see applications of depth-first search to sequencing tasks (Section 8.5) and to understanding the structure of the Web graph (Section 8.7).

8.1.2 For-Free Graph Primitives

The examples in Section 8.1.1 demonstrate that graph search is a fundamental and widely applicable primitive. I'm happy to report that, in this chapter, all our algorithms will be blazingly fast, running in just $O(m + n)$ time, where m and n denote the number of edges and vertices of the graph.[5] That's just a constant factor larger than the amount of time required to read the input![6] We conclude that these algorithms are "for-free primitives"—whenever you have graph data, you should feel free to apply any of these primitives to glean information about what it looks like.[7]

For-Free Primitives

We can think of an algorithm with linear or near-linear running time as a primitive that we can use essentially "for free" because the amount of computation used is barely more than the amount required just to read the input. When you have a primitive relevant to your problem that is so blazingly fast, why not use it? For example, you can always compute the connected components of your graph data in a preprocessing step, even if you're not quite sure how it will help later. One of the goals of this book series is to stock your algorithmic toolbox with as many for-free primitives as possible, ready to be applied at will.

8.1.3 Generic Graph Search

The point of a graph search algorithm is to solve the following problem.

[5] Also, the constants hidden in the big-O notation are reasonably small.

[6] In graph search and connectivity problems, there is no reason to expect that the input graph is connected. In the disconnected case, where m might be much smaller than n, the size of a graph is $\Theta(m + n)$ but not necessarily $\Theta(m)$.

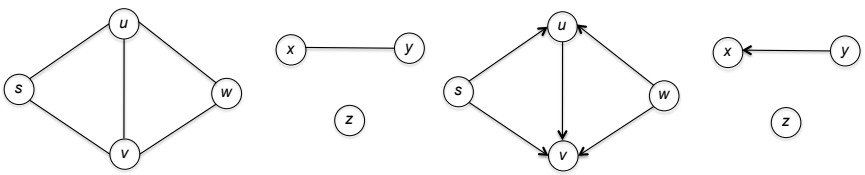
[7] Can we do better? No, up to the hidden constant factor: every correct algorithm must at least read the entire input in some cases.

Problem: Graph Search

Input: An undirected or directed graph $G = (V, E)$, and a starting vertex $s \in V$.

Goal: Identify the vertices of V reachable from s in G.

By a vertex v being "reachable," we mean that there is a sequence of edges in G that travels from s to v. If G is a directed graph, all the path's edges should be traversed in the forward (outgoing) direction. For example, in Figure 8.2(a), the set of reachable vertices (from s) is $\{s, u, v, w\}$. In the directed version of the graph in Figure 8.2(b), there is no directed path from s to w, and only the vertices s, u, and v are reachable from s via a directed path.[8]



(a) An undirected graph (b) A directed version

Figure 8.2: In (a), the set of vertices reachable from s is $\{s, u, v, w\}$. In (b), it is $\{s, u, v\}$.

The two graph search strategies that we'll focus on—breadth-first search and depth-first search—are different ways of instantiating a generic graph search algorithm. The generic algorithm systematically finds all the reachable vertices, taking care to avoid exploring anything twice. It maintains an extra variable with each vertex that keeps track of whether or not it has already been explored, planting a flag the first time that vertex is reached. The main loop's responsibility is to reach a new unexplored vertex in each iteration.

[8] In general, most of the algorithms and arguments in this chapter apply equally well to undirected and directed graphs. The big exception is computing connected components, which is a trickier problem in directed graphs than in undirected graphs.

```
GenericSearch

Input: graph G = (V, E) and a vertex s ∈ V.
Postcondition: a vertex is reachable from s if and
 only if it is marked as "explored."

---

mark s as explored, all other vertices as unexplored
while there is an edge (v, w) ∈ E with v explored and
 w unexplored do
    choose some such edge (v, w)  // underspecified
    mark w as explored
```

The algorithm is essentially the same for both directed and undirected graphs. In the directed case, the edge (v, w) chosen in an iteration of the while loop should be directed from an explored vertex v to an unexplored vertex w.

On Pseudocode

This book series explains algorithms using a mixture of high-level pseudocode and English (as above). I'm assuming that you have the skills to translate such high-level descriptions into working code in your favorite programming language. Several other books and resources on the Web offer concrete implementations of various algorithms in specific programming languages.

The first benefit of emphasizing high-level descriptions over language-specific implementations is flexibility. While I assume familiarity with *some* programming language, I don't care which one. Second, this approach promotes the understanding of algorithms at a deep and conceptual level, unencumbered by low-level details. Seasoned programmers and computer scientists generally think and communicate about algorithms at a similarly high level.

Still, there is no substitute for the detailed understanding of an algorithm that comes from providing

your own working implementation of it. I strongly encourage you to implement as many of the algorithms in this book as you have time for. (It's also a great excuse to pick up a new programming language!) For guidance, see the end-of-chapter Programming Problems and supporting test cases.

For example, in the graph in Figure 8.2(a), initially only our home base s is marked as explored. In the first iteration of the while loop, two edges meet the loop condition: (s, u) and (s, v). The `GenericSearch` algorithm chooses one of these edges—(s, u), say—and marks u as explored. In the second iteration of the loop, there are again two choices: (s, v) and (u, w). The algorithm might choose (u, w), in which case w is marked as explored. With one more iteration (after choosing either (s, v) or (w, v)), v is marked as explored. At this point, the edge (x, y) has two unexplored endpoints and the other edges have two explored endpoints, and the algorithm halts. As one would hope, the vertices marked as explored—s, u, v, and w—are precisely the vertices reachable from s.

This generic graph search algorithm is underspecified, as multiple edges (v, w) can be eligible for selection in an iteration of the while loop. Breadth-first search and depth-first search correspond to two specific decisions about which edge to explore next. No matter how this choice is made, the `GenericSearch` algorithm is guaranteed to be correct (in both undirected and directed graphs).

Proposition 8.1 (Correctness of Generic Graph Search) *At the conclusion of the `GenericSearch` algorithm, a vertex $v \in V$ is marked as explored if and only if there is a path from s to v in G.*

Section 8.1.5 provides a formal proof of Proposition 8.1; feel free to skip it if the proposition seems intuitively obvious.

On Lemmas, Theorems, and the Like

In mathematical writing, the most important technical statements are labeled *theorems*. A *lemma* is a technical statement that assists with the proof of a theorem (much as a subroutine assists with the

implementation of a larger program). A *corollary* is a statement that follows immediately from an already-proved result, such as a special case of a theorem. We use the term *proposition* for stand-alone technical statements that are not particularly important in their own right.

What about the running time of the `GenericSearch` algorithm? The algorithm explores each edge at most once—after an edge (v, w) has been explored for the first time, both v and w are marked as explored and the edge will not be considered again. This suggests that it should be possible to implement the algorithm in linear time, as long as we can quickly identify an eligible edge (v, w) in each iteration of the while loop. We'll see how this works in detail for breadth-first search and depth-first search in Sections 8.2 and 8.4, respectively.

8.1.4 Breadth-First and Depth-First Search

Every iteration of the `GenericSearch` algorithm chooses an edge that is "on the frontier" of the explored part of the graph, with one endpoint explored and the other unexplored (Figure 8.3). There can be many such edges, and to specify the algorithm fully we need a method for choosing one of them. We'll focus on the two most important strategies: breadth-first search and depth-first search. Both are excellent ways to explore a graph, and each has its own set of applications.

Breadth-first search (BFS). The high-level idea of *breadth-first search*—or *BFS* to its friends—is to explore the vertices of a graph cautiously, in "layers." Layer 0 consists only of the starting vertex s. Layer 1 contains the vertices that neighbor s, meaning the vertices v such that (s, v) is an edge of the graph (directed from s to v, in the case that G is directed). Layer 2 comprises the neighbors of layer-1 vertices that do not already belong to layer 0 or 1, and so on. In Sections 8.2 and 8.3, we'll see:

- how to implement BFS in linear time using a queue (first-in first-out) data structure;
- how to use BFS to compute (in linear time) the length of a shortest path between one vertex and all other vertices, with the

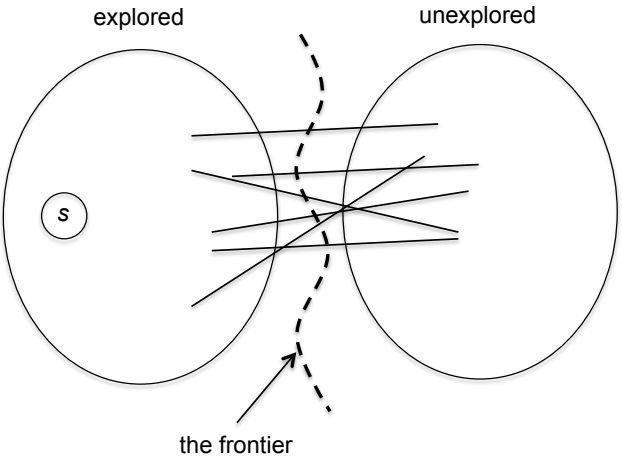


Figure 8.3: Every iteration of the `GenericSearch` algorithm chooses an edge "on the frontier," with one endpoint explored and the other unexplored.

layer-i vertices being precisely the vertices at distance i from s;

- how to use BFS to compute (in linear time) the connected components of an undirected graph.

Depth-first search (DFS). *Depth-first search—DFS* to its friends—is perhaps even more important. DFS employs a more aggressive strategy for exploring a graph, very much in the spirit of how you might explore a maze, going as deeply as you can and backtracking only when absolutely necessary. In Sections 8.4–8.7, we'll see:

- how to implement DFS in linear time using either recursion or an explicit stack (last-in first-out) data structure;

- how to use DFS to compute (in linear time) a topological ordering of the vertices of a directed acyclic graph, a useful primitive for task sequencing problems;

- how to use DFS to compute (in linear time) the "strongly connected components" of a directed graph, with applications to understanding the structure of the Web.

8.1.5 Correctness of the `GenericSearch` Algorithm

We now prove Proposition 8.1, which states that at the conclusion of the `GenericSearch` algorithm with input graph $G = (V, E)$ and starting vertex $s \in V$, a vertex $v \in V$ is marked as explored if and only if there is a path from s to v in G. As usual, if G is a directed graph, the $s \rightsquigarrow v$ path should also be directed, with all edges traversed in the forward direction.

The "only if" direction of the proposition should be intuitively clear: The only way that the `GenericSearch` algorithm discovers new vertices is by following paths from s.[9]

The "if" direction asserts the less obvious fact that the `GenericSearch` algorithm doesn't miss anything—it finds every vertex that it could conceivably discover. For this direction, we'll use a proof by contradiction. Recall that in this type of proof, you assume the *opposite* of what you want to prove, and then build on this assumption with a sequence of logically correct steps that culminates in a patently false statement. Such a contradiction implies that the assumption can't be true, which proves the desired statement.

So, assume that there is a path from s to v in the graph G, but the `GenericSearch` algorithm somehow misses it and concludes with the vertex v marked as unexplored. Let $S \subseteq V$ denote the vertices of G marked as explored by the algorithm. The vertex s belongs to S (by the first line of the algorithm), and the vertex v does not (by assumption). Because the $s \rightsquigarrow v$ path travels from a vertex inside S to one outside S, at least one edge e of the path has one endpoint u in S and the other w outside S (with e directed from u to w in the case that G is directed); see Figure 8.4. But this, my friends, is impossible: The edge e would be eligible for selection in the while loop of the `GenericSearch` algorithm, and the algorithm would have explored at least one more vertex, rather than giving up! There's no way that the `GenericSearch` algorithm could have halted at this point, so we've reached a contradiction. This contradiction concludes the proof of Proposition 8.1. $\mathscr{QED}$[10]

[9] If we wanted to be pedantic about it, we'd prove this direction by induction on the number of loop iterations.

[10] "Q.e.d." is an abbreviation for *quod erat demonstrandum*, and means "that which was to be demonstrated." In mathematical writing, it is used at the end of a proof to mark its completion.

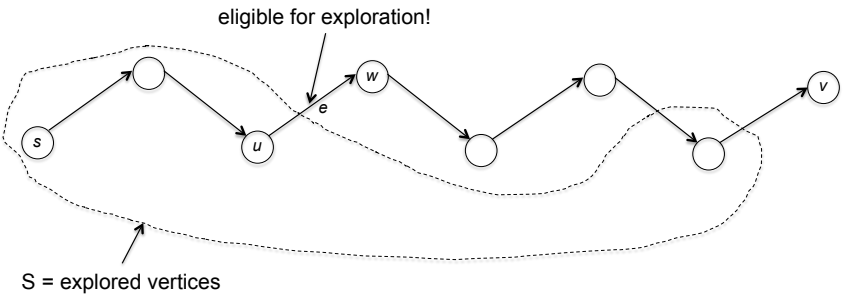


Figure 8.4: Proof of Proposition 8.1. As long as the `GenericSearch` algorithm has not yet discovered all the reachable vertices, there is an eligible edge along which it can explore further.

8.2 Breadth-First Search and Shortest Paths

Let's drill down on our first specific graph search strategy, *breadth-first search*.

8.2.1 High-Level Idea

Breadth-first search explores the vertices of a graph in layers, in order of increasing distance from the starting vertex. Layer 0 contains the starting vertex s and nothing else. Layer 1 is the set of vertices that are one hop away from s—that is, s's neighbors. These are the vertices that are explored immediately after s in breadth-first search. For example, in the graph in Figure 8.5, a and b are the neighbors of s and constitute layer 1. In general, the vertices in a layer i are those that neighbor a vertex in layer $i-1$ and that do not already belong to one of the layers $0, 1, 2, \ldots, i-1$. Breadth-first search explores all of layer-i vertices immediately after completing its exploration of layer-$(i-1)$ vertices. (Vertices not reachable from s do not belong to any layer.) For example, in Figure 8.5, the layer-2 vertices are c and d, as they neighbor layer-1 vertices but do not themselves belong to layer 0 or 1. (The vertex s is also a neighbor of a layer-1 vertex, but it already belongs to layer 0.) The last layer of the graph in Figure 8.5 comprises only the vertex e.

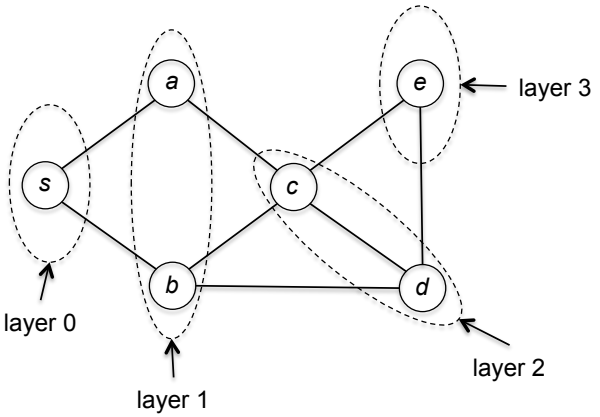


Figure 8.5: Breadth-first search discovers vertices in layers. The layer-i vertices are the neighbors of the layer-$(i-1)$ vertices that do not appear in any earlier layer.

Quiz 8.1

Consider an undirected graph with $n \geq 2$ vertices. What are the minimum and maximum number of different layers that the graph could have, respectively?

a) 1 and $n-1$

b) 2 and $n-1$

c) 1 and n

d) 2 and n

(See Section 8.2.6 for the solution and discussion.)

8.2.2 Pseudocode for `BFS`

Implementing breadth-first search in linear time requires a simple "first-in first-out" data structure known as a *queue*. BFS uses a queue to keep track of which vertices to explore next. If you're unfamiliar with queues, now is a good time to read up on them in your favorite introductory programming book (or on Wikipedia). The gist is that

a queue is a data structure for maintaining a list of objects, and you can remove stuff from the front or add stuff to the back in constant time.[11]

```
                              BFS

Input: graph G = (V, E) in adjacency-list
  representation, and a vertex s ∈ V.
Postcondition: a vertex is reachable from s if and
  only if it is marked as "explored."

-------------------------------------------------------

1 mark s as explored, all other vertices as unexplored
2 Q := a queue data structure, initialized with s
3 while Q is not empty do
4     remove the vertex from the front of Q, call it v
5     for each edge (v, w) in v's adjacency list do
6         if w is unexplored then
7             mark w as explored
8             add w to the end of Q
```

Each iteration of the while loop explores one new vertex. In line 5, BFS iterates through all the edges incident to the vertex v (if G is undirected) or through all the outgoing edges from v (if G is directed).[12] Unexplored neighbors of v are added to the end of the queue and are marked as explored; they will eventually be processed in later iterations of the algorithm.

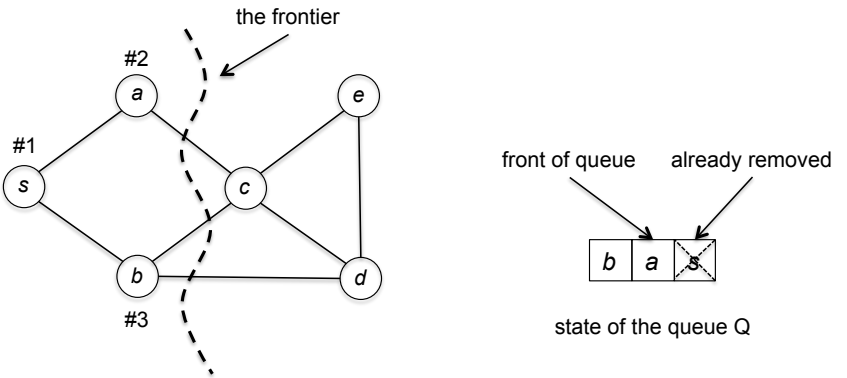
8.2.3 An Example

Let's see how our pseudocode works for the graph in Figure 8.5, numbering the vertices in order of insertion into the queue (equivalently, in order of exploration). The starting vertex s is always the first to

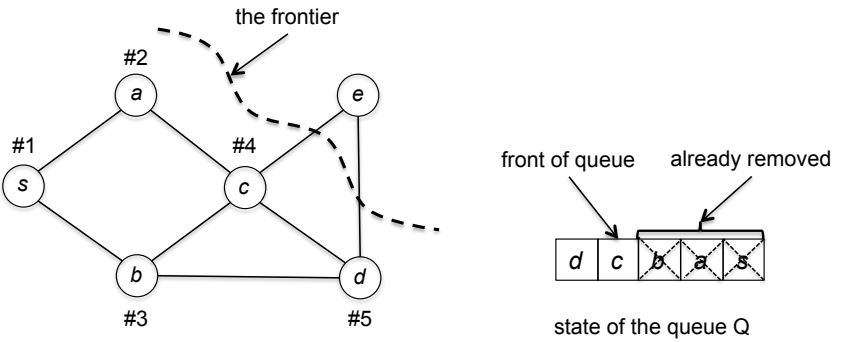
[11] You may never need to implement a queue from scratch, as they are built in to most modern programming languages. If you do, you can use a doubly linked list. Or, if you have advance knowledge of the maximum number of objects that you might have to store (which is $|V|$, in the case of `BFS`), you can get away with a fixed-length array and a couple of indices (which keep track of the front and back of the queue).

[12] This is the step where it's so convenient to have the input graph represented via adjacency lists.

be explored. The first iteration of the while loop extracts s from the queue Q and the subsequent for loop examines the edges (s, a) and (s, b), in whatever order these edges appear in s's adjacency list. Because neither a nor b is marked as explored, both get inserted into the queue. Let's say that edge (s, a) came first and so a is inserted before b. The current state of the graph and the queue is now:

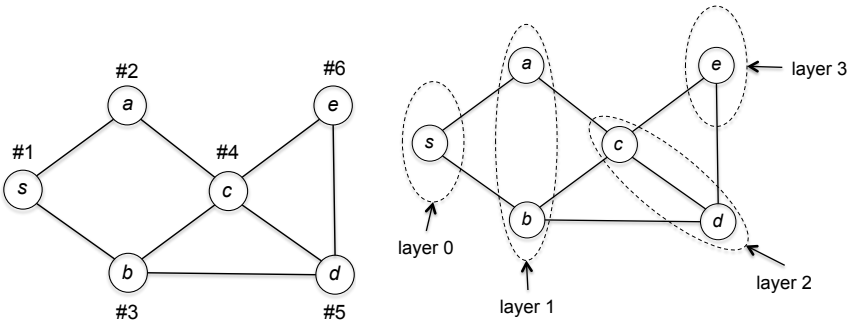


The next iteration of the while loop extracts the vertex a from the front of the queue, and considers its incident edges (s, a) and (a, c). It skips over the former after double-checking that s is already marked as explored, and adds the (previously unexplored) vertex c to the end of the queue. The third iteration extracts the vertex b from the front of the queue and adds vertex d to the end (because s and c are already marked as explored, they are skipped over). The new picture is:



In the fourth iteration, the vertex c is removed from the front of the queue. Of its neighbors, the vertex e is the only one not encountered

before, and it is added to the end of the queue. The final two iterations extract d and then e from the queue, and verify that all of their neighbors have already been explored. The queue is then empty, and the algorithm halts. The vertices are explored in order of the layers, with the layer-i vertices explored immediately after the layer-$(i-1)$ vertices (Figure 8.6).



(a) Order of exploration (b) Layers

Figure 8.6: In breadth-first search, the layer-i vertices are explored immediately after the layer-$(i-1)$ vertices.

8.2.4 Correctness and Running Time

Breadth-first search discovers all the vertices reachable from the starting vertex, and it runs in linear time. The more refined running time bound in Theorem 8.2(c) below will come in handy for our linear-time algorithm for computing connected components (described in Section 8.3).

Theorem 8.2 (Properties of `BFS`) *For every undirected or directed graph $G = (V, E)$ in adjacency-list representation and for every starting vertex $s \in V$:*

(a) At the conclusion of `BFS`, a vertex $v \in V$ is marked as explored if and only if there is a path from s to v in G.

(b) The running time of `BFS` is $O(m + n)$, where $m = |E|$ and $n = |V|$.

(c) The running time of lines 2–8 of `BFS` *is*

$$O(m_s + n_s),$$

where m_s and n_s denote the number of edges and vertices, respectively, reachable from s in G.

Proof: Part (a) follows from the guarantee in Proposition 8.1 for the generic graph search algorithm `GenericSearch`, of which `BFS` is a special case.[13] Part (b) follows from part (c), as the overall running time of `BFS` is just the running time of lines 2–8 plus the $O(n)$ time needed for the initialization in line 1.

We can prove part (c) by inspecting the pseudocode. The initialization in line 2 takes $O(1)$ time. In the main while loop, the algorithm only ever encounters the n_s vertices that are reachable from s. Because no vertex is explored twice, each such vertex is added to the end of the queue and removed from the front of the queue exactly once. Each of these operations takes $O(1)$ time—this is the whole point of the first-in first-out queue data structure—and so the total amount of time spent in lines 3–4 and 7–8 is $O(n_s)$. Each of the m_s edges (v, w) reachable from s is processed in line 5 at most twice—once when v is explored, and once when w is explored.[14] Thus the total amount of time spent in lines 5–6 is $O(m_s)$, and the overall running time for lines 2–8 is $O(m_s + n_s)$. *QED*

8.2.5 Shortest Paths

The properties in Theorem 8.2 are not unique to breadth-first search—for example, they also hold for depth-first search. What *is* unique about BFS is that, with just a couple extra lines of code, it efficiently computes shortest-path distances.

[13] Formally, `BFS` is equivalent to the version of `GenericSearch` where, in every iteration of the latter's while loop, the algorithm chooses the eligible edge (v, w) for which v was discovered the earliest, breaking ties among v's eligible edges according to their order in v's adjacency list. If that sounds too complicated, you can alternatively check that the proof of Proposition 8.1 holds verbatim also for breadth-first search. Intuitively, breadth-first search discovers vertices only by exploring paths from s; as long as it hasn't explored every vertex on a path, the "next vertex" on the path is still in the queue awaiting future exploration.

[14] If G is a directed graph, each edge is processed at most once, when its tail vertex is explored.

Problem Definition

In a graph G, we use the notation $dist(v, w)$ for the fewest number of edges in a path from v to w (or $+\infty$, if G contains no path from v to w).[15]

Problem: Shortest Paths (Unit Edge Lengths)

Input: An undirected or directed graph $G = (V, E)$, and a starting vertex $s \in V$.

Output: $dist(s, v)$ for every vertex $v \in V$.[16]

For example, if G is the movie network and s is the vertex corresponding to Kevin Bacon, the problem of computing shortest paths is precisely the problem of computing everyone's Bacon number (Section 8.1.1). The basic graph search problem (Section 8.1.3) corresponds to the special case of identifying all the vertices v with $dist(s, v) \neq +\infty$.

Pseudocode

To compute shortest paths, we add two lines to the basic `BFS` algorithm (lines 2 and 9 below); these increase the algorithm's running time by a small constant factor. The first one initializes preliminary estimates of vertices' shortest-path distances—0 for s, and $+\infty$ for the other vertices, which might not even be reachable from s. The second one executes whenever a vertex w is discovered for the first time, and computes w's final shortest-path distance as one more than that of the vertex v that triggered w's discovery.

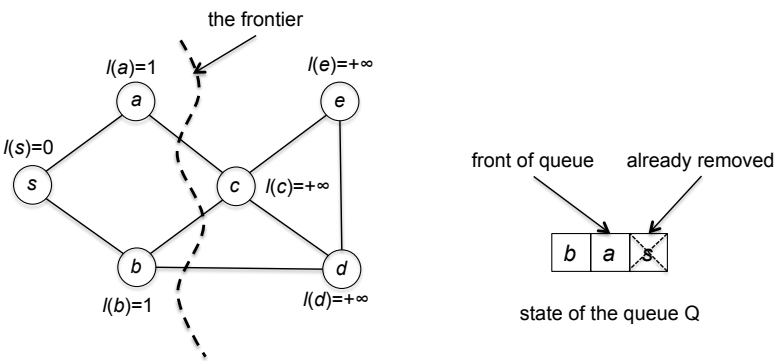
[15] As usual, if G is directed, all the edges of the path should be traversed in the forward direction.

[16] The phrase "unit edge lengths" in the problem statement refers to the assumption that each edge of G contributes 1 to the length of a path. Chapter 9 generalizes `BFS` to compute shortest paths in graphs in which each edge has its own nonnegative length.

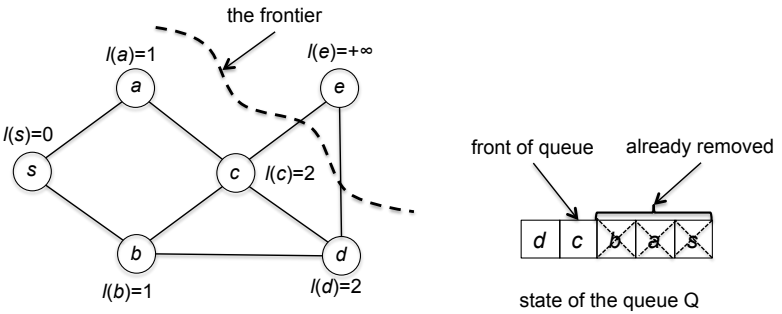
```
Augmented-BFS

Input: graph G = (V, E) in adjacency-list
  representation, and a vertex s ∈ V.
Postcondition: for every vertex v ∈ V, the value l(v)
  equals the true shortest-path distance dist(s, v).

-------------------------------------------------------
 1 mark s as explored, all other vertices as unexplored
 2 l(s) := 0, l(v) := +∞ for every v ≠ s
 3 Q := a queue data structure, initialized with s
 4 while Q is not empty do
 5     remove the vertex from the front of Q, call it v
 6     for each edge (v, w) in v's adjacency list do
 7         if w is unexplored then
 8             mark w as explored
 9             l(w) := l(v) + 1
10             add w to the end of Q
```

Example and Analysis

In our running example (Figure 8.6), the first iteration of the while loop discovers the vertices a and b. Because s triggered their discovery and $l(s) = 0$, the algorithm reassigns $l(a)$ and $l(b)$ from $+\infty$ to 1:



The second iteration of the while loop processes the vertex a, leading to c's discovery. The algorithm reassigns $l(c)$ from $+\infty$ to $l(a) + 1$, which is 2. Similarly, in the third iteration, $l(d)$ is set to $l(b) + 1$, which is also 2:



The fourth iteration discovers the final vertex e via the vertex c, and sets $l(e)$ to $l(c) + 1$, which is 3. At this point, for every vertex v, $l(v)$ equals the true shortest-path distance $dist(s, v)$, which also equals the number of the layer that contains v (Figure 8.6). These properties hold in general, and not just for this example.

Theorem 8.3 (Properties of `Augmented-BFS`) *For every undirected or directed graph $G = (V, E)$ in adjacency-list representation and for every starting vertex $s \in V$:*

(a) At the conclusion of `Augmented-BFS`, for every vertex $v \in V$, the value of $l(v)$ equals the length $dist(s, v)$ of a shortest path from s to v in G (or $+\infty$, if no such path exists).

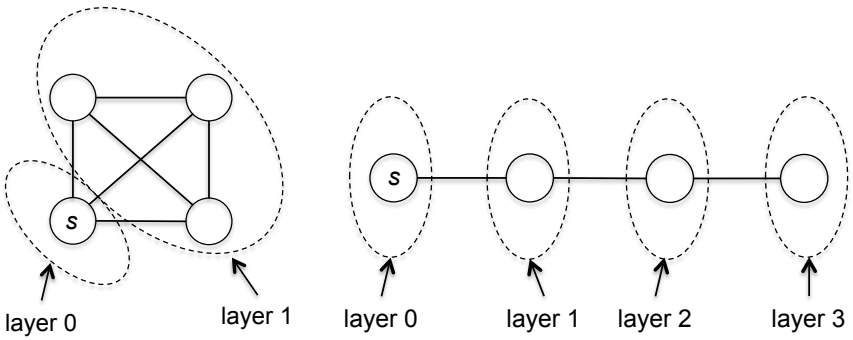
(b) The running time of `Augmented-BFS` is $O(m+n)$, where $m = |E|$ and $n = |V|$.

Because the asymptotic running time of the `Augmented-BFS` algorithm is the same as that of `BFS`, part (b) of Theorem 8.3 follows from the latter's running time guarantee (Theorem 8.2(b)). Part (a) follows from two observations. First, the vertices v with $dist(s, v) = i$ are precisely the vertices in the ith layer of the graph—this is why we defined layers the way we did. Second, for every layer-i vertex w, `Augmented-BFS` eventually sets $l(w) = i$ (since w is discovered via a layer-$(i-1)$ vertex v with $l(v) = i-1$). For vertices not in any layer—that is, not reachable from s—both $dist(s, v)$ and $l(v)$ are $+\infty$.[17]

[17] If you're hungry for a more rigorous proof, then proceed—in the privacy of your own home—by induction on the number of while loop iterations performed by the `Augmented-BFS` algorithm. Alternatively, Theorem 8.3(a) is a special case of the correctness of Dijkstra's shortest-path algorithm, as proved in Section 9.3.

8.2.6 Solution to Quiz 8.1

Correct answer: (d). An undirected graph with $n \geq 2$ vertices has at least two layers and at most n layers. When $n \geq 2$, there cannot be fewer than two layers because s is the only vertex in layer 0. Complete graphs have only two layers (Figure 8.7(a)). There cannot be more than n layers, as layers are disjoint and contain at least one vertex each. Path graphs have n layers (Figure 8.7(b)).



(a) A complete graph (b) A path graph

Figure 8.7: An n-vertex graph can have anywhere from two to n different layers.

8.3 Computing Connected Components

In this section, $G = (V, E)$ will always denote an *undirected* graph. We postpone the more difficult connectivity problems in directed graphs until Section 8.6.

8.3.1 Connected Components

An undirected graph $G = (V, E)$ naturally falls into "pieces," which are called *connected components* (Figure 8.8). More formally, a connected component is a maximal subset $S \subseteq V$ of vertices such that there is a path from any vertex in S to any other vertex in S.[18] For example,

[18] Still more formally, the connected components of a graph can be defined as the *equivalence classes* of a suitable *equivalence relation*. Equivalence relations are usually covered in a first course on proofs or on discrete mathematics. A

graphs and complete graphs (Figure 8.7) are two examples. At the other extreme, in a graph with no edges, each vertex is in its own connected component, for a total of n. There cannot be more than n connected components, as they are disjoint and each contains at least one vertex.

8.4 Depth-First Search

Why do we need another graph search strategy? After all, breadth-first search seems pretty awesome—it finds all the vertices reachable from the starting vertex in linear time, and can even compute shortest-path distances along the way.

There's another linear-time graph search strategy, *depth-first search (DFS)*, which comes with its own impressive catalog of applications (not already covered by BFS). For example, we'll see how to use DFS to compute in linear time a topological ordering of the vertices of a directed acyclic graph, as well as the connected components (appropriately defined) of a directed graph.

8.4.1 An Example

If breadth-first search is the cautious and tentative exploration strategy, depth-first search is its more aggressive cousin, always exploring from the most recently discovered vertex and backtracking only when necessary (like exploring a maze). Before we describe the full pseudocode for DFS, let's illustrate how it works on the same running example used in Section 8.2 (Figure 8.9).

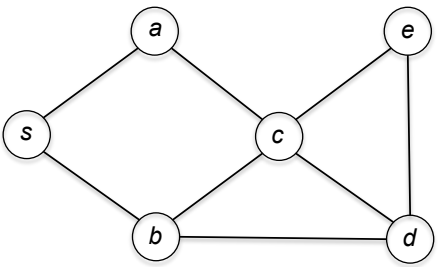
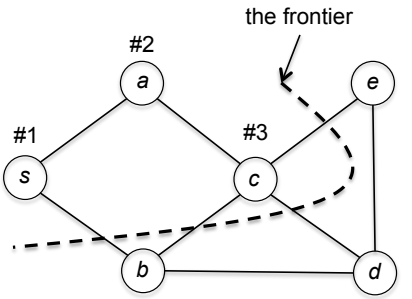
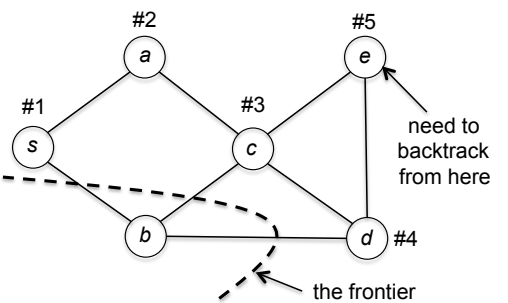


Figure 8.9: Running example for depth-first search.

Like BFS, DFS marks a vertex as explored the first time it discovers it. Because it begins its exploration at the starting vertex s, for the graph in Figure 8.9, the first iteration of DFS examines the edges (s, a) and (s, b), in whatever order these edges appear in s's adjacency list. Let's say (s, a) comes first, leading DFS to discover the vertex a and mark it as explored. The second iteration of DFS is where it diverges from BFS—rather than considering next s's other layer-1 neighbor b, DFS immediately proceeds to exploring the neighbors of a. (It will eventually get back to exploring (s, b).) Perhaps from a it checks s first (which is already marked as explored) and then discovers the vertex c, which is where it travels next:



Then DFS examines in some order the neighbors of c, the most recently discovered vertex. To keep things interesting, let's say that DFS discovers d next, followed by e:



From e, DFS has nowhere to go—both of e's neighbors are already marked as explored. DFS is forced to retreat to the previous vertex, namely d, and resume exploring the rest of its neighbors. From d, DFS will discover the final vertex b (perhaps after checking c and finding it marked as explored). Once at b, the dominoes fall quickly. DFS

discovers that all of b's neighbors have already been explored, and must backtrack to the previously visited vertex, which is d. Similarly, because all of d's remaining neighbors are already marked as explored, DFS must rewind further, to c. DFS then retreats further to a (after checking that all of c's remaining neighbors are marked as explored), then to s. It finally stops once it checks s's remaining neighbor (which is b) and finds it marked as explored.

8.4.2 Pseudocode for `DFS`

Iterative Implementation

One way to think about and implement `DFS` is to start from the code for `BFS` and make two changes: (i) swap in a stack data structure (which is last-in first-out) for the queue (which is first-in first-out); and (ii) postpone checking whether a vertex has already been explored until after removing it from the data structure.[20,21]

`DFS` (Iterative Version)

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.
Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

```
mark all vertices as unexplored
S := a stack data structure, initialized with s
while S is not empty do
    remove ("pop") the vertex v from the front of S
    if v is unexplored then
        mark v as explored
        for each edge (v, w) in v's adjacency list do
            add ("push") w to the front of S
```

[20] A *stack* is a "last-in first-out" data structure—like those stacks of upside-down trays at a cafeteria—that is typically studied in a first programming course (along with queues, see footnote 11). A stack maintains a list of objects, and you can add an object to the beginning of the list (a "push") or remove one from the beginning of the list (a "pop") in constant time.

[21] Would the algorithm behave the same if we made only the first change?

As usual, the edges processed in the for loop are the edges incident to v (if G is an undirected graph) or the edges outgoing from v (if G is a directed graph).

For example, in the graph in Figure 8.9, the first iteration of `DFS`'s while loop pops the vertex s and pushes its two neighbors onto the stack in some order, say, with b first and a second. Because a was the last to be pushed, it is the first to be popped, in the second iteration of the while loop. This causes s and c to be pushed onto the stack, let's say with c first. The vertex s is popped in the next iteration; since it has already been marked as explored, the algorithm skips it. Then c is popped, and all of its neighbors (a, b, d, and e) are pushed onto the stack, joining the first occurrence of b. If d is pushed last, and also b is pushed before e when d is popped in the next iteration, then we recover the order of exploration from Section 8.4.1 (as you should check).

Recursive Implementation

Depth-first search also has an elegant recursive implementation.[22]

`DFS` **(Recursive Version)**

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.
Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

```
// all vertices unexplored before outer call
```
mark s as explored
for each edge (s, v) in s's adjacency list **do**
    if v is unexplored **then**
        `DFS` (G, v)

In this implementation, all recursive calls to `DFS` have access to the same set of global variables which track the vertices that have been marked as explored (with all vertices initially unexplored). The aggressive nature of `DFS` is perhaps more obvious in this implementation—the

[22] I'm assuming you've heard of recursion as part of your programming background. A recursive procedure is one that invokes itself as a subroutine.

algorithm immediately recurses on the first unexplored neighbor that it finds, before considering the remaining neighbors.[23] In effect, the explicit stack data structure in the iterative implementation of `DFS` is being simulated by the program stack of recursive calls in the recursive implementation.[24]

8.4.3 Correctness and Running Time

Depth-first search is just as correct and just as blazingly fast as breadth-first search, for the same reasons (cf., Theorem 8.2).[25]

Theorem 8.5 (Properties of `DFS`) *For every undirected or directed graph $G = (V, E)$ in adjacency-list representation and for every starting vertex $s \in V$:*

(a) At the conclusion of `DFS`, a vertex $v \in V$ is marked as explored if and only if there is a path from s to v in G.

(b) The running time of `DFS` is $O(m + n)$, where $m = |E|$ and $n = |V|$.

Part (a) holds because depth-first search is a special case of the generic graph search algorithm `GenericSearch` (see Proposition 8.1).[26] Part (b) holds because `DFS` examines each edge at most twice (once from each endpoint) and, because the stack supports pushes and pops in $O(1)$ time, performs a constant number of operations per edge examination (for $O(m)$ total). The initialization requires $O(n)$ time.[27]

[23] As stated, the two versions of `DFS` explore the edges in a vertex's adjacency list in opposite orders. (Do you see why?) If one of the versions is modified to iterate backward through a vertex's adjacency list, then the iterative and recursive implementations explore the vertices in the same order.

[24] Pro tip: If your computer runs out of memory while executing the recursive version of `DFS` on a big graph, you should either switch to the iterative version or increase the program stack size in your programming environment.

[25] The abbreviation "cf." stands for *confer* and means "compare to."

[26] Formally, `DFS` is equivalent to the version of `GenericSearch` in which, in every iteration of the latter's while loop, the algorithm chooses the eligible edge (v, w) for which v was discovered most recently. Ties among v's eligible edges are broken according to their order (for the recursive version) or their reverse order (for the iterative version) in v's adjacency list.

[27] The refined bound in Theorem 8.2(c) also holds for `DFS` (for the same reasons), which means `DFS` can substitute for `BFS` in the linear-time `UCC` algorithm for computing connected components in Section 8.3.