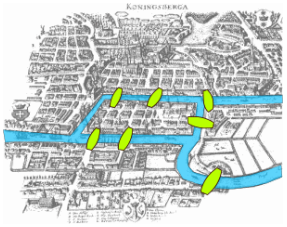
13 Basic Graph Algorithms

Graphs, also known as networks, offer a great way of expressing relationships between pairs of entities. As an abstraction, we are deliberately vague about what entities are because they are used to refer to many different sorts of things, depending on the application and context. Indeed, graphs are one of the most important and versatile abstractions in modeling and solving problems in computing.



The 7 bridges of Königsberg in Euler's time. https://upload.wikimedia.org/wikipedia/commons/5/5d/Konigsberg_bridges.png Bogdan Giuşcă CC BY-SA 3.0

An Ancient Problem. Königsberg was the name of a historic Prussian city. As the story has it, Königsberg, during the great mathematician Leonhard Euler's time, spanned both sides of the Pregel river and included two islands connected to each other and to the other portions of the "mainland" city via 7 bridges, as depicted in the map. The problem was, *how to walk through the city crossing each of the bridges exactly once?*

Readers are encouraged to spend a moment solving this puzzle before proceeding. Centuries ago, Leonhard Euler proved that the problem has no solution—it is impossible to come up with such a walk. The process of working it out has led to the development of what is known today as graph theory and this particular problem became known as the *Euler tour* problem.

To strip this down to its bare essence, we will redraw the map as follows: Each land mass is represented as a circle (later called a vertex or a node), and a bridge between land mass x and land mass y is represented by a line (later called an edge) connecting them. This is depicted below. For this problem, where the circles are drawn do not really matter; it is the interconnection between them that determines whether the problem has a solution.

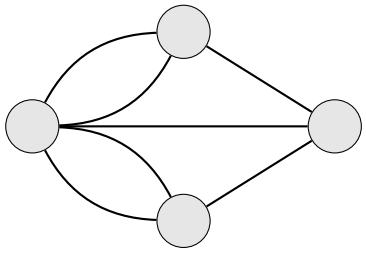


Figure 13.1: The 7 bridges of Königsberg abstracted as an undirected graph.

More generally, a graph is made up a set of vertices and connections between them. It can either be directed, with the directed edges (arcs) pointing

from one vertex to another, or undirected, with the edges symmetrically connecting vertices. Graphs can have weights or other values associated with either the vertices and/or the edges. Before discussing graphs more formally, we will touch on a short list of example graph applications.

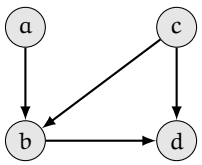
- **Road networks.** Vertices are intersections; edges are the road segments between them. Such networks are used by Google Maps and similar programs to help us navigate between locations. They are also used to study traffic patterns.
- **Dependence graphs.** Graphs are used to represent dependencies or precedences among items. For example, task A has to be completed before task B can begin. Such graphs are used as input into algorithms that minimize the total time or cost to completion.
- **Document link graphs.** Perhaps, the most prevalent example is the link graph of the web, where each web page is a vertex, and each (hyper)link, a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.
- **Social network graphs.** Graphs are used to represent, for example, who knows whom, who communicates with whom, and who influences whom. An example is who follows whom on Twitter, which can be used to determine how information flows. Other questions that can be answered in this way include how topics become hot, how communities develop, or even who might be a good match for who!

13.1 They Call Me Graphs

We will now lay out the foundation for studying graphs in the following sections. Let us start with directed graphs, which are useful for representing asymmetric relationships.

Definition 13.1 (Directed Graph)**.** A *directed graph* or a *digraph* is a pair $G = (V, A)$, where V is a set of *vertices* (or nodes), and $A \subseteq V \times V$ is a set of directed *edges* (or arcs).

A Directed Graph. It has 4 vertices and 4 arcs:



Under this definition, each arc is an ordered pair $e = (u, v) \in V \times V$ and a graph is allowed *self loops* (v, v)—meaning an arc coming out of v and going back into v.

Example. This graph is $G = (V, A)$, where

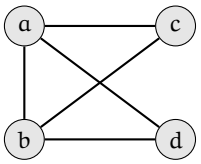
- $V = \{a, b, c, d\}$ and
- $A = \{(a, b), (c, b), (b, d), (c, d)\}$.

Next, let us define undirected graphs, which are used to represent symmetric relationships.

Definition 13.2. An *undirected* graph is a pair $G = (V, E)$, where V is a set of *vertices* (or nodes), and $E \subseteq \binom{V}{2}$ is a set of undirected *edges*.

Under this definition, each edge is a set of size 2, i.e., $e = \{u, v\}$. This means that the edge $\{u, v\}$ is the same as the edge $\{v, u\}$. In this notation, self loops are discouraged. Often, an undirected graph is represented by a directed graph by placing an arc in each direction for each undirected edge. In this sense, directed graphs are often seen as being more general.

An Undirected Graph. It has 4 vertices and 5 edges:



Example. This undirected graph example is $G = (V, E)$, where

- $V = \{a, b, c, d\}$ and
- $E = \Big\{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}\Big\}$.

Graph Terminology

Graphs come with many specialized terms, though most of them are intuitive. For now, we will discuss graphs without data or weights on the edges.

Neighbors and Neighborhood. A vertex u is a *neighbor* of (or *adjacent* to) a vertex v if there is an edges between them. For directed graphs, we distinguish between in- and out- neighbors of a vertex. For an undirected graph $G = (V, E)$, the *neighborhood* of G, denoted by $N_G(v)$ or simply $N(v)$, is the set of neighbors of v—that is to say,

$$N_G(v) = \big\{u \in V \mid \{u, v\} \in E\big\}.$$

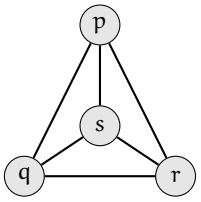
When $G = (V, A)$ is directed, we write $N^+(v)$ for the set of *out-neighbors* of v and $N^-(v)$ for the set of *in-neighbors* of v. The neighborhood of a set of vertices $U \subseteq V$ is given by $N(U) = \cup_{v \in U} N(v)$. We can similarly extend N^+ and N^-.

There is a specific term for the size of a neighborhood. The *degree* of a vertex v, denoted by $\deg(v)$, is given by $\deg(v) = |N(v)|$. For directed graphs, $\deg^+$ and $\deg^-$ can be similarly defined.

Paths, Cycles, and Reachability. A *path* in a graph is a sequence of adjacent vertices. That is to say, for a graph $G = (V, E)$, a path $p = p_0 p_2 \dots p_{|p|}$ is a sequence of vertices such that $p_i p_{i+1}$ is an edge in G for all $i = 0, 1, \dots, |p| - 1$. The *length* of a path, denote by $|p|$, is the number of edges on the path. A *simple path* is a path with no repeated vertices.

A vertex v is *reachable* from a vertex u in G if there is a path starting at u and ending at v. An undirected graph is *connected* if all vertices are reachable from all the other vertices.

A *cycle* is a path that starts and ends at the same vertex. A *simple cycle* is a cycle that has no repeated vertices other than the starting/ending vertex.



This undirected graph has $n = 4$ vertices and $m = 6$ edges. As an example, p is adjacent to q, s, and r. Hence, $N(p) = \{q, s, r\}$. Moreover, the path $spqr$ is a (simple) path in this graph. Also, the cycle $sqrs$ is a (simple) cycle in this graph.

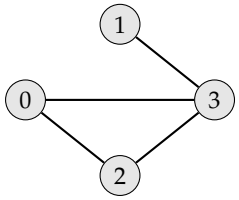
Size and Sparsity. We use $n = |V|$ and $m = |E|$. This means $m \leqslant \binom{n}{2} = n(n-1)/2$ for undirected graphs and $m \leqslant n^2$ for directed graphs. We say that a graph is *dense* if the number of edges is about the same order as the maximum number of edges (i.e., $m = \Theta(n^2)$) and otherwise *sparse*.

13.2 Graph Representation

How should graphs be kept in our computer programs? The choice of graph representation largely depends on the operations we intend to support. The following operations are common among standard graph algorithms:

- Find the degree (i.e., the number of neighbors) of a vertex.
- Find out if u and v are adjacent.
- Iterate over the neighbors of a given vertex.
- Iterate over all the edges of the graph.

To simplify matters, it is traditionally assumed that the vertices are numbered from 0 to $n-1$. This suggests the following representations:



Edge List. This representation simply stores a list of pairs (i, j) of edges. This can work for both directed and undirected graphs (with each edge doubling up). For example, the graph next to this paragraph is stored as

```
{ (0,2), (0,3), (1,3), (2,0), (2,3), (3,0), (3,1), (3,2) }
```

Adjacency Matrix. This representation stores a graph $G = (V, E)$ as an $n \times n$ matrix $A(G)$ of binary values, in which $A_{i,j}$ is 1 if and only if $(i, j) \in E$. For our running example, this gives

$$A(G) = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

It should be noted that for undirected graphs, the matrix is symmetric (i.e., $A^\top = A$) and the lack of self loops means that the diagonal is all 0s.

Adjacency Array. This representation keeps n arrays, where the i-th array stores the neighbors of vertex i. More precisely, if `a[]` is an array of length n, then `a[i]` is an array listing the neighbors of vertex i. For undirected graphs, it is customary to store each edge $\{u, v\}$ in both directions. For our running example, we have

```
int[][] a = {
    {2, 3},     // node 0's neighbor(s)
    {3},        // node 1's neighbor(s)
    {0, 3},     // node 2's neighbor(s)
    {0, 1, 2},  // node 3's neighbor(s)
}
```

All the classic representations have one or more of the following issues:

- We cannot easily change the graph (e.g., adding vertices/edges or removing them)
- We cannot easily work with meaningful vertex names (the vertices are stored as $0, 1, 2, \ldots$).
- Some common operations are inefficient.

Adjacency Map

Using richer data structures, we can develop a more versatile graph representation. The representation, which we will term the *adjacency map*, stores a `Map` from every vertex to the set of its neighbors. Hence, we aim to internally keep the graph as

```
Map<Vertex, Set<Vertex>> graph;
```

where `Vertex` is a generic type for how we wish the vertices to be.

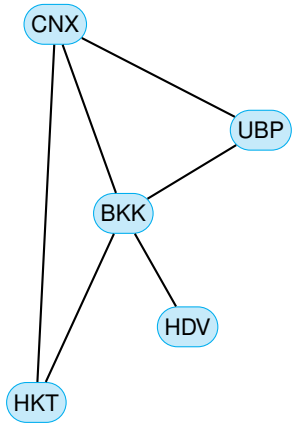
More concretely, our running graph example uses `Integer` vertices and would be stored as follows (abusing notation):

```
graph = {
    0: {2, 3},
    1: {3},
    2: {0, 3},
    3: {0, 1, 2}
}
```

This means that accessing the set of neighbors of any vertex is inexpensive; it is just one lookup, and once the corresponding set has been fetched, operations on it can be supported rather inexpensively. We can use both the `HashMap` and `TreeMap` for this, though the running time will be different.

It is a good idea to encapsulate the graph representation and the actions we do on it. To this end, we will work with the following interface, whose implementation is given as Exercise 13.1.

Often, the vertices are richer entities than simple numbers $0, 1, \ldots$. In this case, allowing an arbitrary vertex type (e.g., `String`) can be convenient.



Code 13.1: An interface to represent an undirected graph.

```
public interface UndirectedGraph<Vertex> {
    int numEdges(); /** How many edges? */
    int numVertices(); /** How many vertices? */
    int deg(Vertex v); /** Return the degree of v  */

    /** Return an iterable of vertices adjacent to v */
    Iterable<Vertex> adj(Vertex v);

    /** Is there an edge between u and v? */
    boolean isEdge(Vertex u, Vertex v);

    /** Add a new vertex */
    void addVertex(Vertex v);

    /** Add an edge between u and v */
    void addEdge(Vertex u, Vertex v);

    /** Remove an edge */
    void removeEdge(Vertex u, Vertex v);
}
```

13.3 Graph Traversal

Many questions about a graph can be answered by systematically walking in that graph. For example, which vertices can be reached from a vertex s? Is it true that a given graph is bipartite*? The process, known as graph traversal, typically starts from a source vertex or a set of source vertices. Then, it visits vertices that have not yet been explored until all the vertices have been seen or a target vertex has been reached. Various strategies have been proposed, mainly differing in how the new vertices are chosen. We will begin this section with a strategy known as breadth-first search.

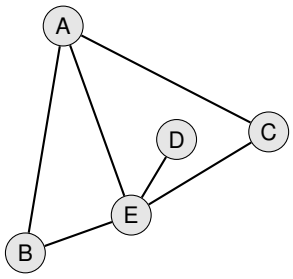
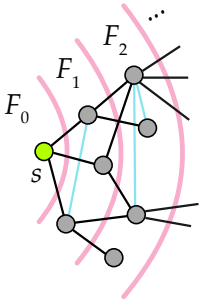


Figure 13.2: An example for graph traversal illustration.

With source vertex D, the layers of vertices visited are:

Layer #	Vertices
0	D
1	E
2	A, C, B
3	-



Breadth-first search (BFS) explores the graph in layers as if it is radiating from the source vertex in increasing order of "hop count." The frontiers are, by definition, disjoint, but vertices of the same frontier may be neighbors.

Breadth-First Search (BFS)

Breadth-first search (BFS) starts at a source vertex s, visits the neighbor of s, visits the neighbors of the neighbor of s, and so on. In other words, it explores vertices in layers:

- Layer 0 (starting) consists of the source vertex.
- Layer 1 is the neighbor of the vertex in Layer 0, excluding vertices seen in previous layers.
- Layer 2 is the neighbors of the vertices in Layer 1, excluding vertices seen in previous layers.
- This goes on until there are no more unvisited neighbors left to visit.

Notice that this discipline of exploration visits vertices in a breadth-first manner, hence the name breadth-first search.

To begin formalizing the process, we introduce the term the i-*th frontier*, denoted by F_i, for the set of vertices in layer i. In this notation, we can spell out the above idea as follows: First, the 0-th frontier consists of only the source vertex, hence $F_0 = \{s\}$. Each subsequent frontier is the neighbors of the preceding one, with vertices that have been seen before excluded. That is,

$$F_{i+1} = N(F_i) \setminus X_i,$$

where we define $X_i = \bigcup_{k=0}^{i} F_k$ to mean all the vertices that have been seen in layers at most i. Later, it will be more convenient to equivalently express X_i iteratively as $X_0 = F_0$ and $X_{i+1} = X_i \cup F_{i+1}$.

In this notation, the breadth-first search algorithm works as follows:

Algorithm 13.1: $\texttt{BFS}(G = (V, E), s)$ — Breadth-first search algorithm

```
F_0 ← {s}, X_0 ← {s}, i ← 0
while F_i ≠ ∅ do
    F_{i+1} ← N_G(F_i) \ X_i // Derive the next frontier of vertices
    X_{i+1} ← X_i ∪ F_{i+1} // Remember what has been seen
    i ← i + 1
return X_i
```

*A graph is bipartite if the vertices can be partitioned into two parts such that the endpoints of every edge span both parts.

An example is in order to illustrate a run of the algorithm.

Example. Consider the graph in Figure 13.2. If we start with the source vertex D, running the BFS algorithm results in have the following F_i's and X_i's:

i	F_i	X_i
0	{D}	{D}
1	{E}	{D, E}
2	{A, C, B}	{D, E, A, C, B}
3	$\emptyset$	{D, E, A, C, B}

Observations and Applications

From this setup, several properties are clear:

- The i-th frontier F_i is the set of vertices that are *exactly* at i hops away from the starting point s. By hops, we mean the shortest path in terms of edge count from the source requires that many edges.
- The final X_i contains all vertices reachable from s.

These properties directly mean breadth-first search can be used to help solve the following problems:

- find all the vertices reachable from a vertex v;
- find if an undirected graph is connected (i.e., all the vertices can reach each other);
- find the shortest path (in terms of edge count) from a source vertex s to all other reachable vertices; and
- check if a graph is bipartite (Exercise 13.8.).

Basic BFS Implementation

Implementing the basic breadth-first search algorithm is straightforward. Here, we will focus on implementing it for undirected graphs and will make use of the `UndirectedGraph` interface developed earlier in this chapter. We will keep the frontier sets and visited sets as Java's `Sets`, in particular the `HashSet` data structure.

We begin by describing how to derive the neighbors of a set of vertices, a computation that is central to the derivation of the next frontier. The code below takes as input a graph `G` and a set of vertices `F`, intended to be the current frontier set. It returns the union of the neighbors of the vertices of `F`—effectively the set $N(F)$.

```
Set<Vertex> nbrs(UndirectedGraph<Vertex> G, Set<Vertex> F) {
    Set<Vertex> nbrSet = new HashSet<>();
    for (Vertex src: F) {
        for (Vertex dst: G.adj(src)) { nbrSet.add(dst); }
    }
    return nbrSet;
}
```

Having written this utility function, the breadth-first search code itself is simple[†], mimicking the algorithm's description discussed earlier.

```
public Set<Vertex> bfs(UndirectedGraph<Vertex> G, Vertex s) {
    Set<Vertex> frontier = new HashSet<>(List.of(s));
    Set<Vertex> visited = new HashSet<>(List.of(s));

    while (!frontier.isEmpty()) {
        frontier = nbrs(G, frontier);
        frontier.removeAll(visited); // nbrs(frontier) - visited
        visited.addAll(frontier);
    }

    return visited;
}
```

Remarks: The view and implementation of breadth-first search in this book makes it clear that vertices are looked at layer by layer. Hence, the shortest edge-count property is self-evident. The overall running time of the algorithm, however, requires some analysis, which will be discussed soon. On the other hand, breadth-first search is more traditionally implemented using a queue: The queue stores vertices in the current and future frontiers. Until the queue becomes empty, the vertex at the front of the queue is removed and its unvisited neighbors are added to the queue. This discipline guarantees that the ordering in the queue respects the frontier ordering.

Running Time Analysis

How fast is the breadth-first search algorithm implemented above? We start by analyzing the running time of the `nbrs` function. Consider the `nbrs` function. If an adjacency map is used, `.adj` is an $O(1)$-operation. Moreover, each `.add` on the `HashSet` takes $O(1)$ time. Therefore, because the outer loop is run for each vertex v in `F`, the inner loop takes $O(\deg(v))$ time for that vertex `v` and the overall running time of a `nbrs` call is given by

$$O\left(\sum_{v \in F} 1 + \deg(v)\right),$$

where the 1 term is added to account for when that vertex has no neighbors but some constant work is still required to look at that vertex itself.

The total running time of BFS can now be analyzed as follows: Suppose the breadth-first search algorithm ran for `d` iterations, generating $F_0, F_1, \ldots, F_d$, where $F_d = \emptyset$. Observe that in each iteration, the running time of `.removeAll` and `.addAll` is at most the running time of `nbrs`. Hence, in terms of big-O, the

[†]Readers may be concerned about the running time of `removeAll`. It turns out `removeAll` on a `HashSet` takes time proportional to the size of the smaller set.

total running time is given by

$$\sum_{i=1}^{d} \sum_{v \in F_{i-1}} 1 + \deg(v).$$

By construction, the frontiers are disjoint, i.e., for all $i \neq j$, $F_i \cap F_j = \emptyset$. This means that each vertex $v \in V$ appears at most once in all the frontiers combined. Thus, this summation is upper-bounded by

$$\sum_{i=1}^{d} \sum_{v \in F_{i-1}} 1 + \deg(v) \leqslant \sum_{v \in V} 1 + \deg(v) = n + 2m,$$

where the last step used the common graph theoretic fact that $\sum_{v \in V} \deg(v) = 2m$ (Exercise 13.2.). Hence, we conclude that the running time of breadth-first search is bounded by $O(n + m)$.

Remarks: This running time is quite remarkable. The size of the graph with n vertices and m edges is approximately $n + m$. The fact that the total running time is $n + 2m = O(n + m)$ means that the algorithm merely looks at each edge at most twice and overall performs no more work than the footprint of the graph itself.

BFS: Remember the Way Back Home

The basic implementation we just discussed returns the set of vertices reachable from the source. Often, we would like to know more, for example, the distance of each vertex from the source vertex s or the actual shortest path from s to some vertex of interest v. It turns out to be easy to extend BFS for these purposes.

The main idea will be to remember as a vertex is entered into a frontier how that vertex is reached. To offer an example, consider the following graph and its frontiers. If we remember for each vertex v which vertex u enters it into the frontier, we will have the structure in Figure 13.4.

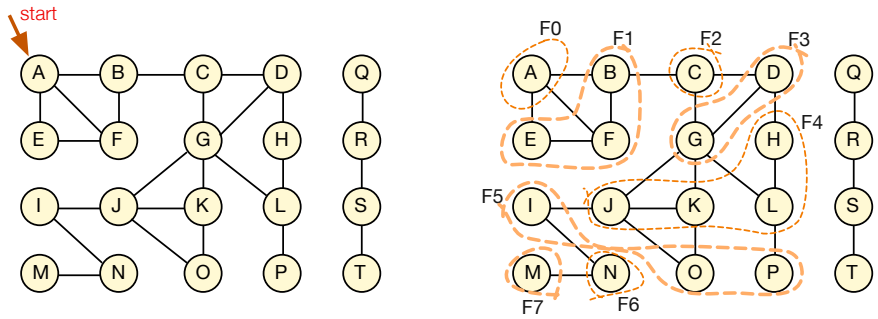


Figure 13.3: An undirected graph and its corresponding frontiers

The arrow $X \to Y$ means X is entered into the frontier by Y, i.e., the BFS exploration reaches X via Y.

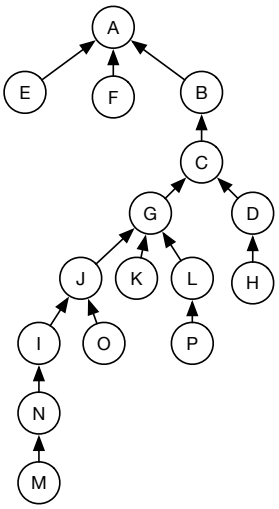


Figure 13.4: The tree structure corresponding to the BFS traversal of the graph in Figure 13.3

Such a structure is a tree; we call it the *breadth-first search tree* of a BFS traversal. With this tree, figuring out the shortest path to v is simply a matter of walking from v to s and reversing the path. Hence, the remaining question is, *how to update the code to store this tree?*

First, we will update the `nbrs` function to return a `Map` instead of a `Set`. This `Map` will remember, as has been discussed, for each vertex which vertex entered it into the neighbor set, like so:

```
Map<Vertex, Vertex> nbrs(UndirectedGraph<Vertex> G,
                         Set<Vertex> F) {
    Map<Vertex, Vertex> nbrSet = new HashMap<>();
    for (Vertex src: F) {
        for (Vertex dst: G.adj(src))
            nbrSet.put(dst, src);
    }
    return nbrSet;
}
```

This function is rather intricate and deserves further discussion. An example run is now in order.

Example. Suppose we run `nbrs` on the graph in Figure 13.3 with $F = \{G, D\}$. We will have the following returned:

```
{C: D, H: D, L: G, K: G, J: G}
```

Notice that C could have come from either D or G; either works fine.

We are now positioned to update the breadth-first search itself. The BFS tree will stored as a child-to-parent map inside `visited`. That is to say, `visited.get(u)` returns the parent in the BFS tree of `u`.

Code 13.2: Extended breadth-first search.

```
1  Map<Vertex, Vertex> bfs(UndirectedGraph<Vertex> G, Vertex s) {
2      Map<Vertex, Vertex> frontier = new HashMap<>();
3      Map<Vertex, Vertex> visited = new HashMap<>();
4
5      frontier.put(s, null); visited.put(s, null);
6
7      while (!frontier.isEmpty()) {
8          frontier = nbrs(G, frontier.keySet());
9          // nbrs(frontier) - visited
10         frontier.keySet().removeAll(visited.keySet());
11         visited.putAll(frontier);
12     }
13
14     return visited;
15 }
```

The running time stays the same. To deeply understand the implementation, readers are encouraged to step through this updated implementation

using the mesh graph in Figure 13.3. The bits where the new frontier is derived and how the visited map is updated are somewhat delicate. Below, we give the end result.

> **Example.** The updated BFS implementation when run on the graph in Figure 13.3 returns the map
>
> ```
> {A: null, E: A, F: A, B: A, C: B, G: C, D: C, H: D,
>  J: G, K: G, L: G, P: L, I: J, O: J, N: I, M: N}
> ```

Other Traversal Techniques

We have just discussed breadth-first search—a traversal strategy that explores vertices in a breadth-first manner. Other traversal strategies exist and are useful for answering other kinds of questions. We will briefly look at two other traversal techniques now.

Depth-First Search. Depth-first search (DFS) is an equally common traversal strategy. As the name suggests, this strategy focuses on going deep—i.e., it will go as deep as it can until it runs out of unvisited vertices, at which point it backs out until it reaches a vertex that has an unvisited neighbor, then traverses to that vertex and continues the same manner. More precisely, the following pseudocode shows how basic DFS works:

```
def dfs(G, v):
    if v has not been visited:
        mark v as visited
        for each vertex u in G.adj(v):
            dfs(G, u)
```

Depth-first search can be used to find all vertices reachable from a starting point (the same use case as BFS), to verify whether a graph is connected, and to generate a spanning tree. But this traversal strategy does not give the shortest path. However, because it will not back out until it has exhausted all reachable vertices, it is useful in many other applications such as cycle detection[‡] and topological sorting[§] These standard graph problems, however, are beyond the scope of this book.

Example. In contrast to breadth-first search, the depth-first search traversal of the graph in Figure 13.2 starting at the same vertex D will be D, E, B, A, C (among many possible traversals).

> **Tips**
>
> In real code, DFS is implemented as a recursive program or through the help of a stack data structure.

Priority-First Search. Another common traversal strategy is to pick the next vertex to visit based on priority. That is to say, each vertex is given a priority, which may change over time, and the next vertex to visit is the highest priority vertex that has not been visited so far. This strategy is useful for

[‡]Given a graph, directed or undirected, is there a cycle?

[§]Given a directed, acyclic graph (DAG), find an ordering of the graph's vertices $v_1, v_2, \ldots, v_n$ such that if there is an edge from v_i to v_j, then v_j comes after v_i in the ordering.

finding shortest paths in cases where the cost on each edge may be different and one seeks to minimize the total cost (e.g., distance traveled).

13.4 Minimum Spanning Trees

How many edges have to be retained to connect up an undirected graph with n vertices? For motivation, consider a graph with 6 vertices and 7 edges:

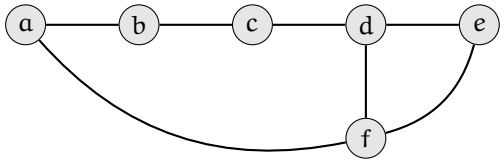


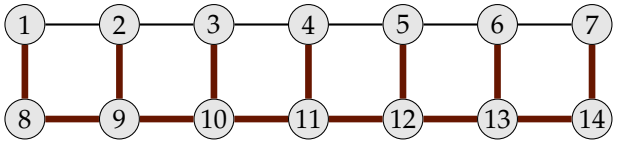
Figure 13.5: An undirected graph: what is the fewest number of edges necessary to connect up all these vertices?

What's a subgraph? A *subgraph* of a graph G is a graph forming by taking a subset of the vertices and edges of G. Hence, the subgraph has some or all of the vertices and edges of the original graph.

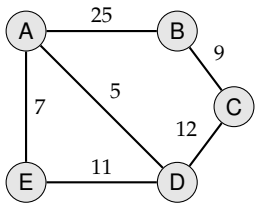
A moment's thought suggests that $n-1$ edges are necessary for an n-vertex graph to be connected. Indeed, it has been shown that if a graph is a connected graph, the structure that has the fewest edges which can connect up all the vertices is a spanning tree. As the name suggests, a spanning tree is a tree on all the vertices—i.e., it spans the graph, as in it stretches out to all these vertices and connect them up.

Definition 13.3 (Spanning Tree)**.** A *spanning tree* T of an undirected graph $G = (V, E)$ is a subgraph of G such that T is a tree that includes all vertices of V.

Example. The grid graph below is made up of 10 vertices and 13 edges. There are many possible spanning trees on this graph. One possible spanning tree is shown in thick edges.



Notice that this spanning tree is *not* the only one. There are many other possible spanning trees.



A Weighted Graph.

Weighted Graphs and Representation

Our graphs so far are made up simply of vertices and edges connecting them. They could be directed or undirected. At some level, all the edges are equal. In many applications, though, we may wish to annotate each of these edges with a label or sometimes a number representing, e.g., its strength or its cost. To add such annotations to a graph, we can just graphically add these labels next to the edges.

Mathematically, we add these annotations as a side function. Because these annotations are often weights, we call such graphs weighted graphs:

> **Definition 13.4.** A *weight graph*, denoted by $G = (V, E, w)$, is a graph G, together with a weight function $w : E \to L$, where L is the set of possible annotations.

Often, the set L is simply $\mathbb{R}_+$ (the positive reals). In this view, for every edge e, $w(e)$ gives the label (i.e., weight) on that edge.

Representation in Code. One possibility is to strictly follow the mathematical definition and store on the side a `Map` from every edge to its label/weight. However, we most often wish to access the label/weight on an edge at the same time we access that edge. This motivates a representation that bundles the label/weight with the associated edge. To accomplish this, instead of keeping a `Map<Vertex, Set<Vertex>>` like we did for the unweighted case, we will keep a map `Map<Vertex, Map<Vertex, Label>>` from each vertex to a map storing for each neighbor the label/weight associated with that edge. Specifically, if `G` is such a map, then `G.get(u)` returns a `Map<Vertex, Label>`, where the keys are the neighbors of `u` and the value for the key `v` will be the labels/weights corresponding to the edge `uv`.

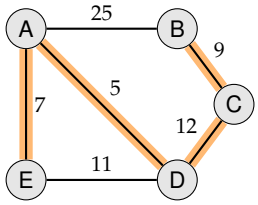
An example is in order:

> **Example.** The weighted graph example on the previous page has vertices named A, B, ... Their labels/weights are integers. This can be stored as a
>
> ```
> Map<String, Map<String, Integer>>
> ```
>
> which reflects the fact that each vertex is referred to by a `String`. Hence, the graph is stored as follows (we are rendering the map in Python's dictionary notation for compactness):
>
> ```
> G = {
>     "A": {"B": 25, "D": 5, "E": 7},
>     "B": {"A": 25, "C": 9},
>     "C": {"B": 9, "D": 12},
>     "D": {"A": 5, "C": 12, "E": 11},
>     "E": {"A": 7, "D": 11}
> }
> ```
>
> Notice that for instance, the entry corresponding to `"D"` is `{"A": 5, "C": 12, "E": 11}`, which indicates that `"D"` has neighbors `A`, `C`, and `E`. Moreover, as an example, the edge `DC` has weight 12.

An Interface for Weighted Undirected Graph. Small modifications to the `UndirectedGraph` will make our previously-discussed interface ready to take on labels/weights. Specifically, we will:

- make `adj` return a `Map<Vertex, Label>` instead of an `Iterable<Vertex>`.
- add a method `Label getWeight(Vertex u, Vertex v)`.

The Minimum Spanning Tree Problem



The highlighted edges form a spanning tree with weight $w(T) = 7+5+12+9 = 33$, which happens to be the minimum possible cost. All other spanning trees, as can be checked, have a higher cost.

We have just discussed how to model a graph so that the edges can carry information. In practice, various kinds of information are stored on the edges. For example, in a road network, the edge weights could represent the distance or fuel cost to drive along them. In a power distribution graph, the edges could represent transmission lines and the weights could represent the lengths or how much power is needed.

Earlier, we saw that $n-1$ edges are necessary to connect up an undirected graph with n nodes. If edge weights represent how costly selecting the edges is, then different spanning trees do not necessary have the same cost—some less expensive than others. We wish to find one that is the least costly.

The *minimum spanning tree* (MST) problem is to find a spanning tree whose total weight is the smallest possible among all spanning trees on the given graph. More formally, the minimum (weight) spanning tree (MST) problem is, given an connected undirected graph $G = (V, E)$, where each edge e has weight $w(e) \geqslant 0$, find a spanning tree of minimum weight (i.e., the sum of the weights of the edges). That is to say, we are interested in finding the spanning tree T that minimizes

$$w(T) = \sum_{e \in E(T)} w(e).$$

It is clear that for any given connected graph, there is a spanning tree that has the lowest cost. But how many MSTs are there? The following lemma makes our lives easy when the weights are distinct:

Lemma 13.5. There is a unique minimum spanning tree of G provided that G is connected and has distinct weights.

We leave the proof of this lemma as Exercise 13.3. We note, however, that even though there may be duplicate weights, we could break ties in a consistent way and make them distinct for the purpose of MST. Hence, in the rest of this chapter, we will assume that the weight edges are distinct[¶].

An Underlying Principle: Light-Edge Rule

How can we find the minimum spanning tree of a graph quickly? The main property that underlines many MST algorithms is a simple fact about cuts in a graph. To begin, we'll make a few observations about a tree: If T is a tree,

- adding an edge between vertices of T creates a cycle.
- removing an edge from T breaks it into exactly two trees.

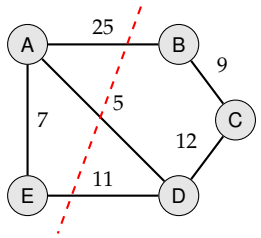
(In some sense, a tree is the minimally connected graph on this set of vertices.)

What is a cut? For a graph $G = (V, E)$, a *cut* is defined in terms of a proper subset $U \subset V$, where $U \neq \emptyset$ and $U \neq V$. This set U partitions the graph into $(U, V \setminus U)$, and we refer to the edges between the two parts as the cut edges $E(U, \overline{U})$, where as is typical in literature, we write $\overline{U} = V \setminus U$. The subset U

[¶]This is not a requirement for the implementation in practice, but it simplifies the exposition.

might include a single vertex v, in which case the cut edges would be all edges incident on v. But the subset U must be a proper subset of V (i.e., $U \neq \emptyset$ and $U \neq V$). Hence, both sides of the cut are nonempty.

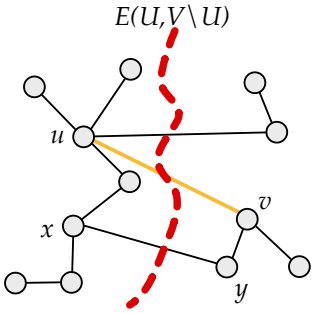
Example. Below, an example cut is $U = \{A, E\}$, which partitions the graph into two portions as indicated by the dashed line. The edges that go across the cut $E(U, \overline{U})$ are AB, AD, and ED.



Light-Edge Rule. Given any cut in a graph G, the following theorem states that the lightest edge across the cut is in the MST of G:

Theorem 13.6 (Light-Edge Rule)**.** Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any nonempty proper subset $U \subseteq V$, the minimum weight edge e between U and $V \setminus U$ is in the minimum spanning tree MST(G) of G.

Proof. The proof is by contradiction. Let a cut $(U, V \setminus U)$ be given. We will denote by $e = \{u, v\}$ the lightest edge going across this cut. Now consider the minimum spanning tree (MST) T of G. Suppose for a contradiction that e is not in the MST T. Since the MST spans the graph, there must be a path P between u and v using just the edges of the MST. This path must cross the cut between U and $V \setminus U$ at least once since u and v are on the opposite sides. Call an edge on P that crosses the cut $f = \{x, y\}$. By attaching e to T and removing f, we form a new spanning tree $T' = T + e - f$. Notice that T' is a spanning tree because there are still $n - 1$ edges and all the vertices are still connected: any path that went through f can now go through e instead. But importantly, $w(T') = w(T) + w(e) - w(f)$, and since $w(e) < w(f)$, we have that $w(T') < w(T)$, which is a contradiction as we assumed T is an MST. Hence, we conclude that e—the minimum-weighted edge across the cut $(U, V \setminus U)$—must appear in the MST. □



From a given tree T, form T' by removing $f = \{x, y\}$ and adding $e = \{u, v\}$. The argument in the proof shows that $w(T')$ is smaller than the original weight $w(T)$.

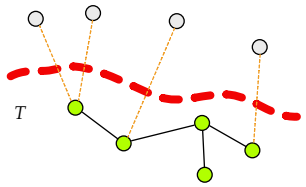
Prim's Algorithm

We apply the light-edge rule to derive an efficient algorithm for MST known in the literature as *Prim's algorithm*. The idea is to maintain a single, connected tree and keep growing it one edge at a time using the light-edge rule until we finally have the MST. Algorithm 13.2 shows this idea in more detail.

Correctness of this algorithm follows immediately from the light-edge rule. Notice that by construction, it is an invariant that at the end of iteration i, T

Algorithm 13.2: `Prim`$(G = (V, E, w))$ — Prim's algorithm for MST

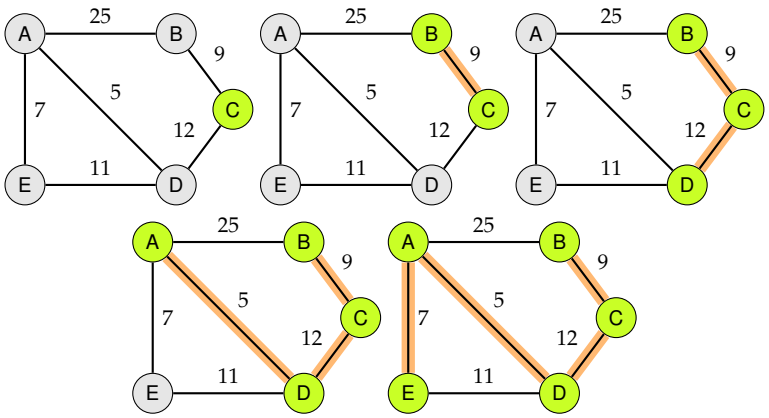
Pick a starting vertex $s \in V$ arbitrarily
Let $U_0 \leftarrow \{s\}$, $T \leftarrow \{\}$
for $i = 1, 2, \ldots, n-1$ **do**
  Apply the light-edge rule on the cut $(U_{i-1}, V \setminus U_{i-1})$
  Let $e_i = (x, y)$ be the minimum-weighted edge across the cut
  Add e_i to T
  Set $U_i \leftarrow U_{i-1} \cup \{x, y\}$ // *Either* x *or* y *was already in* U_{i-1}
return T



Prim's algorithm maintains a connected spanning tree T and in each step, it locates the lightest edge across the cut (dashed line) and adds it to T.

is connected and is a spanning tree on the vertex set U_i. Furthermore, there cannot be any cycle because each e_i joins U_{i-1} with a vertex outside of U_{i-1}, extending it to U_i.

Example. Suppose vertex C is used as the starting point, so $U_0 = \{C\}$. The lightest edge across the cut $(U_0, \overline{U_0})$ is BC, with weight 9. Therefore, $U_1 = \{B, C\}$. Then, the lightest edge across the cut $(U_1, \overline{U_1})$ is 12, so $U_2 = \{B, C, D\}$. Prim's algorithm continues in this way until the minimum spanning tree on this graph is formed.



But in real code, how can one find the minimum-weighted edge across the cut efficiently? Code 13.3 shows an implementation in Java. The tree vertices (U_i) are maintained in the set `treeVertices`, initially containing just the arbitrary starting vertex $\{s\}$. To quickly find the minimum-weighted edge across the cut, we use a priority queue. Ideally, we would like the priority queue to store exactly the edges across the current cut. However, the cut changes with every new edge added to the MST and it is too expensive to keep the priority queue up-to-date with the current cut. Therefore, we settle with the invariant that the priority queue contains two types of edges:

- Edges that are crossing the current cut; and
- Edges that used to cross a previous cut but are currently internal to tree side (i.e., both endpoints are in U_i).

For this reason, when the minimum-weighted edge is removed from the priority queue, we have to check whether it is still crossing the current cut. It is easy to see that if this edge is indeed crossing the current cut, the edge is

the minimum-weighted edge crossing the current cut.

Then, when a new edge e is added to the tree T, the endpoint that is new to the tree will be "expanded on." The edges coming out from this vertex will be new to the current cut.

Code 13.3: Prim's algorithm for minimum spanning tree.

```
List<Edge> primMST(WeightedUndirectedGraph<Vertex, Weight> G) {
    PriorityQueue<Edge> pq = new PriorityQueue<Edge>(
         (Edge x, Edge y) -> x.cost.compareTo(y.cost)
      );
    List<Edge> T = new ArrayList<>();
    Set<Vertex> treeVertices = new HashSet<>();
    expandNeighbors(0, G, pq, treeVertices);
    while (!pq.isEmpty()) {
      Edge e = pq.poll();
      if (treeVertices.contains(e.u) &&
          treeVertices.contains(e.v))
          continue; // e is internal, skip
      T.add(e); // e is the lightest edge across the cut

      // Determine which endpoint is new to T
      if (!treeVertices.contains(e.u))
          expandNeighbors(e.u, G, pq, treeVertices);
      else
          expandNeighbors(e.v, G, pq, treeVertices);
    }
    return T;
}

void expandNeighbors(Vertex vtx,
        WeightedUndirectedGraph<Vertex, Weight> G,
        PriorityQueue<Edge> pq, Set<Vertex> treeVertices) {
     treeVertices.add(vtx);
     for (Vertex w : G.adj(vtx)) {
       // put into the queue the edge (vtx, w) and its weight
       if (!treeVertices.contains(w))
           pq.add(G.getEdge(vtx, w));
     }
  }
```

Running Time. What's the running time of this code? We reason as follows: Each vertex can be an argument to `expandNeighbors` at most once. Each time it is called with v, the cost is $O(\deg(v) \log n)$ as it goes over the neighbors of v, adding each of the $O(\deg(v))$ neighbors to the priority queue. Also, the number of entries added to the priority queue by this vertex is $\deg(v)$. Hence, the total cost due to the `expandNeighbors` function is $\sum_{v \in V} \deg(v) \log n =$

$O(m \log n)$. Notice also that the number of times the priority queue is added to is at most $\sum_{v \in V} \deg(v) = 2m$.

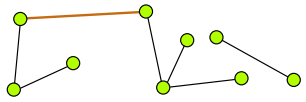
The total time of the `while`-loop can be bounded as follows: We do not know how many times exactly that loop will run. But we do know that (i) each time it is run, it takes at most $O(\log n)$ time, excluding the time taken by `expandNeighbors`. This $O(\log n)$ is due to the `.poll` call. Furthermore, we know that (ii) the priority queue cannot be `.poll`-ed more than than the number times it was added to. We have argued already that this number is at most $2m$. Hence, the total time of the `while`-loop is $O(m \log n)$, plus the time taken by `expandNeighbors`. In all, we conclude that Prim's algorithm runs in $O(m \log n)$ time.

Kruskal's Algorithm

Another MST algorithm that can be derived using the light-edge rule is *Kruskal's algorithm*. Unlike Prim's algorithm, which maintains a single tree and keeps on growing it, Kruskal's algorithm maintains a forest (multiple disjoint trees) and incrementally connects up the pieces to eventually form the MST by considering edges in increasing order of their weights (i.e., light to heavy). Initially, the forest contains n trees, each a singleton tree. In more detail, Kruskal's algorithm works as follows:

Algorithm 13.3: `Kruskal`$(G = (V, E, w))$ — Kruskal's algorithm

```
T ← {}
Sort the edges so w(e_1) < w(e_2) < ··· < w(e_m), where e_i's are the
  edges of G
for i = 1, 2, ..., m do
    Let {x, y} ← e_i (i.e., x and y are the endpoints of e_i)
    if x cannot reach y in T then
        Add e_i to T // Add to the MST
return T
```

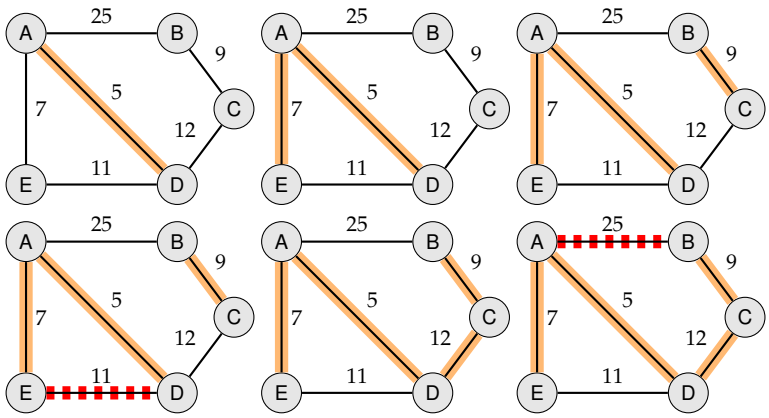


Kruskal's algorithm maintains a spanning forest T (multiple trees) and in each step, it locates the lightest edge that will not form a cycle. Adding this to T combines two trees.

The algorithm is intuitive: To create the least expensive spanning tree, we should aim to use the cheapest edges unless an edge cannot be used (e.g., form a cycle). But mathematically, why does it result in the minimum spanning tree? The light-edge rule we proved earlier has an answer. When the algorithm considers $e_i = \{x, y\}$, if e_i does not form a cycle with existing MST edges (i.e., x cannot reach y using existing edges of T), we know that it crosses a cut described as follows: Let U_x be all the vertices reachable from x using the edges of T so far. The cut of interest is the cut $(U_x, V \setminus U_x)$. Notice that e_i is the minimum-weighted edge crossing this cut since after all the edges have been sorted and we consider them from small to large. Hence, every edge added by Kruskal's algorithm is part of the MST.

Example. The cheapest edge is AD with weight 5. Initially T is empty, so A cannot reach D using the edges of T, so AD is added to T. The next cheapest edges are AE and BC. For the same reason, they are added to T, respectively. After that, the next edge to consider is ED, but E can already reach D using the edges of T, so this edge is not added to T. We

continue in this manner until the whole MST is formed.



But how should we implement Kruskal's algorithm in real code? It is clear that the efficiency of the algorithm hinges on the fact that we can determine quickly whether x can reach y using the edges of T. There is an elegant data structure that can efficiently handle this task. Chapter 14 presents data structures to represent disjoint sets, allowing the sets to be united and the checking of whether two members belong in the same set. In this abstraction, the members are vertices and two vertices are in the same set if they can reach each other using the edges of T. Hence, adding an edge $\{x, y\}$ has the effect of uniting the set containing x with the set containing y, and asking whether x can reach y is the same as asking whether x and y belong in the same set. We can update the algorithm's description as follows:

Algorithm 13.4: `Kruskal`$(G = (V, E, w))$ — Kruskal's algorithm with disjoint sets

$T \leftarrow \{\}$
Sort the edges so $w(e_1) < w(e_2) < \cdots < w(e_m)$, where e_i's are the edges of G
$D \leftarrow$ disjoint-sets data structure with each vertex of V in its own set
for $i = 1, 2, \ldots, m$ **do**
    Let $\{x, y\} \leftarrow e_i$ (i.e., x and y are the endpoints of e_i)
    if `D.isConnected`(x, y) **then**
        Add e_i to T *// Add to the MST*
        `D.link`(x, y)
return T

Both `isConnected` and `link` can be supported in $O(\log n)$ time[||]. Therefore, Kruskal's algorithm can be implemented by first sorting the edges (expending $O(m \log n)$ time). After that, The `for`-loop is run m times, each taking $O(\log n)$ time. Hence, the total running time is $O(m \log n)$. Readers may also notice that the algorithm can stop as soon as T has accumulated $n - 1$ edges. There is no need to continue past that point.

[||] Faster data structures exist but are not covered in this book.

Exercises

Exercise 13.1. Using the adjacency map representation discussed earlier in the chapter, write a class `UndirectedAdjMap<Vertex>` that implements the `UndirectedGraph<Vertex>` interface from Code 13.1. Use a `HashMap` for the map implementation.

Exercise 13.2. Show that for an undirected graph $G = (V, E)$ with no self-loops, the sum of the degrees is equal to twice the number of edges:

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

Exercise 13.3. Let $G = (V, E, w)$ be an undirected connected graph with distinct edge weights. Show that G has a unique minimum spanning tree.

Exercise 13.4. You will extend to the breadth-first search implementation discussed earlier in this chapter to return the number of edges on the shortest path from the source to a given vertex. Specifically, given an undirected graph and a source vertex `s`, write a method

```
Map<Integer, Integer> distTo(UndirectedGraph<Integer> G,
                             Integer s)
```

that returns a `Map m` such that for each vertex `v`, `m.get(v)` returns the number of edges on the shortest path from `s` to `v`.

Exercise 13.5. You will extend to the breadth-first search implementation discussed earlier in this chapter to report an actual shortest path. Specifically, given an undirected graph and a pair of nodes, write a method

```
void findShortest(UndirectedGraph<Integer> G,
                  Integer a, Integer b)
```

that prints out the shortest path from vertex `a` to vertex `b`.

Exercise 13.6. Given a connected undirected graph, design an $O(m + n)$-time algorithm that computes a spanning tree (necessarily a minimum-weighted spanning tree) of the graph. Notice that the algorithms we studied for MST are not fast enough here.

Exercise 13.7. Given a connected undirected graph, design an $O(m + n)$-time algorithm to identify a vertex whose removal (removing that vertex and its incident edges) does not disconnect the graph.

Exercise 13.8. Design and implement an $O(m + n)$-time algorithm to detect whether or not a given undirected graph is bipartite. As was mentioned earlier, a graph is bipartite if the vertices can be partitioned into two parts

(say red and blue) such that every edge has one endpoint in red and the other in blue.

Exercise 13.9. Develop a complete implementation of Prim's algorithm based on Code 13.3 that runs in $O(m \log n)$ time.

Exercise 13.10. Develop an implementation of Kruskal's algorithm that runs in $O(m \log n)$ time. It is helpful to study the disjoint-sets data structure chapter (Chapter 14) before completing this exercise.

Exercise 13.11. Develop an algorithm that runs in $O(m \log n + n)$ or faster to determine whether a given graph has a unique minimum spanning tree.

Chapter Notes

Graph theory and graph algorithms have been extensively studied with a vast literature behind them. For more exposition of mathematical graph theory, check out the books by Lehman et al. [LLM15], Matoušek and Nešetřil [MN98], and Diestel [Die12]. Easley and Kleinberg [EK10] explore networks, crowds, and markets through graph-theoretic views. Excellent references on graph representations, traversal techniques (e.g., breadth-first search, depth-first search, etc.), and minimum-spanning tree algorithms (e.g., Prim's and Kruskal's) include Cormen et al. [Cor+09], Sedgewick and Wayne [SW11], and Goodrich et al. [GTG14]. For minimum spanning trees, Borůvka's algorithm is probably the oldest published algorithm and it happens to be amendable to parallel computation.