Chapter 12

Graphs

In this chapter, we study two representations of graphs and basic algorithms that use these representations.

Mathematically, a *(directed) graph* is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of ordered pairs of vertices called *edges*. An edge (i, j) is *directed* from i to j; i is called the *source* of the edge and j is called the *target*. A *path* in G is a sequence of vertices $v_0, \ldots, v_k$ such that, for every $i \in \{1, \ldots, k\}$, the edge (v_{i-1}, v_i) is in E. A path $v_0, \ldots, v_k$ is a *cycle* if, additionally, the edge (v_k, v_0) is in E. A path (or cycle) is *simple* if all of its vertices are unique. If there is a path from some vertex v_i to some vertex v_j then we say that v_j is *reachable* from v_i. An example of a graph is shown in Figure 12.1.

Due to their ability to model so many phenomena, graphs have an enormous number of applications. There are many obvious examples. Computer networks can be modelled as graphs, with vertices corresponding to computers and edges corresponding to (directed) communication links between those computers. City streets can be modelled as graphs, with vertices representing intersections and edges representing streets joining consecutive intersections.

Less obvious examples occur as soon as we realize that graphs can model any pairwise relationships within a set. For example, in a university setting we might have a timetable *conflict graph* whose vertices represent courses offered in the university and in which the edge (i, j) is present if and only if there is at least one student that is taking both class i and class j. Thus, an edge indicates that the exam for class i should not

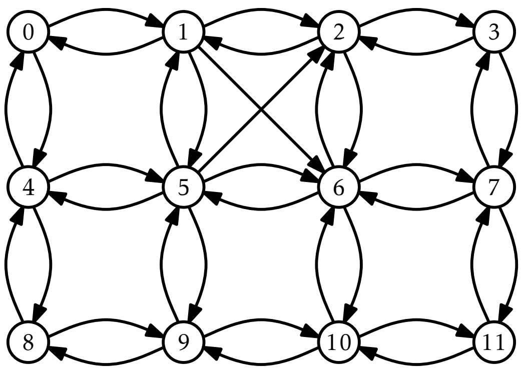


Figure 12.1: A graph with twelve vertices. Vertices are drawn as numbered circles and edges are drawn as pointed curves pointing from source to target.

be scheduled at the same time as the exam for class j.

Throughout this section, we will use n to denote the number of vertices of G and m to denote the number of edges of G. That is, $n = |V|$ and $m = |E|$. Furthermore, we will assume that $V = \{0, \ldots, n-1\}$. Any other data that we would like to associate with the elements of V can be stored in an array of length n.

Some typical operations performed on graphs are:

- addEdge(i, j): Add the edge (i, j) to E.
- removeEdge(i, j): Remove the edge (i, j) from E.
- hasEdge(i, j): Check if the edge $(i, j) \in E$
- outEdges(i): Return a List of all integers j such that $(i, j) \in E$
- inEdges(i): Return a List of all integers j such that $(j, i) \in E$

Note that these operations are not terribly difficult to implement efficiently. For example, the first three operations can be implemented directly by using a USet, so they can be implemented in constant expected time using the hash tables discussed in Chapter 5. The last two operations can be implemented in constant time by storing, for each vertex, a list of its adjacent vertices.

However, different applications of graphs have different performance requirements for these operations and, ideally, we can use the simplest implementation that satisfies all the application's requirements. For this reason, we discuss two broad categories of graph representations.

12.1 AdjacencyMatrix: Representing a Graph by a Matrix

An *adjacency matrix* is a way of representing an n vertex graph $G = (V, E)$ by an $n \times n$ matrix, a, whose entries are boolean values.

AdjacencyMatrix
```
int n;
boolean[][] a;
AdjacencyMatrix(int n0) {
  n = n0;
  a = new boolean[n][n];
}
```

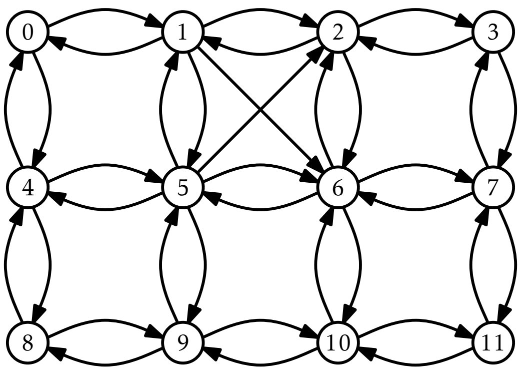
The matrix entry a[i][j] is defined as

$$a[i][j] = \begin{cases} \texttt{true} & \text{if } (i, j) \in E \\ \texttt{false} & \text{otherwise} \end{cases}$$

The adjacency matrix for the graph in Figure 12.1 is shown in Figure 12.2.

In this representation, the operations addEdge(i, j), removeEdge(i, j), and hasEdge(i, j) just involve setting or reading the matrix entry a[i][j]:

AdjacencyMatrix
```
void addEdge(int i, int j) {
  a[i][j] = true;
}
void removeEdge(int i, int j) {
  a[i][j] = false;
}
boolean hasEdge(int i, int j) {
  return a[i][j];
}
```



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

Figure 12.2: A graph and its adjacency matrix.

These operations clearly take constant time per operation.

Where the adjacency matrix performs poorly is with the outEdges(i) and inEdges(i) operations. To implement these, we must scan all n entries in the corresponding row or column of a and gather up all the indices, j, where a[i][j], respectively a[j][i], is true.

```
                         AdjacencyMatrix
List<Integer> outEdges(int i) {
  List<Integer> edges = new ArrayList<Integer>();
  for (int j = 0; j < n; j++)
    if (a[i][j]) edges.add(j);
  return edges;
}
List<Integer> inEdges(int i) {
  List<Integer> edges = new ArrayList<Integer>();
  for (int j = 0; j < n; j++)
    if (a[j][i]) edges.add(j);
  return edges;
}
```

These operations clearly take $O(n)$ time per operation.

Another drawback of the adjacency matrix representation is that it is large. It stores an $n \times n$ boolean matrix, so it requires at least n^2 bits of memory. The implementation here uses a matrix of boolean values so it actually uses on the order of n^2 bytes of memory. A more careful implementation, which packs w boolean values into each word of memory, could reduce this space usage to $O(n^2/w)$ words of memory.

Theorem 12.1. *The AdjacencyMatrix data structure implements the Graph interface. An AdjacencyMatrix supports the operations*

- addEdge(i, j), removeEdge(i, j), *and* hasEdge(i, j) *in constant time per operation; and*
- inEdges(i), *and* outEdges(i) *in* $O(n)$ *time per operation.*

The space used by an AdjacencyMatrix is $O(n^2)$.

Despite its high memory requirements and poor performance of the inEdges(i) and outEdges(i) operations, an AdjacencyMatrix can still be useful for some applications. In particular, when the graph G is *dense*, i.e., it has close to n^2 edges, then a memory usage of n^2 may be acceptable.

The AdjacencyMatrix data structure is also commonly used because algebraic operations on the matrix a can be used to efficiently compute properties of the graph G. This is a topic for a course on algorithms, but we point out one such property here: If we treat the entries of a as integers (1 for true and 0 for false) and multiply a by itself using matrix multiplication then we get the matrix a^2. Recall, from the definition of matrix multiplication, that

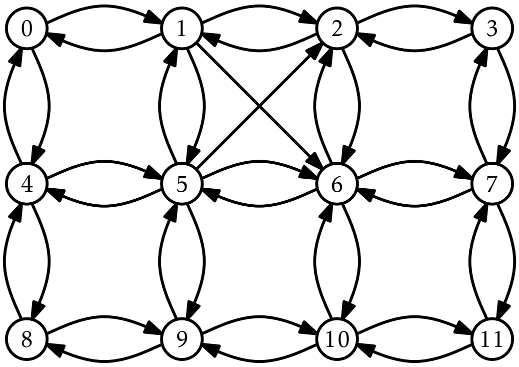
$$a^2[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot a[k][j] \ .$$

Interpreting this sum in terms of the graph G, this formula counts the number of vertices, k, such that G contains both edges (i,k) and (k, j). That is, it counts the number of paths from i to j (through intermediate vertices, k) whose length is exactly two. This observation is the foundation of an algorithm that computes the shortest paths between all pairs of vertices in G using only $O(\log n)$ matrix multiplications.

12.2 AdjacencyLists: A Graph as a Collection of Lists

Adjacency list representations of graphs take a more vertex-centric approach. There are many possible implementations of adjacency lists. In this section, we present a simple one. At the end of the section, we discuss different possibilities. In an adjacency list representation, the graph $G = (V,E)$ is represented as an array, adj, of lists. The list adj[i] contains a list of all the vertices adjacent to vertex i. That is, it contains every index j such that $(i, j) \in E$.

AdjacencyLists
```
int n;
List<Integer>[] adj;
AdjacencyLists(int n0) {
  n = n0;
  adj = (List<Integer>[])new List[n];
  for (int i = 0; i < n; i++)
```



0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	0	1	5	6	4	8	9	10
4	2	3	7	5	2	2	3	9	5	6	7
	6	6		8	6	7	11		10	11	
	5				9	10					
					4						

Figure 12.3: A graph and its adjacency lists

```
    adj[i] = new ArrayStack<Integer>();
}
```

(An example is shown in Figure 12.3.) In this particular implementation, we represent each list in adj as an ArrayStack, because we would like constant time access by position. Other options are also possible. Specifically, we could have implemented adj as a DLList.

The addEdge(i, j) operation just appends the value j to the list adj[i]:

```
AdjacencyLists
void addEdge(int i, int j) {
  adj[i].add(j);
}
```

This takes constant time.

The removeEdge(i, j) operation searches through the list adj[i] until it finds j and then removes it:

AdjacencyLists
```
void removeEdge(int i, int j) {
  Iterator<Integer> it = adj[i].iterator();
  while (it.hasNext()) {
    if (it.next() == j) {
      it.remove();
      return;
    }
  }
}
```

This takes $O(\deg(\mathtt{i}))$ time, where $\deg(\mathtt{i})$ (the *degree* of i) counts the number of edges in E that have i as their source.

The hasEdge(i, j) operation is similar; it searches through the list adj[i] until it finds j (and returns true), or reaches the end of the list (and returns false):

AdjacencyLists
```
boolean hasEdge(int i, int j) {
  return adj[i].contains(j);
}
```

This also takes $O(\deg(\mathtt{i}))$ time.

The outEdges(i) operation is very simple; it returns the list adj[i]:

AdjacencyLists
```
List<Integer> outEdges(int i) {
  return adj[i];
}
```

This clearly takes constant time.

The inEdges(i) operation is much more work. It scans over every vertex j checking if the edge (i, j) exists and, if so, adding j to the output list:

AdjacencyLists
```
List<Integer> inEdges(int i) {
  List<Integer> edges = new ArrayStack<Integer>();
  for (int j = 0; j < n; j++)
    if (adj[j].contains(i))  edges.add(j);
  return edges;
}
```

This operation is very slow. It scans the adjacency list of every vertex, so it takes $O(\mathtt{n}+\mathtt{m})$ time.

The following theorem summarizes the performance of the above data structure:

Theorem 12.2. *The* `AdjacencyLists` *data structure implements the* `Graph` *interface. An* `AdjacencyLists` *supports the operations*

- `addEdge(i, j)` *in constant time per operation;*
- `removeEdge(i, j)` *and* `hasEdge(i, j)` *in* $O(\deg(\mathtt{i}))$ *time per operation;*
- `outEdges(i)` *in constant time per operation; and*
- `inEdges(i)` *in* $O(\mathtt{n}+\mathtt{m})$ *time per operation.*

The space used by a `AdjacencyLists` *is* $O(\mathtt{n}+\mathtt{m})$.

As alluded to earlier, there are many different choices to be made when implementing a graph as an adjacency list. Some questions that come up include:

- What type of collection should be used to store each element of `adj`? One could use an array-based list, a linked-list, or even a hashtable.
- Should there be a second adjacency list, `inadj`, that stores, for each `i`, the list of vertices, `j`, such that $(\mathtt{j},\mathtt{i}) \in E$? This can greatly reduce the running-time of the `inEdges(i)` operation, but requires slightly more work when adding or removing edges.
- Should the entry for the edge $(\mathtt{i},\mathtt{j})$ in `adj[i]` be linked by a reference to the corresponding entry in `inadj[j]`?
- Should edges be first-class objects with their own associated data? In this way, `adj` would contain lists of edges rather than lists of vertices (integers).

Most of these questions come down to a tradeoff between complexity (and space) of implementation and performance features of the implementation.

12.3 Graph Traversal

In this section we present two algorithms for exploring a graph, starting at one of its vertices, i, and finding all vertices that are reachable from i. Both of these algorithms are best suited to graphs represented using an adjacency list representation. Therefore, when analyzing these algorithms we will assume that the underlying representation is an AdjacencyLists.

12.3.1 Breadth-First Search

The *bread-first-search* algorithm starts at a vertex i and visits, first the neighbours of i, then the neighbours of the neighbours of i, then the neighbours of the neighbours of the neighbours of i, and so on.

This algorithm is a generalization of the breadth-first traversal algorithm for binary trees (Section 6.1.2), and is very similar; it uses a queue, q, that initially contains only i. It then repeatedly extracts an element from q and adds its neighbours to q, provided that these neighbours have never been in q before. The only major difference between the breadth-first-search algorithm for graphs and the one for trees is that the algorithm for graphs has to ensure that it does not add the same vertex to q more than once. It does this by using an auxiliary boolean array, seen, that tracks which vertices have already been discovered.

```
                         Algorithms
void bfs(Graph g, int r) {
  boolean[] seen = new boolean[g.nVertices()];
  Queue<Integer> q = new SLList<Integer>();
  q.add(r);
  seen[r] = true;
  while (!q.isEmpty()) {
    int i = q.remove();
    for (Integer j : g.outEdges(i)) {
```

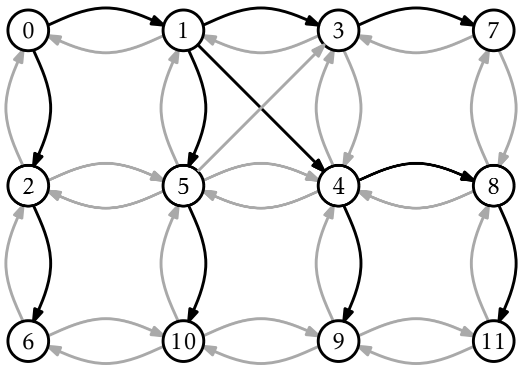


Figure 12.4: An example of bread-first-search starting at node 0. Nodes are labelled with the order in which they are added to q. Edges that result in nodes being added to q are drawn in black, other edges are drawn in grey.

```
        if (!seen[j]) {
          q.add(j);
          seen[j] = true;
        }
      }
    }
}
```

An example of running bfs(g,0) on the graph from Figure 12.1 is shown in Figure 12.4. Different executions are possible, depending on the ordering of the adjacency lists; Figure 12.4 uses the adjacency lists in Figure 12.3.

Analyzing the running-time of the bfs(g,i) routine is fairly straightforward. The use of the seen array ensures that no vertex is added to q more than once. Adding (and later removing) each vertex from q takes constant time per vertex for a total of $O(n)$ time. Since each vertex is processed by the inner loop at most once, each adjacency list is processed at most once, so each edge of G is processed at most once. This processing, which is done in the inner loop takes constant time per iteration, for a total of $O(m)$ time. Therefore, the entire algorithm runs in $O(n+m)$ time.

The following theorem summarizes the performance of the bfs(g,r) algorithm.

Theorem 12.3. *When given as input a* `Graph`, g, *that is implemented using the* `AdjacencyLists` *data structure, the* bfs(g, r) *algorithm runs in* $O(n+m)$ *time.*

A breadth-first traversal has some very special properties. Calling bfs(g, r) will eventually enqueue (and eventually dequeue) every vertex j such that there is a directed path from r to j. Moreover, the vertices at distance 0 from r (r itself) will enter q before the vertices at distance 1, which will enter q before the vertices at distance 2, and so on. Thus, the bfs(g, r) method visits vertices in increasing order of distance from r and vertices that cannot be reached from r are never visited at all.

A particularly useful application of the breadth-first-search algorithm is, therefore, in computing shortest paths. To compute the shortest path from r to every other vertex, we use a variant of bfs(g, r) that uses an auxilliary array, p, of length n. When a new vertex j is added to q, we set p[j] = i. In this way, p[j] becomes the second last node on a shortest path from r to j. Repeating this, by taking p[p[j]], p[p[p[j]]], and so on we can reconstruct the (reversal of) a shortest path from r to j.

12.3.2 Depth-First Search

The *depth-first-search* algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree. Another way to think of depth-first-search is by saying that it is similar to breadth-first search except that it uses a stack instead of a queue.

During the execution of the depth-first-search algorithm, each vertex, i, is assigned a colour, c[i]: white if we have never seen the vertex before, grey if we are currently visiting that vertex, and black if we are done visiting that vertex. The easiest way to think of depth-first-search is as a recursive algorithm. It starts by visiting r. When visiting a vertex i, we first mark i as grey. Next, we scan i's adjacency list and recursively visit any white vertex we find in this list. Finally, we are done processing i, so we colour i black and return.

Algorithms
```
void dfs(Graph g, int r) {
```

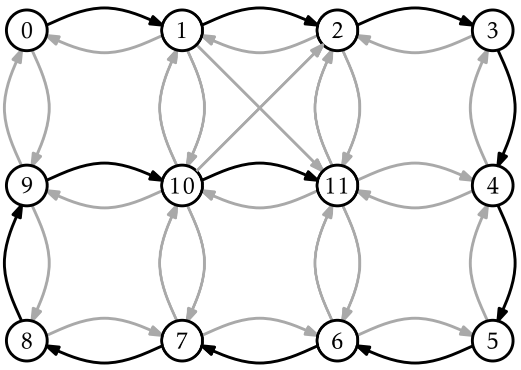


Figure 12.5: An example of depth-first-search starting at node 0. Nodes are labelled with the order in which they are processed. Edges that result in a recursive call are drawn in black, other edges are drawn in grey.

```
  byte[] c = new byte[g.nVertices()];
  dfs(g, r, c);
}
void dfs(Graph g, int i, byte[] c) {
  c[i] = grey;  // currently visiting i
  for (Integer j : g.outEdges(i)) {
    if (c[j] == white) {
      c[j] = grey;
      dfs(g, j, c);
    }
  }
  c[i] = black; // done visiting i
}
```

An example of the execution of this algorithm is shown in Figure 12.5.

Although depth-first-search may best be thought of as a recursive algorithm, recursion is not the best way to implement it. Indeed, the code given above will fail for many large graphs by causing a stack overflow. An alternative implementation is to replace the recursion stack with an explicit stack, s. The following implementation does just that:

Algorithms

```
void dfs2(Graph g, int r) {
  byte[] c = new byte[g.nVertices()];
  Stack<Integer> s = new Stack<Integer>();
  s.push(r);
  while (!s.isEmpty()) {
    int i = s.pop();
    if (c[i] == white) {
      c[i] = grey;
      for (int j : g.outEdges(i))
        s.push(j);
    }
  }
}
```

In the preceding code, when the next vertex, i, is processed, i is coloured grey and then replaced, on the stack, with its adjacent vertices. During the next iteration, one of these vertices will be visited.

Not surprisingly, the running times of dfs(g,r) and dfs2(g,r) are the same as that of bfs(g,r):

Theorem 12.4. *When given as input a* Graph*,* g*, that is implemented using the* AdjacencyLists *data structure, the* dfs(g,r) *and* dfs2(g,r) *algorithms each run in* $O(n+m)$ *time.*

As with the breadth-first-search algorithm, there is an underlying tree associated with each execution of depth-first-search. When a node $i \neq r$ goes from white to grey, this is because dfs(g,i,c) was called recursively while processing some node i′. (In the case of dfs2(g,r) algorithm, i is one of the nodes that replaced i′ on the stack.) If we think of i′ as the parent of i, then we obtain a tree rooted at r. In Figure 12.5, this tree is a path from vertex 0 to vertex 11.

An important property of the depth-first-search algorithm is the following: Suppose that when node i is coloured grey, there exists a path from i to some other node j that uses only white vertices. Then j will be coloured first grey then black before i is coloured black. (This can be proven by contradiction, by considering any path P from i to j.)

One application of this property is the detection of cycles. Refer to Figure 12.6. Consider some cycle, C, that can be reached from r. Let i be the first node of C that is coloured grey, and let j be the node that

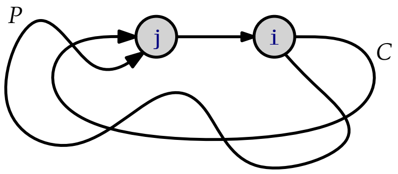


Figure 12.6: The depth-first-search algorithm can be used to detect cycles in G. The node j is coloured grey while i is still grey. This implies that there is a path, P, from i to j in the depth-first-search tree, and the edge (j,i) implies that P is also a cycle.

precedes i on the cycle C. Then, by the above property, j will be coloured grey and the edge (j,i) will be considered by the algorithm while i is still grey. Thus, the algorithm can conclude that there is a path, P, from i to j in the depth-first-search tree and the edge (j,i) exists. Therefore, P is also a cycle.

12.4 Discussion and Exercises

The running times of the depth-first-search and breadth-first-search algorithms are somewhat overstated by the Theorems 12.3 and 12.4. Define n_r as the number of vertices, i, of G, for which there exists a path from r to i. Define m_r as the number of edges that have these vertices as their sources. Then the following theorem is a more precise statement of the running times of the breadth-first-search and depth-first-search algorithms. (This more refined statement of the running time is useful in some of the applications of these algorithms outlined in the exercises.)

Theorem 12.5. *When given as input a* Graph, g, *that is implemented using the* AdjacencyLists *data structure, the* bfs(g,r), dfs(g,r) *and* dfs2(g,r) *algorithms each run in* $O(n_r + m_r)$ *time.*

Breadth-first search seems to have been discovered independently by Moore [52] and Lee [49] in the contexts of maze exploration and circuit routing, respectively.

Adjacency-list representations of graphs were presented by Hopcroft and Tarjan [40] as an alternative to the (then more common) adjacency-

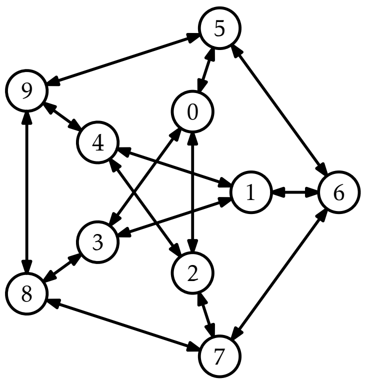


Figure 12.7: An example graph.

matrix representation. This representation, as well as depth-first-search, played a major part in the celebrated Hopcroft-Tarjan planarity testing algorithm that can determine, in $O(n)$ time, if a graph can be drawn, in the plane, and in such a way that no pair of edges cross each other [41].

In the following exercises, an undirected graph is one in which, for every i and j, the edge (i, j) is present if and only if the edge (j, i) is present.

Exercise 12.1. Draw an adjacency list representation and an adjacency matrix representation of the graph in Figure 12.7.

Exercise 12.2. The *incidence matrix* representation of a graph, G, is an $n \times m$ matrix, A, where

$$A_{i,j} = \begin{cases} -1 & \text{if vertex } i \text{ the source of edge } j \\ +1 & \text{if vertex } i \text{ the target of edge } j \\ 0 & \text{otherwise.} \end{cases}$$

1. Draw the incident matrix representation of the graph in Figure 12.7.
2. Design, analyze and implement an incidence matrix representation of a graph. Be sure to analyze the space, the cost of addEdge(i, j), removeEdge(i, j), hasEdge(i, j), inEdges(i), and outEdges(i).

Exercise 12.3. Illustrate an execution of the bfs(G, 0) and dfs(G, 0) on the graph, G, in Figure 12.7.

Exercise 12.4. Let G be an undirected graph. We say G is *connected* if, for every pair of vertices `i` and `j` in G, there is a path from `i` to `j` (since G is undirected, there is also a path from `j` to `i`). Show how to test if G is connected in $O(\mathtt{n}+\mathtt{m})$ time.

Exercise 12.5. Let G be an undirected graph. A *connected-component labelling* of G partitions the vertices of G into maximal sets, each of which forms a connected subgraph. Show how to compute a connected component labelling of G in $O(\mathtt{n}+\mathtt{m})$ time.

Exercise 12.6. Let G be an undirected graph. A *spanning forest* of G is a collection of trees, one per component, whose edges are edges of G and whose vertices contain all vertices of G. Show how to compute a spanning forest of of G in $O(\mathtt{n}+\mathtt{m})$ time.

Exercise 12.7. We say that a graph G is *strongly-connected* if, for every pair of vertices `i` and `j` in G, there is a path from `i` to `j`. Show how to test if G is strongly-connected in $O(\mathtt{n}+\mathtt{m})$ time.

Exercise 12.8. Given a graph $G = (V, E)$ and some special vertex $\mathtt{r} \in V$, show how to compute the length of the shortest path from `r` to `i` for every vertex $\mathtt{i} \in V$.

Exercise 12.9. Give a (simple) example where the `dfs(g,r)` code visits the nodes of a graph in an order that is different from that of the `dfs2(g,r)` code. Write a version of `dfs2(g,r)` that always visits nodes in exactly the same order as `dfs(g,r)`. (Hint: Just start tracing the execution of each algorithm on some graph where `r` is the source of more than 1 edge.)

Exercise 12.10. A *universal sink* in a graph G is a vertex that is the target of $\mathtt{n}-1$ edges and the source of no edges.[1] Design and implement an algorithm that tests if a graph G, represented as an `AdjacencyMatrix`, has a universal sink. Your algorithm should run in $O(\mathtt{n})$ time.

[1] A universal sink, `v`, is also sometimes called a *celebrity*: Everyone in the room recognizes `v`, but `v` doesn't recognize anyone else in the room.