fix a vertex v in the connected graph $G = (V, E)$ and let n denote the number of vertices of G, $n = |V|$. If there are no negative weight cycles in G then there exists a shortest path from v to any other vertex $w \in V$ which 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. Let P denote such a path,

$$P = (v_0 = v \to v_1 \to v_2 \to \cdots \to v_k = w).$$

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 path P' from v to w of lower weight. Since the final path is acyclic, it must have no more than $n - 1$ edges. □

2.2.1 Dijkstra's algorithm

See Dijkstra [12], section 24.3 of Cormen et al. [11], and section 12.6 of Berman and Paul [5].

Dijkstra's algorithm, discovered by E. Dijkstra in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights. For example, if 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 the shortest route from a fixed city to all other cities.

Let $G = (V, E)$ be a graph with non-negative edge weights, as above. Fix a start or source vertex $v_0 \in V$.

Dijkstra's algorithm performs a number of steps, basically one step for each vertex in V. We partition the vertex set V into two subsets: the set F of vertices where we have found the shortest path to v_0; and the "queue" Q where we do not yet know for sure the shortest path to v_0. The vertices $v \in F$ are labeled with $\delta(v, v_0)$. The vertices $v \in Q$ are labeled with a temporary label $L(v)$. This temporary label can be either ∞ if no path from v to v_0 has yet been examined, or an upper bound on $\delta(v, v_0)$ obtained by computing $\delta(P)$ for a path P from v to v_0 which has been found (but may not be the shortest path).

The simplest implementation of Dijkstra's algorithm has running time $O(|V|^2) = O(n^2)$ (where $n = |V|$ is the number of vertices of the graph)[1].

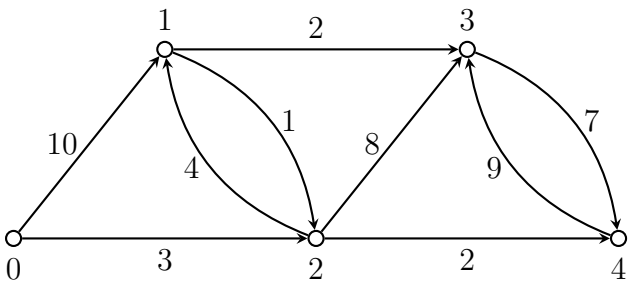


Figure 2.1: Searching a weighted digraph using Dijkstra's algorithm.

[1]This can be improved, with some clever programming, in the case of "sparse" graphs to $O(n \log n)$.

Input : A connected graph $G = (V, E)$ having non-negative edge weights and a starting vertex $v_0 \in V$.
Output: A shortest path from v_0 to an vertex in V.

1 Create a queue Q of "unseen" vertices initially being all of V.
2 Start a list F of "already seen" vertices initially empty.
3 Initialize labels $L(v_0) = 0$ and $L(v) = \infty$ for all $v \in V$ with $v \neq v_0$.
4 Find $v \in Q$ for which $L(v)$ is finite and minimum.
5 **if** no such v exists **then**
6 | **return**
7 **else**
8 | Label v with the distance $\delta(v, v_0) = L(v)$.
9 | Add v to F.
10 | Remove v from Q.
11 | **if** $F = V$ **then**
12 | | **return**
13 | **end**
14 **end**
15 **for** $w \in Q$ such that w is adjacent to v **do**
16 | Replace $L(w)$ by $\min(L(w),\ L(v) + wt(v, w))$.
17 | Go to step 4.
18 **end**

Algorithm 2.3: Dijkstra's algorithm.

v_0	v_1	v_2	v_3	v_4
$\underline{0}$	∞	∞	∞	∞
	10	$\underline{3}$	∞	∞
	7		11	$\underline{5}$
	$\underline{7}$		11	
			$\underline{9}$	

Table 2.1: Stepping through Dijkstra's algorithm.

Example 2.3. *Apply Dijkstra's algorithm to the graph in Figure 2.1.*

Solution. Dijkstra's algorithm applied to the graph in Figure 2.1 yields Table 2.1. The steps below explain how this table is created.

1. Start at v_0, let $Q = V$ and $F = \emptyset$. Initialize the labels $L(v)$ to be ∞ for all $v \neq v_0$. This is the first row of the table. Take the vertex v_0 out of the queue.

2. Consider the set of all adjacent nodes to v_0. Replace the labels in the first row by the weights of the associated edges. Underline the smallest one and take its vertex (i.e. v_2) out of the queue. This is the second row of the table.

3. Consider the set of all nodes w which are adjacent to v_2. Replace the labels in the second row by $\min(L(w),\, L(v_2) + wt(v_2, w))$. Underline the smallest one and take its vertex (i.e. v_4) out of the queue. This is the third row of the table.

4. Finally, start from v_4 and find the path to the remaining vertex v_3 in Q. Take the smallest distance from v_0 to v_3. This is the last row of the table.

□

Exercise 2.4. *Dijkstra's algorithm applied to the graph in Figure 2.2 results in Table 2.2. Verify the steps to create this table.*

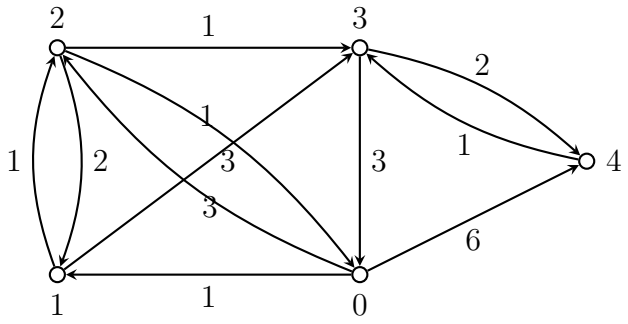


Figure 2.2: Searching a directed house graph using Dijkstra's algorithm.

v_0	v_1	v_2	v_3	v_4
$\underline{0}$	∞	∞	∞	∞
	$\underline{1}$	3	∞	6
		$\underline{2}$	4	6
			$\underline{3}$	6
				$\underline{5}$

Table 2.2: Another walk-through of Dijkstra's algorithm.

2.2.2 Bellman-Ford algorithm

See section 24.1 of Cormen et al. [11], and section 8.5 of Berman and Paul [5].

The Bellman-Ford algorithm computes single-source shortest paths in a weighted graph or digraph, where some of the edge weights may be negative. Instead of the "greedy" approach that Dijkstra's algorithm took, 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 implementation below takes in a graph or digraph, and creates two Python dictionaries `dist` and `predecessor`, keyed on the list of vertices, which store the distance and shortest paths. However, if a negative weight cycle exists (in the case of a digraph), then an error is raised.

```
def bellman_ford(Gamma, s):
    """
    Computes the shortest distance from s to all other vertices in Gamma.
    If Gamma has a negative weight cycle, then return an error.

    INPUT:

    - Gamma -- a graph.
    - s -- the source vertex.

    OUTPUT:

    - (d,p) -- pair of dictionaries keyed on the list of vertices,
      which store the distance and shortest paths.

    REFERENCE:

    http://en.wikipedia.org/wiki/Bellman-Ford_algorithm
    """
    P = []
    dist = {}
    predecessor = {}
    V = Gamma.vertices()
    E = Gamma.edges()
    for v in V:
        if v == s:
            dist[v] = 0
        else:
            dist[v] = infinity
        predecessor[v] = 0
    for i in range(1, len(V)):
        for e in E:
            u = e[0]
            v = e[1]
            wt = e[2]
            if dist[u] + wt < dist[v]:
                dist[v] = dist[u] + wt
                predecessor[v] = u
    # check for negative-weight cycles
    for e in E:
        u = e[0]
        v = e[1]
        wt = e[2]
        if dist[u] + wt < dist[v]:
            raise ValueError("Graph contains a negative-weight cycle")
    return dist, predecessor
```

Bellman-Ford runs in $O(|V| \cdot |E|)$-time, which is $O(n^3)$ for "dense" connected graphs (where $n = |V|$).

Here are some examples.

```
sage: M = matrix([[0,1,4,0], [0,0,1,5], [0,0,0,3], [0,0,0,0]])
sage: G = Graph(M, format="weighted_adjacency_matrix")
sage: bellman_ford(G, G.vertices()[0])
  {0: 0, 1: 1, 2: 2, 3: 5}
```

The plot of this graph is given in Figure 2.3.

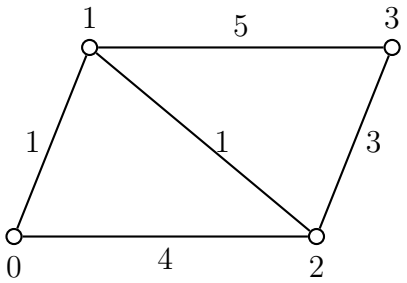


Figure 2.3: Shortest paths in a weighted graph using the Bellman-Ford algorithm.

The following example illustrates the case of a negative-weight cycle.

```
sage: M = matrix([[0,1,0,0],[1,0,-4,1],[1,1,0,0],[0,0,1,0]])
sage: G = DiGraph(M, format = "weighted_adjacency_matrix")
sage: bellman_ford(G, G.vertices()[0])
---------------------------------------------------------------------------
...
ValueError: Graph contains a negative-weight cycle
```

The plot of this graph is given in Figure 2.4.

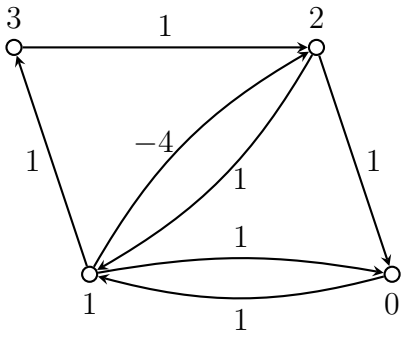


Figure 2.4: Searching a digraph with negative weight using the Bellman-Ford algorithm.

2.2.3 Floyd-Roy-Warshall algorithm

See section 25.2 of Cormen et al. [11], and section 14.4 of Berman and Paul [5].

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 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 G as $V = \{1, 2, \ldots, n\}$. Call $SD(i, j, k)$ a shortest distance from vertex i to vertex j that only uses vertices 1 through k. This can be computed using the recursive expression

$$SD(i, j, k) = \min\{SD(i, j, k-1),\ SD(i, k, k-1) + SD(k, j, k-1)\}.$$

The key to the Floyd-Roy-Warshall algorithm lies in exploiting this formula. 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 eminating 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 interated implementation of Bellman-Ford.

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 2.5.

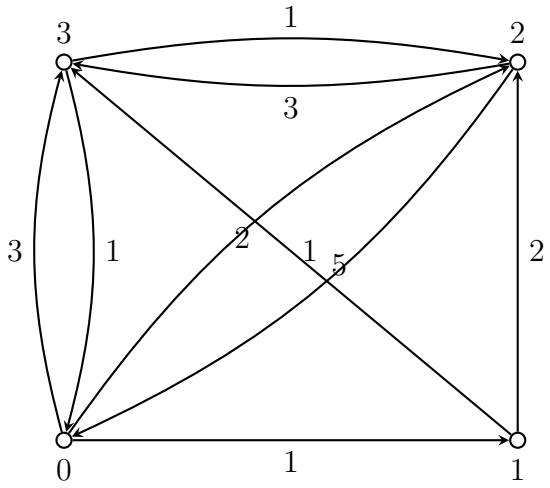


Figure 2.5: Demonstrating the Floyd-Roy-Warshall algorithm.

```
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 2.6.

2.2.4 Johnson's algorithm

See section 25.3 of Cormen et al. [11] and Johnson [16].

Let $G = (V, E)$ be a graph with edge weights but no negative cycles. *Johnson's algorithm* finds a shortest path between all pairs of vertices in a "sparse" directed graph.

The time complexity, for sparse graphs, is $O(|V|^2 \log |V| + |V| \cdot |E)| = O(n^2 \log n)$, where $n = |V|$ is the number of vertices of the original graph G.

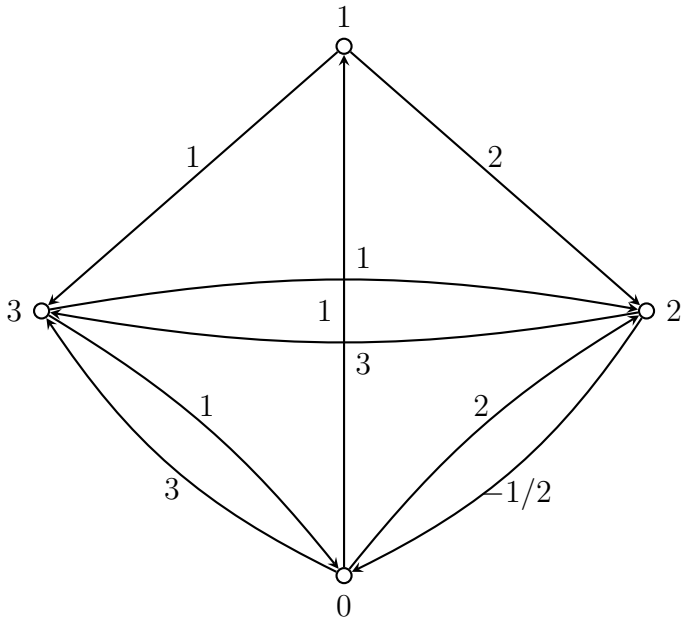


Figure 2.6: Another demonstration of the Floyd-Roy-Warshall algorithm.

Input : A connected graph $G = (V, E)$ having (possibly negative) edge weights.
Output: A shortest path between all pairs of vertices in V (or terminate if a negative edge cycle is detected).

1 Add a new vertex v_0 with zero weight edges from it to all $v \in V$.
2 Run the Bellman-Ford algorithm to check for negative weight cycles and find $h(v)$, the least weight of a path from the new node v_0 to $v \in V$.
3 If the last step detects a negative cycle, the algorithm is terminated.
4 Reweight the edges using the vertices' $h(v)$ values: an edge from $v \in V$ to $w \in V$, having length $wt(v, w)$, is given the new length $wt(v, w) + h(v) - h(w)$.
5 For each $v \in V$, run Dijkstra's algorithm and store the computed least weight to other vertices.

Algorithm 2.4: Johnson's algorithm.

Chapter 3

Trees and Forests

Recall, 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. A vertex of a forest of degree one is called an *endpoint* or a *leaf*. A connected forest is a *tree*.

A *rooted tree* is a tree with a specified *root* vertex v_0. (However, if G is a rooted tree with root vertex v_0 and if the degree of v_0 is one then, by convention, we do not call v_0 an endpoint or a leaf.) A *directed tree* is a directed graph which would be a tree if the directions on the edges were ignored. A rooted tree can be regarded as a directed tree since you imagine an edge $E = \{u, v\}$, for $u, v \in V$, being directed from u to v, $e = (u, v)$, if and only if v is further away from v_0 than u is. If $e = (u, v)$ is an edge in a rooted tree, then we call v a *child* vertex with *parent* u. 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*.

Directed trees are pervasive in theoretical computer science, as they are useful structures for describing algorithms and relationships between objects in certain data sets.

A *spanning tree* T of a connected, undirected graph G is a subgraph containing all vertices of G which is a tree.

Example 3.1. Consider the 3×3 grid graph with 16 vertices and 18 edges. Two examples of a spanning tree are given in Figure 3.1 by using thicker line width for its edges.

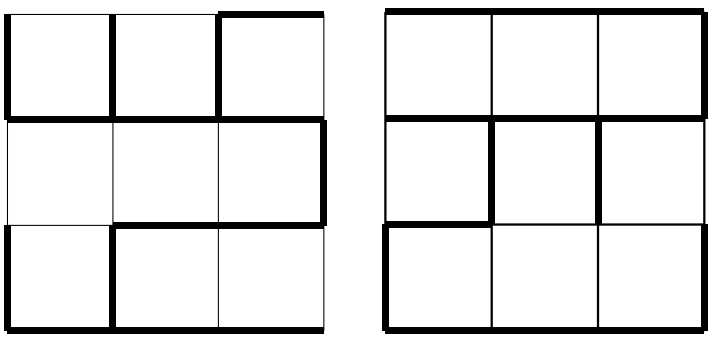


Figure 3.1: Spanning trees for the 4×4 grid graph.

The following game is a variant of the Shannon switching game, due to Edmunds and Lehman. We follow the description in Oxley's survey (*What is a matroid?' ...* add reference later ...).

Recall 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.

Theorem 3.2. *The following statements are equivalent for a connected graph G.*

- *Bob plays first and Alice can win against all possible strategies of Bob.*
- *The graph G has 2 edge-disjoint spanning trees.*
- *For all partitions P of the vertex set V of G, the number of edges of G that join vertices in different classes of the partition is at least $2(|P| - 1)$.*

3.1 Properties of trees

The following theorem gives several basic characterizations of trees.

Theorem 3.3. *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 $u, v \in V$, there is exactly one u-v path.*
6. *For any new edge e, the join $T + e$ has exactly one cycle.*

Let $G = (V_1, E_2)$ be a graph and $T = (V_2, E_2)$ a subgraph of G which is a tree. As in (6) we see 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.)

Solution. (1) $\implies$ (2): This basically follows by induction on the number of vertices.

By definition, a tree has no cycles. Make the following induction hypothesis: for any tree $T = (V, E)$, $|E| = |V| - 1$. This holds in the base case where $|V| = 1$ since in that case, there can be no edges. Assume it is true for all trees with $|V| = k$, for some $k > 1$. Let $T = (V, E)$ be a tree having $k + 1$ vertices. Remove an edge (but not the vertices it is incident to). This disconnects T into $T_1 = (V_1, E_1)$ union $T_2 = (V_2, E_2)$,

where $|E| = |E_1| + |E_2| + 1$ and $|V| = |V_1| + |V_2|$ (and possibly one of the E_i is empty), each of which is a tree satisfying the conditions of the induction hypothesis. Therefore,

$$|E| = |E_1| + |E_2| + 1 = |V_1| - 1 + |V_2| - 1 + 1 = |V| - 1.$$

(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. Each of these satisfy, by (2),

$$|E_i| = |V_i| - 1,$$

so

$$|E| = \sum_{i=1}^{k} |E_i| = \sum_{i=1}^{k} |V_i| - k = |V| - k.$$

This contradicts (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 (3). Therefore e is a cut set.
(4) $\implies$ (5): Let

$$P = (v_0 = u \to v_1 \to v_2 \to \cdots \to v_k = v)$$

and

$$P' = (v'_0 = u \to v'_1 \to v'_2 \to \cdots \to v'_\ell = v)$$

be two paths from u to v.
(5) $\implies$ (6): Let $e = (u, v)$ be a new edge connecting $u, v \in V$. Suppose that

$$P = (v_0 = w \to v_1 \to v_2 \to \cdots \to v_k = w)$$

and

$$P' = (v'_0 = w \to v'_1 \to v'_2 \to \cdots \to v'_\ell = w)$$

are two cycles in $T \cup (\{u, v\}, \{e\})$.

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 $v_0 = w \to v_1$ is a u-v path and the sequence $v = v_1 \to v_2 \to \cdots \to v_k = w = u$ taken in reverse order is another u-v path. This is a contradiction to (5).

We may suppose now that P and P' both contain e. Therefore, 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 (5), these u-v paths p_0 and P'_0 must be the same. This forces P and P' to be the same, which proves (6).
(6) $\implies$ (1): Condition (6) 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 $e = (u, v)$ does not create a cycle (if it did then T_1 and T_2 would not be disjoint), which contradicts (6). □

Exercise 3.4. *Let $G = (V_1, E_2)$ be a graph and $T = (V_2, E_2)$ a spanning tree of G. Show there is a one-to-one correspondence between fundamental cycles in G and edges not in T.*

Exercise 3.5. *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 which is not a fundamental cycle in G for T_2.*

Exercise 3.6. *Usually there exist many spanning trees of a graph. Can you 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 then T is unique.*

3.2 Minimum spanning trees

Suppose you want to design an electronic circuit connecting several components. If these components represent the vertices of the graph and a wire connecting two components represents an edge of the graph then, for economical reasons, you will want to connect these together using the least amount of wire. This amounts to finding a minimum spanning tree in the complete graph on these vertices.

- spanning trees

  We can characterize a spanning tree in several ways. Each of these conditions lead to an algorithm for constructing them.

  One condition is that spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle. Another condition is that it is a minimal set of edges that connect all vertices.

  Exploiting the former criteria gives rise to Kruskal's algorithm. Exploiting the latter criteria gives rise to Prim's algorithm. Both of these argorithms are discussed in more detail below.

- minimum-cost spanning trees

  A *minimum spanning tree* (MST) is a spanning tree of an edge weighted graph having lowest total weight among all possible spanning trees.

- Kruskal's algorithm [18]; see also section 23.2 of Cormen et al. [11].

  Kruskal's algorithm is a greedy algorithm to compute a MST. It was discovered by J. Kruskal in the 1950's.

  Kruskal's algorithm can be shown to run in $O(|E| \log |E|)$ time.

- Prim's algorithm [22]; see also section 23.2 of Cormen et al. [11].

  Prim's algorithm is a greedy algorithm to compute a MST. In can be implemented in time $O(|E| + |V| \log |V|)$, which is $O(n^2)$ for a dense graph having n vertices.

  The algorithm was developed in the 1930's by Czech mathematician V. Jarník and later independently by both the computer scientists R. Prim and E. Dijkstra in the 1950's.

- Borůvka's algorithm [8, 9]

  Borůvka's algorithm is an algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct. It was first published in 1926 by Otakar Borøuvka but then rediscovered by many others.

  Borůvka's algorithm can be shown to run in time $O(|E| \log |V|)$.

3.2.1 Kruskal's algorithm

Kruskal's algorithm starts with an edge-weighted digraph $G = (V, E)$ as input. Let $w : E \to \mathbb{R}$ denote the weight function. The first stage is to create a "skeleton" of the tree T which is initially set to be a graph with no edges: $T = (V, \emptyset)$. The next stage is to sort the edges of G by weight. In other words, we label the edges of G as

$$E = \{e_1, e_2, \ldots, e_m\},$$

where $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$. Next, start a for loop over $e \in E$. You add e to T as an edge provided it does not create a cycle. The only way adding $e = (u, v)$ to T would create a cycle would be if both u and v were endpoints of an edge already in T. As long as this cycle condition fails, you add e to T and otherwise, go to the next element of E in the for loop. At the end of the for loop, the edges of T have been completely found and the algorithm stops.

SAGE

```
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 are some examples.

We start with the grid graph. This is implemented in Sage in a way 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 assumes that the vertices are $V = \{0, 1, \ldots, n-1\}$, we first redefine the graph suitable and run the Kruskal algorithm on that.

```
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)]
```

The plot of this graph is given in Figure 3.2.1.

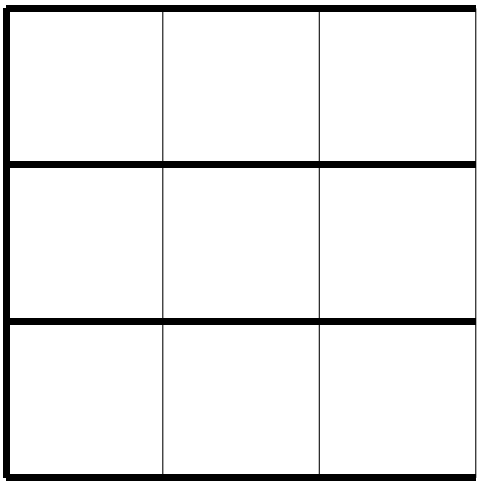


Figure 3.2: Kruskal's algorithm for the 4×4 grid graph.

3.2.2 Prim's algorithm

Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph $\Gamma = (V, E)$. It is very similar to Kruskal's algorithm except that it starts with an empty vertex set, rather than a full one.

Input : A connected graph $G = (V, E)$ having edge weights.
Output: A MST T for G.

1 Initialize: $V(T) = \{v_0\}$, where v_0 is an arbitrary vertex, $E(T) = \emptyset$
2 While $V(T) \neq V$:
3 Choose edge (u, v) with minimal weight such that u is in $V(T)$ but v is not,
4 Add v to $V(T)$, add (u, v) to $E(T)$.

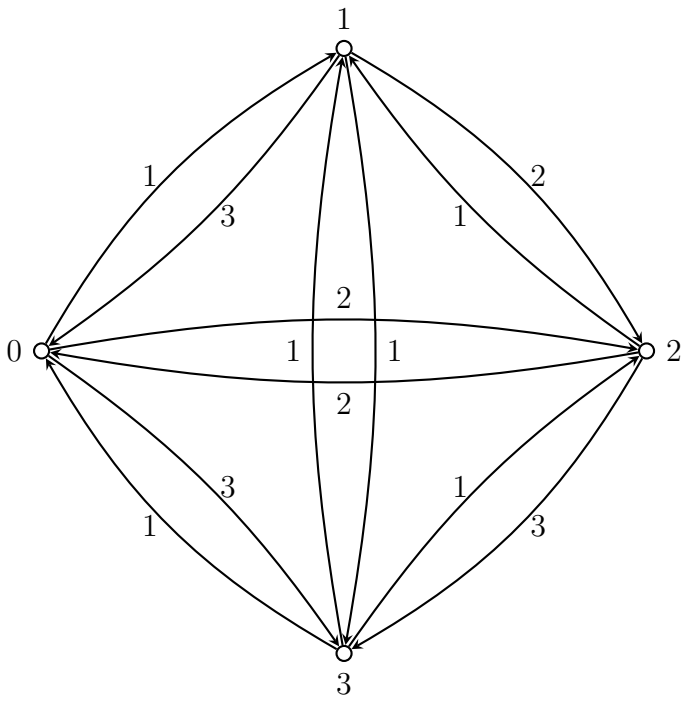
Algorithm 3.1: Prim's algorithm.

SAGE

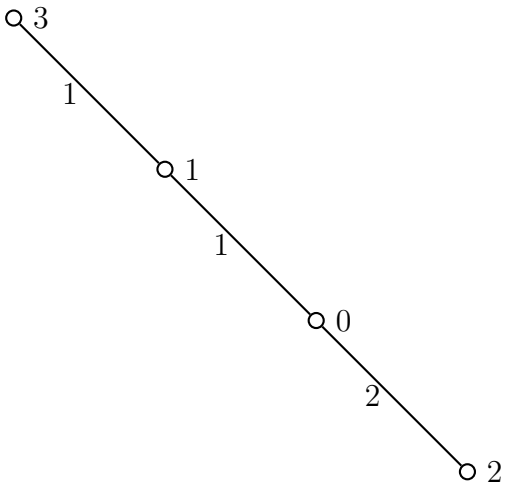
```
def prim(G):
    """
    Implements Prim's algorithm to compute a MST of a graph.

    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)



(b)

Figure 3.3: Prim's algorithm for digraphs. Above is the original digraph and below is the MST produced by Prim's algorithm.

```
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],[0,0,0,0,0,0,8,9],[0,0,0,0,0,0,0,11],[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)]
```

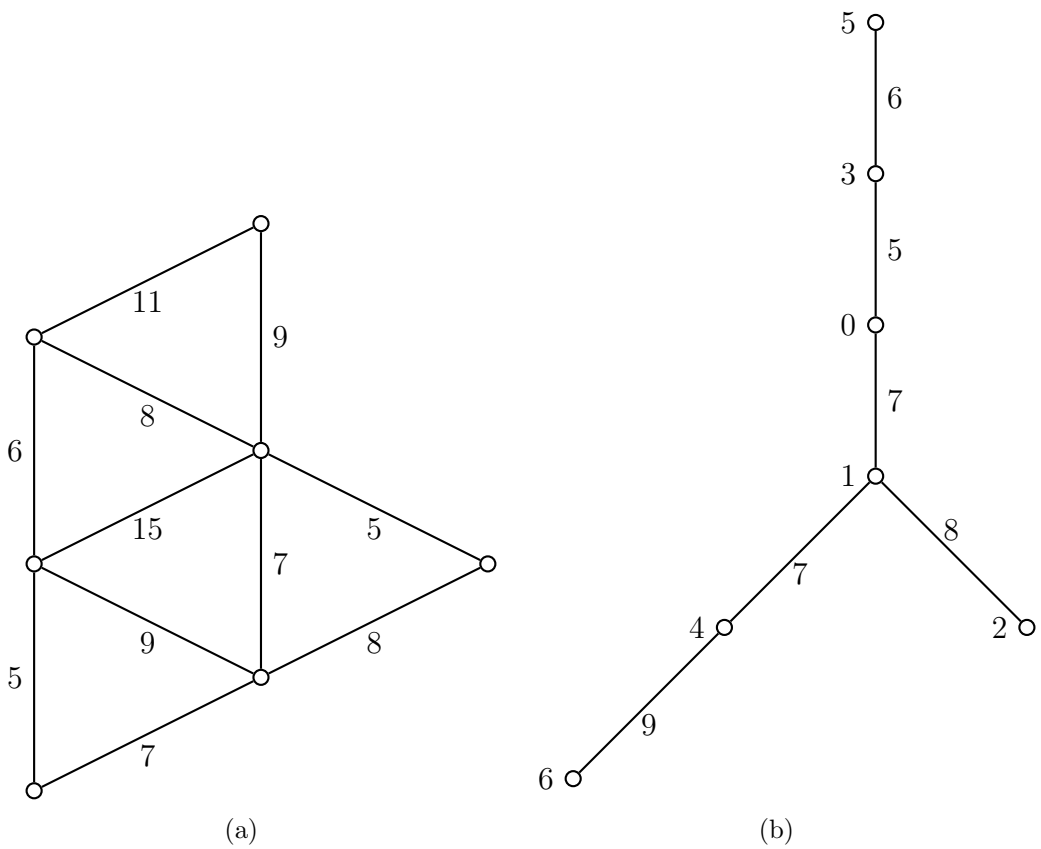


Figure 3.4: Another example of Prim's algorithm. On the left is the original graph. On the right is the MST produced by Prim's algorithm.

3.2.3 Borůvka's algorithm

Borůvka's algorithm algorithm is an algorithm for finding a minimum spanning tree in a connected graph for which all edge weights are distinct.

Pseudocode for Borůvka's algorithm is:

- Begin with a connected graph G containing edges of distinct weights, and an empty set of edges T
- While the vertices of G connected by T are disjoint:
  - Begin with an empty set of edges E
  - For each component:
    - Begin with an empty set of edges S
    - For each vertex in the component:

· Add the cheapest edge from the vertex in the component to another vertex in a disjoint component to S

Add the cheapest edge in S to E

Add the resulting set of edges E to T.

The resulting set of edges T is the minimum spanning tree of G.

Example 3.7. *In Figure 3.5 , we plot the following example.*

```
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)
```

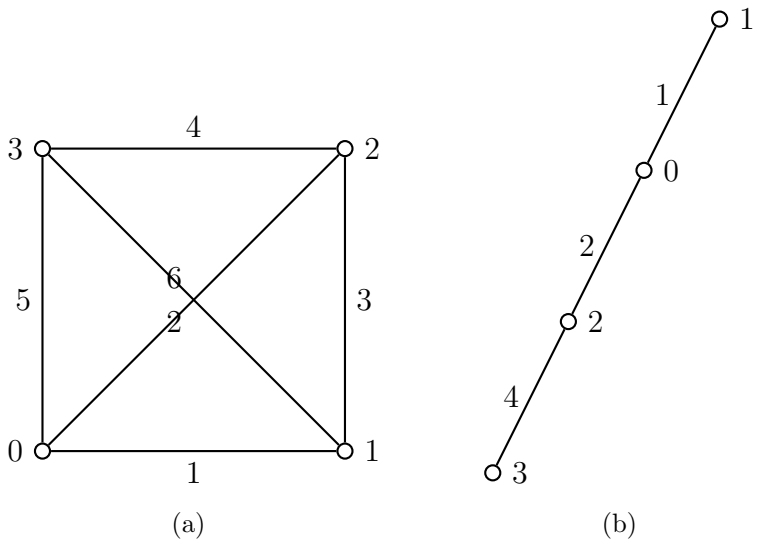


Figure 3.5: An example of Borovka's algorithm. On the left is the original graph. On the right is the MST produced by Boruvka's algorithm.

SAGE

```
def which_index(x,L):
    """
    L is a list of sublists (or tuple of sets or list
    of tuples, etc).

    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 Binary trees

See section 3.3 of Gross and Yellen [14].

A binary tree is a rooted tree with at most 2 children per parent.

In this section, we consider

- binary codes,
- Gray codes, and
- Huffman codes.

3.3.1 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 (and 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 $\{0, 1, \ldots, 255\}$. If these numbers are written in binary then each codeword of a letter has length 8. 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.

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 A, one might be given a "weighted alphabet," i.e., an alphabet for which each symbol $a \in A$ is associated with a non-negative number $w_a \geq 0$ (in practice, 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 ae called "cryptosystems." In this case, the inverse of the coding function $c : A \to B^*$ 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 (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 (start) of any other codeword[1]. This is the *prefix-free condition*.

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 Figure 3.6 (we use 0 for $\cdot$ ("dit") and 1 for $-$ ("dah")).

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	

Figure 3.6: Morse code

For example, look at the Morse code for `a` and the Morse code for `w`. These codewords violate the prefix-free condition.

[1]In other words, a codeword $s = s_1 \dots s_m$ is a *prefix* of a codeword $t = t_1 \dots t_n$ if and only if $m \leq n$ and $s_1 = t_1, \dots, s_m = t_m$. Codes which are prefix-free are easier to decode than codes which are not prefix-free.

Gray codes

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 patented a device (used for television sets) based on it in 1953. 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 E. Baudot's (a French engineer) telegraph machine of 1878 and in a French booklet by L. Gros on the solution to the "Chinese ring puzzle" published in 1872.

The term "Gray code" is ambiguous. It is actually a large family of sequences of n-tuples. Let $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. More precisely, an *$m$-ary Gray code of length n* (called a *binary Gray code* when $m = 2$) is a sequence of all possible (namely, $N = m^n$) n-tuples

$$g_1, g_2, \ldots, g_N,$$

where

- each $g_i \in \mathbb{Z}_m^n$,
- 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 $\mathbb{Z}_m$. 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 you can compare an n-tuple with the previous one. If they difer in more than one coordinate, you know an error was made.

Example 3.8. 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 here is a binary Gray code of length 3:

$$[0,0,0], [1,0,0], [1,1,0], [0,1,0], [0,1,1], [1,1,1], [1,0,1], [0,0,1].$$

Gray codes have applications to engineering, recreational mathematics (solving the Tower of Hanoi puzzle, "The Brain" puzzle, the "Chinese ring puzzle", and others), and to mathematics (for example, aspects of combinatorics, computational group theory and the computational aspects of linear codes).

[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**.

Binary Gray codes

Consider the so-called n-hypercube graph Q_n. 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 $\mathbb{R}^n$ connecting two "neighboring" vertices (namely, two vertices which differ in exactly one coordinate). A binary Gray code of length n can be regarded as a path on the hypercube graph Q_n which visits each vertex of the cube exactly once. In other words, a binary Gray code of length n may be identified with a Hamiltonian cycle on the graph Q_n (see Figure 3.7 for an example).

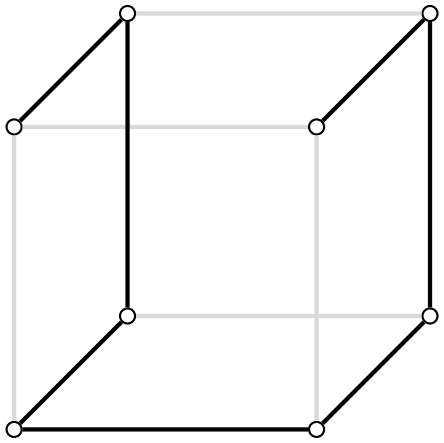


Figure 3.7: Viewing Γ_3 as a Hamiltonian path on Q_3.

How do you 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:

$$\Gamma_0 = [], \quad \Gamma_n = [0, \Gamma_{n-1}], [1, \Gamma_{n-1}^{rev}],$$

where Γ_m^{rev} means the Gray code in reverse order. For instance, we have

$$\Gamma_0 = [],$$
$$\Gamma_1 = [0], [1],$$
$$\Gamma_2 = [[0, 0], [0, 1], [1, 1], [1, 0],$$

and so on. This is a nice procedure if you want to create the entire list at once (which, by the way, gets very long very fast).

An implementation of the reflected Gray code using Python is given below.

Python 3.0

```
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, 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 $j \in \mathbb{Z}_{256}$ which 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 **??**.

What if you only want to compute the i-th Gray codeword in the Gray code of length n? Can it be computed quickly as well 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.

SAGE

```
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)
```

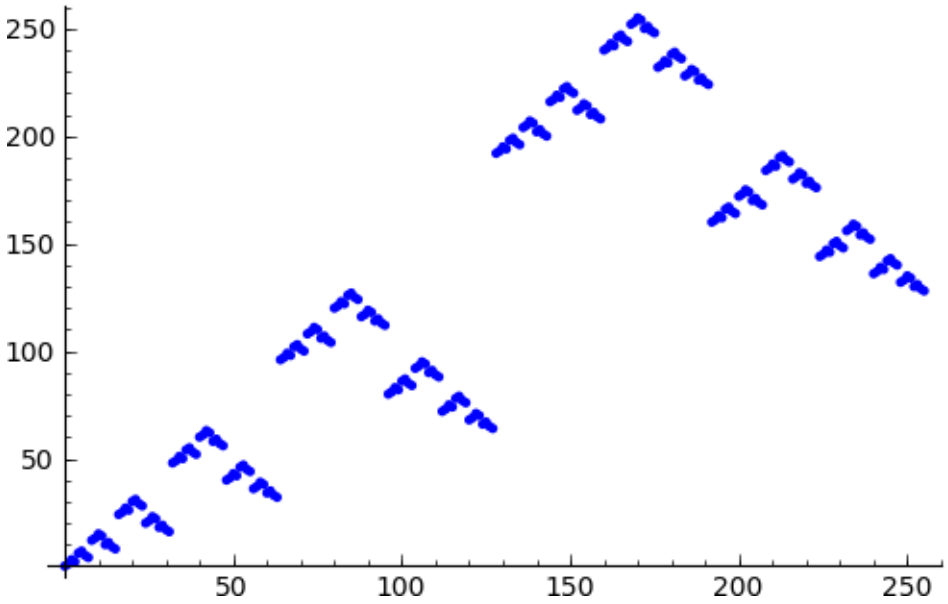


Figure 3.8: List plot of Γ_8 created using Sage.

```
        ....:
        (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 kth codeword in the reflected binary Gray code
    of length n.

    EXAMPLES:
        sage: graycodeword(3,3)
        (0, 1, 0)
    '''
    return int2binary(m,n)+int2binary(int(m/2),n)
```

Exercise 3.9. *Convert the above function* `graycodeword` *into a pure Python function.*

3.3.2 Huffman codes and Huffman's algorithm

An *alphabet* A is a finite set, whose elements are referred to as *symbols*.

A *word* (or *string* or *message*) in A is a finite sequence of symbols in A, usually written by simply concatenating them together: $a_1a_2\ldots a_k$ $(a_i \in A)$ is a message of *length* k. A commonly occurring alphabet in practice is the *binary alphabet* $\{0,1\}$, in which case a word is simply a finite sequence of 0's and 1's. If A is an alphabet, let

$$A^*$$

denote the set of all words in A. The length of a word is denoted by vertical bars: if $w = a_1 \dots a_k$ is a word in A then define $|\dots| : A^* \to \mathbb{R}$ by

$$|a_1 \dots a_k| = k.$$

Let A and B be two alphabets. A *code* for A using B is an injection $c : A \to B^*$. By abuse of notation, we often denote the code simply by the set

$$C = c(A) = \{c(a) \mid a \in A\}.$$

The elements of C are called *codewords*. If B is the binary alphabet then C is called a *binary code*.

Tree representation

Any binary code can be represented by a tree.

Example 3.10. Here is how to represent the code $\mathbb{B}_\ell$ consisting of all binary strings of length $\leq \ell$. Start with the "root node" $v_\emptyset$ 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, v_{00} and v_{01}, correspond to the strings of length 2 which start with a 0, and the two children of v_1, v_{10} and v_{11}, correspond to the strings of length 2 which start with a 1. Continue creating child nodes until you reach length ℓ then 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 3.9 for an example when $\ell = 2$.

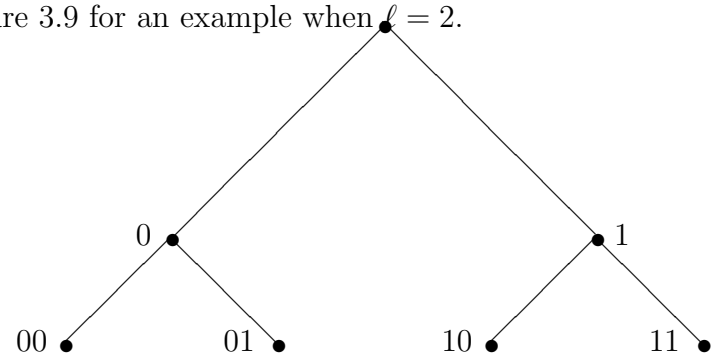


Figure 3.9: Example of a tree representation of a binary code

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 descendents are not in C. Next, remove all labels which do not correspond with 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 one 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

- The root node gets the empty codeword.
- Each left-ward branch gets a 0 appended to the end of its parent and each right-ward branch gets a 1 appended to the end.

Uniquely decodable codes

If $c : A \to B^*$ is a code then we can extend c to A^* by *concatenation*:

$$c(a_1 a_2 \dots a_k) = c(a_1)c(a_2)\dots c(a_k).$$

If the extension $c : A^* \to T^*$ is also an injection then c is called *uniquely decodable*.

Example 3.11. Recall the Morse code in Table **??**. Note these Morse codewords all have length 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 A denote the English alphabet, $B = \{0, 1\}$ the binary alphabet and $C : A \to B^*$ the Morse code. Since $c(ET) = 01 = c(A)$, it is clear that the Morse code is *not* uniquely decodable.

Exercise 3.12. *Show by giving an example that the Morse code is not prefix-free.*

In fact, prefix-free implies uniquely decodable.

Theorem 3.13. If a code $c : A \to B^*$ is prefix-free then it is uniquely decodable.

Solution. The proof is by induction on the length of a message. We want to show that if $x_1 \dots x_k$ and $y_1 \dots y_\ell$ are messages with $c(x_1)\dots c(x_k) = c(y_1)\dots c(y_\ell)$ then $x_1 \dots x_k = y_1 \dots y_\ell$ (which in turn implies $k = \ell$ and $x_i = y_i$ for all i).

The case of length 1 follows from the fact that $c : A \to B^*$ is injective (by the definition of a 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 $c(x_1)\dots c(x_k) = c(y_1)\dots c(y_\ell)$ 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 (for instance) $c(x_1)$ 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, $c(x_1) = c(y_1)$. This implies $x_1 = y_1$. Now remove this codeword from both sides, so $c(x_2)\dots c(x_k) = c(y_2)\dots c(y_\ell)$. By the induction hypothesis, $x_2 \dots x_k = y_2 \dots y_\ell$. These facts together imply $k = \ell$ and $x_i = y_i$ for all i. □

Consider now a weighted alphabet (A, p), where $p : A \to [0, 1]$ satisfies $\sum_{a\in A} p(a) = 1$, and a code $c : A \to B^*$. In other words, p is a probability distribution on A. Think of $p(a)$ as the probability that the symbol a arises in an typical message.

The *average word length* $L(c)$ is[3]:

[3]In probability terminology, this is the expected value $E(X)$ of the random variable X which assigns to a randomly selected symbol in A, the length of the associated codeword in C.

$$L(c) = \sum_{a \in A} p(a) \cdot |c(a)|,$$

where $|\dots|$ denotes the *length* of a codeword. .

Given a weighted alphabet (A, p) as above, a code $c : A \to B^*$ is called *optimal* if there is no such code with a smaller average word length.

Optimal codes satisfy the following amazing property.

Lemma 3.14. Suppose $c : A \to B^*$ is a binary optimal prefix-free code and let $\ell = \max_{a \in A}|c(a)|$ denote the maximum length of a codeword. The following statements hold.

- If $|c(a')| > |c(a)|$ then $p(a') \leq p(a)$.
- The subset of codewords of length ℓ,

  $$C_\ell = \{c \in c(A) \mid |c(a)| = \ell\},$$

  contains two codewords of the form $b0$ and $b1$, for some $b \in B^*$.

For the proof (which is very easy and highly recommended for the student who is curious to see more), see Biggs§3.6, [6].

The Huffman code construction is based on the amazing second property in the above lemma yields an optimal prefix-free binary code.

Huffman code construction: Here is the recursive/inductive construction. We shall regard the binary Huffman code as a tree, as described above.

Suppose that the weighted alphabet (A, p) has n symbols. We assume inductively that there is an optimal prefix-free binary code for any weighted alphabet (A', p') having $< n$ symbols.

Huffman's rule 1: Let $a, a' \in A$ 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 (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 $s1$.

These two rules tell us how to inductively built the tree representation for the Huffman code of (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 don't exist, stop. Replace the weighted alphabet with the new weighted alphabet as in Huffman's rule 1.
- Add two nodes (labeled with a and a', resp.) to the tree, with parent a^* (see Huffman's rule 1).
- If there are no remaining symbols in A, label the parent a^* with the empty set and stop. Otherwise, go to the first step.

An example of this is below.

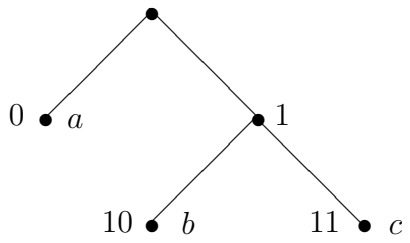


Figure 3.10: Huffman code example

Example 3.15. A very simple of this makes Huffman's ingenious construction easier to understand.

Suppse $A = \{a, b, c\}$ and $p(a) = 0.5$, $p(b) = 0.3$, $p(c) = 0.2$.

A Huffman code for this is $C = \{0, 10, 11\}$, as is depicted in Figure 3.10.

Exercise 3.16. *Verify that* $C = \{1, 00, 01\}$ *is another Huffman code for this weighted alphabet and to draw its tree representation.*

Exercise 3.17. *Find the Huffman code for the letters of the English alphabet weighted by the frequency of common American usage*[4]*.*

3.4 Applications to computer science

3.4.1 Tree traversals

See section 3.5 of Gross and Yellen [14]. See also `http://en.wikipedia.org/wiki/Tree_traversal`.

- stacks and queues
- breadth-first, or level-order, traversal
- depth-first, or pre-order, traversal
- post-order traversal
- symmetric, or in-order, traversal

In computer science, *tree traversal* refers to the process of examining each node in a tree data structure exactly once. We restrict our discussion to binary rooted trees.

Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type.
Depth-first traversal:

- Visit the root vertex.

[4]You can find this on the internet or in the literature. Part of this exercise is finding this frequency distribution yourself.

- Traverse the left subtree recursively.
- Traverse the right subtree recursively.

Breadth-first traversal:

- Initialize $i = 0$ and set N equal to the maximum depth of the tree (i.e., the maximum distance from the root vertex to any other vertex in the tree).
- Visit the vertices of depth i.
- Increment $i = i + 1$. If $i > N$ then stop. Otherwise, go to the previous step.

post-order traversal:

- Traverse the left subtree recursively.
- Visit the root vertex.
- Traverse the right subtree recursively.

symmetric traversal:

- Traverse the left subtree recursively.
- Visit the root vertex.
- Traverse the right subtree recursively.

3.4.2 Binary search trees

See section 3.6 of Gross and Yellen [14], and chapter 12 of Cormen et al. [11]. See also `http://en.wikipedia.org/wiki/Binary_search_tree`.

- records and keys
- searching a binary search tree (BST)
- inserting into a BST
- deleting from a BST
- traversing a BST
- sorting using BST

A *binary search tree* (BST) is a rooted binary tree $T = (V, E)$ having weighted vertices $\mathrm{wt} : V \to \mathbb{R}$ satisfying:

- The left subtree of a vertex v contains only vertices whose label (or "key") is less than the label of v.
- The right subtree of a vertex v contains only vertices whose label is greater than the label of v.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that: *Each vertex has a distinct label.*

Generally, the information represented by each vertex is a record (or list or dictionary), rather than a single data element. However, for sequencing purposes, vertices are compared according to their labels rather than any part of their associated records.

Traversal

The vertices of a BST T can be visited retrieved in-order of the weights of the vertices (i.e., using a symmetric search type) by recursively traversing the left subtree of the root vertex, then accessing the root vertex itself, then recursively traversing the right subtree of the root node.

Searching

We are given a BST (i.e., a binary rooted tree with weighted vertices having distinct weights satisfying the above criteria) T and a label ℓ. For this search, we are looking for a vertex in T whose label is ℓ, if one exists.

We begin by examining the root vertex, v_0. If $\ell = \text{wt}(v_0)$, the search is successful. If the $\ell < \text{wt}(v_0)$, search the left subtree. Similarly, if $\ell > \text{wt}(v_0)$, search the right subtree. This process is repeated until a vertex $v \in V$ is found for which $\ell = \text{wt}(v)$, or the indicated subtree is empty.

Insertion

We are given a BST (i.e., a binary rooted tree with weighted vertices having distinct weights satisfying the above criteria) T and a label ℓ. We assume ℓ is between the lowest weight of T and the highest weight. For this procedure, we are looking for a "parent" vertex in T which can "adopt" a new vertex v having weight ℓ and for which this augmented tree $T \cup v$ satisfies the criteria above.

Insertion proceeds as a search does. However, in this case, you are searching for vertices $v_1, v_2 \in V$ for which $\text{wt}(v_1) < \ell < \text{wt}(v_2)$. Once found, these vertices will tell you where to insert v.

Deletion

As above, we are given a BST T and a label ℓ. We assume ℓ is between the lowest weight of T and the highest weight. For this procedure, we are looking for a vertex v of T which has weight ℓ. We want to remove v from T (and therefore also the weight ℓ from the list of weights), thereby creating a "smaller" tree $T - v$ satisfying the criteria above.

Deletion proceeds as a search does. However, in this case, you are searching for vertix $v \in V$ for which $\text{wt}(v) = \ell$. Once found, we remove v from V and any edge $(u, v) \in E$ is replaced by (u, w_1) and (u, w_2), where $w_1.w_2 \in V$ were the children of v in T.

Sorting

A binary search tree can be used to implement a simple but efficient sorting algorithm. Suppose we wish to sort a list of numbers $L = [\ell_1, \ell_2, \ldots, \ell_n]$. First, let $V = \{1, 2, \ldots, n\}$ be the vertices of a tree and weight vertex i with ℓ_i, for $1 \leq i \leq n$. In this case, we can traverse this tree in order of its weights, thereby building a BST recursively. This BST represents the sorting of the list L.

Chapter 4

Distance and Connectivity

4.1 Paths and distance

- distance and metrics
- distance matrix
- eccentricity, center, radius, diameter
- trees: distance, center, centroid
- distance in self-complementary graphs

4.2 Vertex and edge connectivity

- vertex-cut and cut-vertex
- cut-edge or bridge
- vertex and edge connectivity

Theorem 4.1. ***Menger's Theorem.*** *Let u and v be distinct, non-adjacent vertices in a graph 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.*

Theorem 4.2. ***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.*

4.3 Centrality of a vertex

- degree centrality
- betweenness centrality
- closeness centrality
- eigenvector centrality

4.4 Network reliability

- Whitney synthesis
- Tutte's synthesis of 3-connected graphs
- Harary graphs
- constructing an optimal k-connected n-vertex graph

Chapter 5

Optimal Graph Traversals

5.1 Eulerian graphs

- multigraphs and simple graphs
- Eulerian tours
- Eulerian trails

5.2 Hamiltonian graphs

- hamiltonian paths (or cycles)
- hamiltonian graphs

Theorem 5.1. ***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 5.2. ***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.*

5.3 The Chinese Postman Problem

See section 6.2 of Gross and Yellen [14].

- de Bruijn sequences
- de Bruijn digraphs
- constructing a $(2, n)$-de Bruijn sequence
- postman tours and optimal postman tours
- constructing an optimal postman tour

5.4 The Traveling Salesman Problem

See section 6.4 of Gross and Yellen [14], and section 35.2 of Cormen et al. [11].

- Gray codes and n-dimensional hypercubes
- the Traveling Salesman Problem (TSP)
- nearest neighbor heuristic for TSP
- some other heuristics for solving TSP

Chapter 6

Planar Graphs

See chapter 9 of Gross and Yellen [14].

6.1 Planarity and Euler's Formula

- planarity, non-planarity, planar and plane graphs
- crossing numbers

Theorem 6.1. *The complete bipartite graph $K_{3,n}$ is non-planar for $n \geq 3$.*

Theorem 6.2. ***Euler's Formula.*** *Let G be a connected plane graph having n vertices, e edges and f faces. Then $n - e + f = 2$.*

6.2 Kuratowski's Theorem

- Kuratowski graphs

The objective of this section is to prove the following theorem.

Theorem 6.3. ***Kuratowski's Theorem.*** *A graph is planar if and only if it contains no subgraph homeomorphic to K_5 or $K_{3,3}$.*

6.3 Planarity algorithms

- planarity testing for 2-connected graphs
- planarity testing algorithm of Hopcroft and Tarjan [15]
- planarity testing algorithm of Boyer and Myrvold [10]

Chapter 7

Graph Coloring

7.1 Vertex coloring

- vertex coloring
- chromatic numbers
- algorithm for sequential vertex coloring
- Brook's Theorem
- heuristics for vertex coloring

7.2 Edge coloring

- edge coloring
- edge chromatic numbers
- chromatic incidence
- algorithm for edge coloring by maximum matching
- algorithm for sequential edge coloring
- Vizing's Theorem

7.3 Applications of graph coloring

- assignment problems
- scheduling problems
- matching problems
- map coloring and the Four Color Problem

Chapter 8

Network Flows

See Jungnickel [17], and chapter 12 of Gross and Yellen [14].

8.1 Flows and cuts

- single source-single sink networks
- feasible networks
- maximum flow and minimum cut

8.2 Ford and Fulkerson's theorem

The objective of this section is to prove the following theorem.

Theorem 8.1. ***Maximum flow-minimum cut theorem.*** *For a given network, the value of a maximum flow is equal to the capacity of a minimum cut.*

8.3 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.

8.4 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 9

Random Graphs

See Bollobás [7].

9.1 Erdös-Rényi graphs

Describe the random graph model of Erdös and Rényi [13]. Algorithms for efficient generation of random networks; see Batagelj and Brandes [3].

9.2 Small-world networks

The small-world network model of Watts and Strogatz [24]. The economic small-world model of Latora and Marchiori [19]. See also Milgram [20], Newman [21], and Albert and Barabási [1].

9.3 Scale-free networks

The power-law degree distribution model of Barabási and Albert [2]. See also Newman [21], and Albert and Barabási [1].

9.4 Evolving networks

Preferential attachment models. See Newman [21], and Albert and Barabási [1].

Chapter 10

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.

10.1 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.

LP Formulation :

- Mimimize : 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

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.

corresponding patch :
`http://trac.sagemath.org/sage_trac/ticket/7529`

10.2 Traveling Salesman Problem

Given a graph G whose edges are weighted by a function $w : E(G) \to \mathbb{R}$, 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 10.1.

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 :

- Mimimize

$$\sum_{e \in E(G)} w(e) b_e$$

- 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 ?)

corresponding patch :
`http://trac.sagemath.org/sage_trac/ticket/7529`

10.3 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.

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)

corresponding patch :
`http://trac.sagemath.org/sage_trac/ticket/7476`

10.4 Steiner tree

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 $w : E(G) \to \mathbb{R}$ 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)

corresponding patch :
`http://trac.sagemath.org/sage_trac/ticket/8403`

10.5 Linear arboricity

The linear arboricity of a graph G is the least number k such that the edges of G can be partitionned 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)

10.6 Acyclic edge coloring

An edge coloring with k colors is said to be acyclic if it is proper (each color class is a matching – maximal degree 1), and if the union of the edges of any two color classes is acyclic. The corresponding LP is almost a copy of the previous one, except that we need to ensure that $\binom{k}{2}$ different classes are acyclic.

corresponding patch :
`http://trac.sagemath.org/sage_trac/ticket/8405`

10.7 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 ?)

corresponding patch : `http://trac.sagemath.org/sage_trac/ticket/8404`

Appendix A

GNU Free Documentation License

Version 1.3, 3 November 2008

`http://www.fsf.org`

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.

Bibliography

[1] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, 2002.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[3] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[4] R. A. Beezer. *A First Course in Linear Algebra*. Robert A. Beezer, University of Puget Sound, Tacoma, Washington, USA, 2009. `http://linear.ups.edu`.

[5] K. A. Berman and J. L. Paul. *Fundamentals of Sequential and Parallel Algorithms*. PWS Publishing Company, 1997.

[6] N. Biggs. *Codes: An Introduction to Information, Communication, and Cryptography*. Springer, 2009.

[7] B. Bollobás. *Random Graphs*. Cambridge University Press, 2nd edition, 2001.

[8] O. Borůvka. O jistém problému minimálním (about a certain minimal problem). *Práce mor. přírodověd. spol. v Brně III*, 3:37–58, 1926.

[9] O. Borůvka. Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí (contribution to the solution of a problem of economical construction of electrical networks). *Elektronický Obzor*, 15:153–154, 1926.

[10] J. M. Boyer and W. J. Myrvold. On the cutting edge: Simplified O(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(2):241–273, 2004.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[12] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[13] P. Erdös and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959.

[14] J. Gross and J. Yellen. *Graph Theory and Its Applications*. CRC Press, 1999.

[15] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[16] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[17] D. Jungnickel. *Graphs, Networks and Algorithms*. Springer, 3rd edition, 2008.

[18] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[19] V. Latora and M. Marchiori. Economic small-world behavior in weighted networks. *The European Physical Journal B*, 32(2):249–263, 2003.

[20] S. Milgram. The small world problem. *Psychology Today*, 2:60–67, 1967.

[21] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[22] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[23] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2nd edition, 2008. `http://www.shoup.net/ntb`.

[24] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

Index