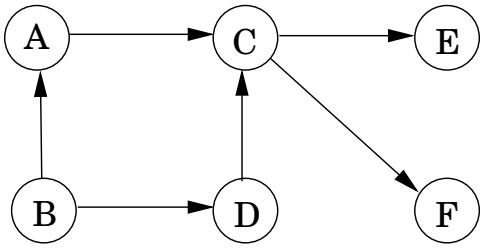
Figure 3.8 A directed acyclic graph with one source, two sinks, and four possible linearizations.



What types of dags can be linearized? Simple: *All of them*. And once again depth-first search tells us exactly how to do it: simply perform tasks in *decreasing* order of their `post` numbers. After all, the only edges (u, v) in a graph for which `post`(u) <`post`(v) are back edges (recall the table of edge types on page 100)—and we have seen that a dag cannot have back edges. Therefore:

Property *In a dag, every edge leads to a vertex with a lower* `post` *number.*

This gives us a linear-time algorithm for ordering the nodes of a dag. And, together with our earlier observations, it tells us that three rather different-sounding properties—acyclicity, linearizability, and the absence of back edges during a depth-first search—are in fact one and the same thing.

Since a dag is linearized by decreasing `post` numbers, the vertex with the smallest `post` number comes last in this linearization, and it must be a *sink*—no outgoing edges. Symmetrically, the one with the highest `post` is a *source*, a node with no incoming edges.

Property *Every dag has at least one source and at least one sink.*

The guaranteed existence of a source suggests an alternative approach to linearization:

> Find a source, output it, and delete it from the graph.
>
> Repeat until the graph is empty.

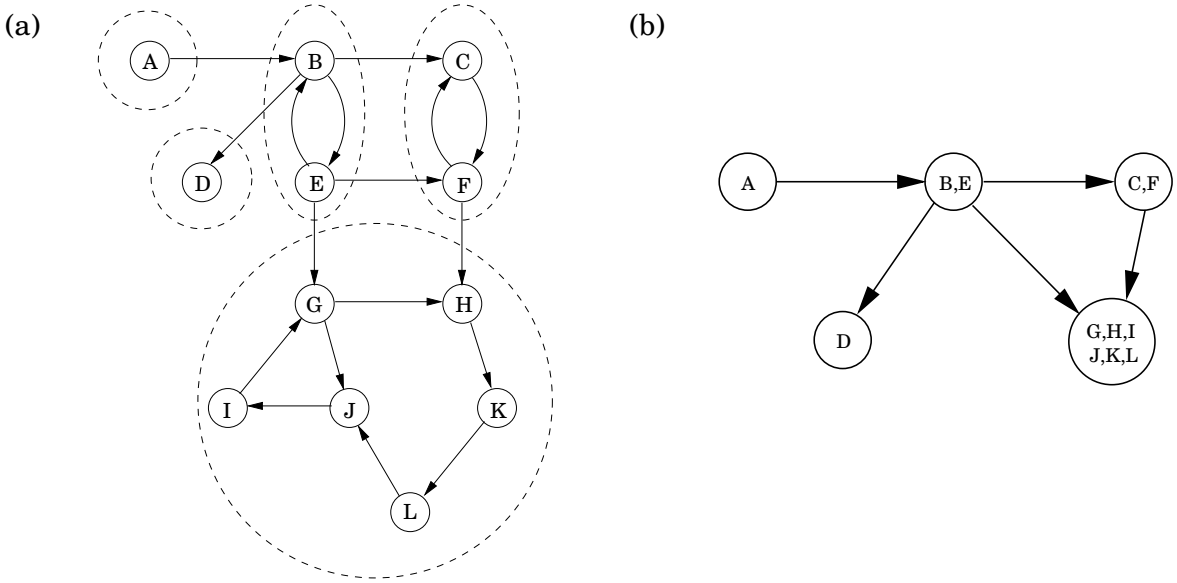
Can you see why this generates a valid linearization for any dag? What happens if the graph has cycles? And, how can this algorithm be implemented in linear time? (Exercise 3.14.)

3.4 Strongly connected components

3.4.1 Defining connectivity for directed graphs

Connectivity in undirected graphs is pretty straightforward: a graph that is not connected can be decomposed in a natural and obvious manner into several connected components (Fig-

Figure 3.9 (a) A directed graph and its strongly connected components. (b) The meta-graph.



ure 3.6 is a case in point). As we saw in Section 3.2.3, depth-first search does this handily, with each restart marking a new connected component.

In directed graphs, connectivity is more subtle. In some primitive sense, the directed graph of Figure 3.9(a) is "connected"—it can't be "pulled apart," so to speak, without breaking edges. But this notion is hardly interesting or informative. The graph cannot be considered connected, because for instance there is no path from G to B or from F to A. The right way to define connectivity for directed graphs is this:

> Two nodes u and v of a directed graph are *connected* if there is a path from u to v *and* a path from v to u.

This relation partitions V into disjoint sets (Exercise 3.30) that we call *strongly connected components*. The graph of Figure 3.9(a) has five of them.

Now shrink each strongly connected component down to a single meta-node, and draw an edge from one meta-node to another if there is an edge (in the same direction) between their respective components (Figure 3.9(b)). The resulting *meta-graph* must be a dag. The reason is simple: a cycle containing several strongly connected components would merge them all into a single, strongly connected component. Restated,

Property *Every directed graph is a dag of its strongly connected components.*

This tells us something important: The connectivity structure of a directed graph is two-tiered. At the top level we have a dag, which is a rather simple structure—for instance, it

can be linearized. If we want finer detail, we can look inside one of the nodes of this dag and examine the full-fledged strongly connected component within.

3.4.2 An efficient algorithm

The decomposition of a directed graph into its strongly connected components is very informative and useful. It turns out, fortunately, that it can be found in linear time by making further use of depth-first search. The algorithm is based on some properties we have already seen but which we will now pinpoint more closely.

Property 1 *If the* `explore` *subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.*

Therefore, if we call `explore` on a node that lies somewhere in a *sink* strongly connected component (a strongly connected component that is a sink in the meta-graph), then we will retrieve exactly that component. Figure 3.9 has two sink strongly connected components. Starting `explore` at node K, for instance, will completely traverse the larger of them and then stop.

This suggests a way of finding one strongly connected component, but still leaves open two major problems: (A) how do we find a node that we know for sure lies in a sink strongly connected component and (B) how do we continue once this first component has been discovered?

Let's start with problem (A). There is not an easy, direct way to pick out a node that is guaranteed to lie in a sink strongly connected component. But there is a way to get a node in a *source* strongly connected component.

Property 2 *The node that receives the highest* `post` *number in a depth-first search must lie in a source strongly connected component.*

This follows from the following more general property.

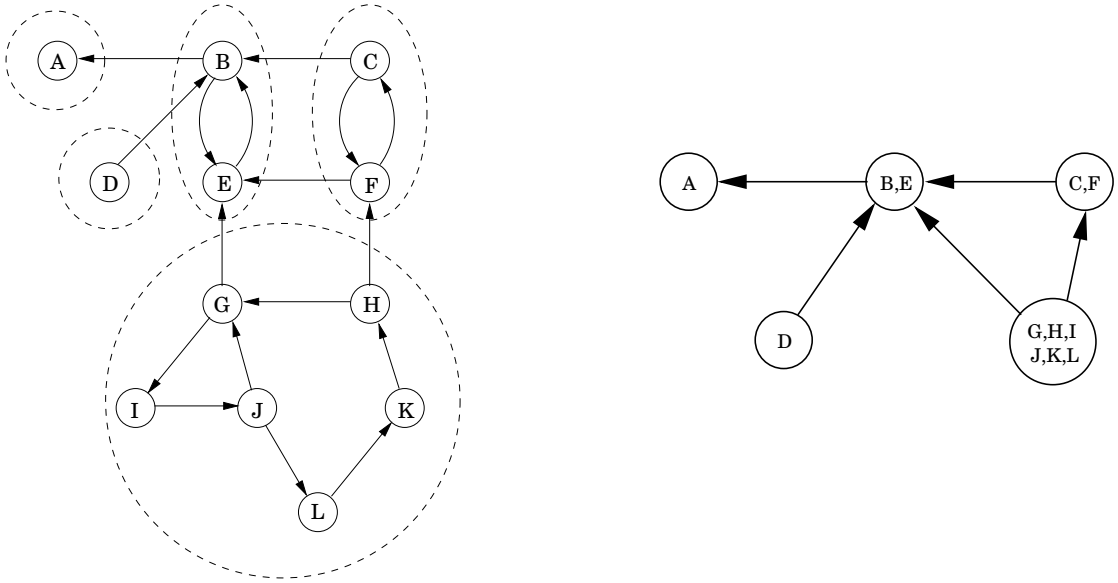
Property 3 *If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest* `post` *number in C is bigger than the highest* `post` *number in C'.*

Proof. In proving Property 3, there are two cases to consider. If the depth-first search visits component C before component C', then clearly all of C and C' will be traversed before the procedure gets stuck (see Property 1). Therefore the first node visited in C will have a higher `post` number than any node of C'. On the other hand, if C' gets visited first, then the depth-first search will get stuck after seeing all of C' but before seeing any of C, in which case the property follows immediately. ∎

Property 3 can be restated as saying that *the strongly connected components can be linearized by arranging them in decreasing order of their highest* `post` *numbers*. This is a generalization of our earlier algorithm for linearizing dags; in a dag, each node is a singleton strongly connected component.

Property 2 helps us find a node in the source strongly connected component of G. However, what we need is a node in the *sink* component. Our means seem to be the opposite of

Figure 3.10 The *reverse* of the graph from Figure 3.9.



our needs! But consider the *reverse* graph G^R, the same as G but with all edges reversed (Figure 3.10). G^R has exactly the same strongly connected components as G (why?). So, if we do a depth-first search of G^R, the node with the highest `post` number will come from a source strongly connected component in G^R, which is to say a sink strongly connected component in G. We have solved problem (A)!

Onward to problem (B). How do we continue after the first sink component is identified? The solution is also provided by Property 3. Once we have found the first strongly connected component and deleted it from the graph, the node with the highest `post` number among those remaining will belong to a sink strongly connected component of whatever remains of G. Therefore we can keep using the `post` numbering from our initial depth-first search on G^R to successively output the second strongly connected component, the third strongly connected component, and so on. The resulting algorithm is this.

1. Run depth-first search on G^R.
2. Run the undirected connected components algorithm (from Section 3.2.3) on G, and during the depth-first search, process the vertices in decreasing order of their `post` numbers from step 1.

This algorithm is linear-time, only the constant in the linear term is about twice that of straight depth-first search. (Question: How does one construct an adjacency list representation of G^R in linear time? And how, in linear time, does one order the vertices of G by decreasing `post` values?)

Let's run this algorithm on the graph of Figure 3.9. If step 1 considers vertices in lexicographic order, then the ordering it sets up for the second step (namely, decreasing `post` numbers in the depth-first search of G^R) is: $G, I, J, L, K, H, D, C, F, B, E, A$. Then step 2 peels off components in the following sequence: $\{G, H, I, J, K, L\}, \{D\}, \{C, F\}, \{B, E\}, \{A\}$.

Crawling fast

All this assumes that the graph is neatly given to us, with vertices numbered 1 to n and edges tucked in adjacency lists. The realities of the World Wide Web are very different. The nodes of the Web graph are not known in advance, and they have to be discovered one by one during the process of search. And, of course, recursion is out of the question.

Still, crawling the Web is done by algorithms very similar to depth-first search. An explicit stack is maintained, containing all nodes that have been discovered (as endpoints of hyperlinks) but not yet explored. In fact, this "stack" is not exactly a last-in, first-out list. It gives highest priority not to the nodes that were inserted most recently (nor the ones that were inserted earliest, that would be a *breadth-first search*, see Chapter 4), but to the ones that look most "interesting"—a heuristic criterion whose purpose is to keep the stack from overflowing and, in the worst case, to leave unexplored only nodes that are very unlikely to lead to vast new expanses.

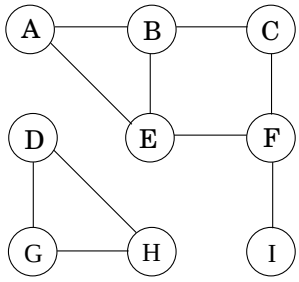
In fact, crawling is typically done by many computers running `explore` simultaneously: each one takes the next node to be explored from the top of the stack, downloads the http file (the kind of Web files that point to each other), and scans it for hyperlinks. But when a new http document is found at the end of a hyperlink, no recursive calls are made: instead, the new vertex is inserted in the central stack.

But one question remains: When we see a "new" document, how do we know that it is indeed new, that we have not seen it before in our crawl? And how do we give it a *name*, so it can be inserted in the stack and recorded as "already seen"? The answer is *by hashing*.

Incidentally, researchers have run the strongly connected components algorithm on the Web and have discovered some very interesting structure.

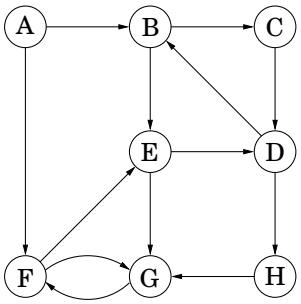
Exercises

3.1. Perform a depth-first search on the following graph; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge or back edge, and give the `pre` and `post` number of each vertex.

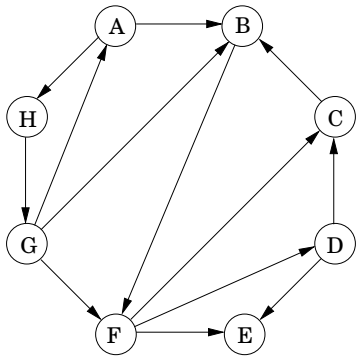


3.2. Perform depth-first search on each of the following graphs; whenever there's a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge, or cross edge, and give the `pre` and `post` number of each vertex.

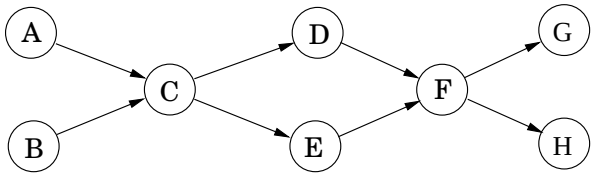
(a)



(b)



3.3. Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.



(a) Indicate the `pre` and `post` numbers of the nodes.

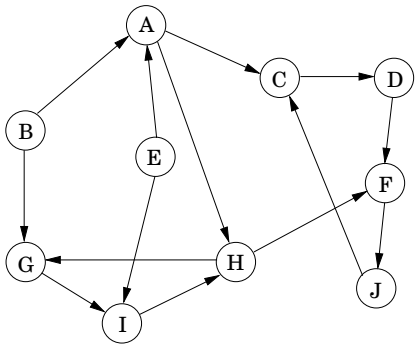
(b) What are the sources and sinks of the graph?

(c) What topological ordering is found by the algorithm?

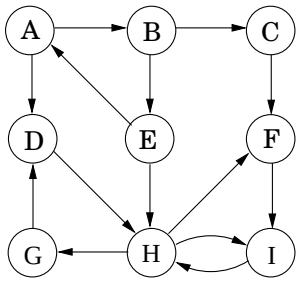
(d) How many topological orderings does this graph have?

3.4. Run the strongly connected components algorithm on the following directed graphs G. When doing DFS on G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

(i)



(ii)



In each case answer the following questions.

(a) In what order are the strongly connected components (SCCs) found?

(b) Which are source SCCs and which are sink SCCs?

(c) Draw the "metagraph" (each meta-node is an SCC of G).

(d) What is the minimum number of edges you must add to this graph to make it strongly connected?

3.5. The *reverse* of a directed graph $G = (V, E)$ is another directed graph $G^R = (V, E^R)$ on the same vertex set, but with all edges reversed; that is, $E^R = \{(v, u) : (u, v) \in E\}$.

Give a linear-time algorithm for computing the reverse of a graph in adjacency list format.

3.6. In an undirected graph, the *degree* $d(u)$ of a vertex u is the number of neighbors u has, or equivalently, the number of edges incident upon it. In a directed graph, we distinguish between the *indegree* $d_{in}(u)$, which is the number of edges into u, and the *outdegree* $d_{out}(u)$, the number of edges leaving u.

(a) Show that in an undirected graph, $\sum_{u \in V} d(u) = 2|E|$.

(b) Use part (a) to show that in an undirected graph, there must be an even number of vertices whose degree is odd.

(c) Does a similar statement hold for the number of vertices with odd indegree in a directed graph?

3.7. A *bipartite graph* is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices in the same set (for instance, if $u, v \in V_1$, then there is no edge between u and v).

(a) Give a linear-time algorithm to determine whether an undirected graph is bipartite.

(b) There are many other ways to formulate this property. For instance, an undirected graph is bipartite if and only if it can be colored with just two colors.

Prove the following formulation: an undirected graph is bipartite if and only if it contains no cycles of odd length.

(c) At most how many colors are needed to color in an undirected graph with exactly *one* odd-length cycle?

3.8. *Pouring water.* We have three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container.

(a) Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.

(b) What algorithm should be applied to solve the problem?

(c) Find the answer by applying the algorithm.

3.9. For each node u in an undirected graph, let `twodegree`$[u]$ be the sum of the degrees of u's neighbors. Show how to compute the entire array of `twodegree`$[\cdot]$ values in linear time, given a graph in adjacency list format.

3.10. Rewrite the `explore` procedure (Figure 3.3) so that it is non-recursive (that is, explicitly use a stack). The calls to `previsit` and `postvisit` should be positioned so that they have the same effect as in the recursive procedure.

3.11. Design a linear-time algorithm which, given an undirected graph G and a particular edge e in it, determines whether G has a cycle containing e.

3.12. Either prove or give a counterexample: if $\{u, v\}$ is an edge in an undirected graph, and during depth-first search `post`$(u) <$`post`(v), then v is an ancestor of u in the DFS tree.

3.13. *Undirected vs. directed connectivity.*

(a) Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ whose removal leaves G connected. (*Hint:* Consider the DFS search tree for G.)

(b) Give an example of a strongly connected directed graph $G = (V, E)$ such that, for every $v \in V$, removing v from G leaves a directed graph that is not strongly connected.

(c) In an undirected graph with 2 connected components it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.

3.14. The chapter suggests an alternative algorithm for linearization (topological sorting), which repeatedly removes source nodes from the graph (page 101). Show that this algorithm can be implemented in linear time.

3.15. The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough time to run a *linear-time* algorithm.

(a) Formulate this problem graph-theoretically, and explain why it can indeed be solved in linear time.

(b) Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

3.16. Suppose a CS curriculum consists of n courses, all of them mandatory. The prerequisite graph G has a node for each course, and an edge from course v to course w if and only if v is a prerequisite for w. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum (assume that a student can take any number of courses in one semester). The running time of your algorithm should be linear.

3.17. *Infinite paths.* Let $G = (V, E)$ be a directed graph with a designated "start vertex" $s \in V$, a set $V_G \subseteq V$ of "good" vertices, and a set $V_B \subseteq V$ of "bad" vertices. An *infinite* trace p of G is an infinite sequence $v_0 v_1 v_2 \cdots$ of vertices $v_i \in V$ such that (1) $v_0 = s$, and (2) for all $i \geq 0$, $(v_i, v_{i+1}) \in E$. That is, p is an infinite path in G starting at vertex s. Since the set V of vertices is finite, every infinite trace of G must visit some vertices infinitely often.

(a) If p is an infinite trace, let *Inf*$(p) \subseteq V$ be the set of vertices that occur infinitely often in p. Show that *Inf*(p) is a subset of a strongly connected component of G.

(b) Describe an algorithm that determines if G has an infinite trace.

(c) Describe an algorithm that determines if G has an infinite trace that visits some good vertex in V_G infinitely often.

(d) Describe an algorithm that determines if G has an infinite trace that visits some good vertex in V_G infinitely often, but visits no bad vertex in V_B infinitely often.

3.18. You are given a binary tree $T = (V, E)$ (in adjacency list format), along with a designated root node $r \in V$. Recall that u is said to be an *ancestor* of v in the rooted tree, if the path from r to v in T passes through u.

You wish to preprocess the tree so that queries of the form "is u an ancestor of v?" can be answered in constant time. The preprocessing itself should take linear time. How can this be done?

3.19. As in the previous problem, you are given a binary tree $T = (V, E)$ with designated root node. In addition, there is an array $x[\cdot]$ with a value for each node in V. Define a new array $z[\cdot]$ as follows: for each $u \in V$,

$$z[u] = \text{the maximum of the } x\text{-values associated with } u\text{'s descendants.}$$

Give a linear-time algorithm which calculates the entire z-array.

3.20. You are given a tree $T = (V, E)$ along with a designated root node $r \in V$. The *parent* of any node $v \neq r$, denoted $p(v)$, is defined to be the node adjacent to v in the path from r to v. By convention, $p(r) = r$. For $k > 1$, define $p^k(v) = p^{k-1}(p(v))$ and $p^1(v) = p(v)$ (so $p^k(v)$ is the kth ancestor of v).

Each vertex v of the tree has an associated non-negative integer label $l(v)$. Give a linear-time algorithm to update the labels of all the vertices in T according to the following rule: $l_{new}(v) = l(p^{l(v)}(v))$.

3.21. Give a linear-time algorithm to find an odd-length cycle in a *directed* graph. (*Hint:* First solve this problem under the assumption that the graph is strongly connected.)

3.22. Give an efficient algorithm which takes as input a directed graph $G = (V, E)$, and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable.

3.23. Give an efficient algorithm that takes as input a directed acyclic graph $G = (V, E)$, and two vertices $s, t \in V$, and outputs the number of different directed paths from s to t in G.

3.24. Give a linear-time algorithm for the following task.

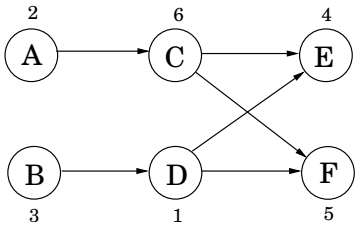
Input: A directed acyclic graph G

Question: Does G contain a directed path that touches every vertex exactly once?

3.25. You are given a directed graph in which each node $u \in V$ has an associated *price* p_u which is a positive integer. Define the array `cost` as follows: for each $u \in V$,

$$\texttt{cost}[u] = \text{price of the cheapest node reachable from } u \text{ (including } u \text{ itself).}$$

For instance, in the graph below (with prices shown for each vertex), the `cost` values of the nodes A, B, C, D, E, F are $2, 1, 4, 1, 4, 5$, respectively.



Your goal is to design an algorithm that fills in the *entire* `cost` array (i.e., for all vertices).

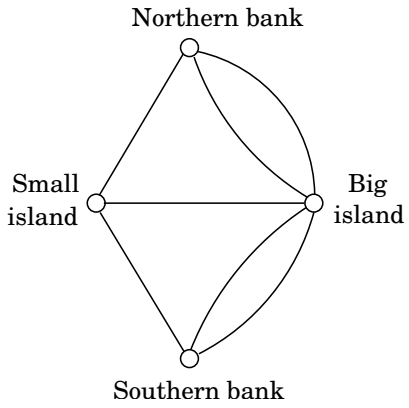
(a) Give a linear-time algorithm that works for directed *acyclic* graphs. (*Hint:* Handle the vertices in a particular *order*.)

(b) Extend this to a linear-time algorithm that works for all directed graphs. (*Hint:* Recall the "two-tiered" structure of directed graphs.)

3.26. An *Eulerian tour* in an undirected graph is a cycle that is allowed to pass through each vertex multiple times, but must use each edge exactly once.

This simple concept was used by Euler in 1736 to solve the famous Konigsberg bridge problem, which launched the field of graph theory. The city of Konigsberg (now called Kaliningrad, in western Russia) is the meeting point of two rivers with a small island in the middle. There are seven bridges across the rivers, and a popular recreational question of the time was to determine whether it is possible to perform a tour in which each bridge is crossed *exactly once*.

Euler formulated the relevant information as a graph with four nodes (denoting land masses) and seven edges (denoting bridges), as shown here.



Notice an unusual feature of this problem: multiple edges between certain pairs of nodes.

(a) Show that an undirected graph has an Eulerian tour if and only if all its vertices have even degree. Conclude that there is no Eulerian tour of the Konigsberg bridges.

(b) An *Eulerian path* is a path which uses each edge exactly once. Can you give a similar if-and-only-if characterization of which undirected graphs have Eulerian paths?

(c) Can you give an analog of part (a) for *directed* graphs?

3.27. Two paths in a graph are called *edge-disjoint* if they have no edges in common. Show that in any undirected graph, it is possible to pair up the vertices of odd degree and find paths between each such pair so that all these paths are edge-disjoint.

3.28. In the 2SAT problem, you are given a set of *clauses*, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value `true` or `false` to each of the variables so that *all* clauses are satisfied – that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \overline{x}_2) \ \wedge \ (\overline{x}_1 \vee \overline{x}_3) \ \wedge \ (x_1 \vee x_2) \ \wedge \ (\overline{x}_3 \vee x_4) \ \wedge \ (\overline{x}_1 \vee x_4).$$

This instance has a satisfying assignment: set x_1, x_2, x_3, and x_4 to `true`, `false`, `false`, and `true`, respectively.

(a) Are there other satisfying truth assignments of this 2SAT formula? If so, find them all.

(b) Give an instance of 2SAT with four variables, and with no satisfying assignment.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance I of 2SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes, one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I (where α, β are literals), G_I has an edge from from the negation of α to β, and one from the negation of β to α.

Note that the clause $(\alpha \vee \beta)$ is equivalent to either of the implications $\overline{\alpha} \Rightarrow \beta$ or $\overline{\beta} \Rightarrow \alpha$. In this sense, G_I records all implications in I.

(c) Carry out this construction for the instance of 2SAT given above, and for the instance you constructed in (b).

(d) Show that if G_I has a strongly connected component containing both x and $\overline{x}$ for some variable x, then I has no satisfying assignment.

(e) Now show the converse of (d): namely, that if none of G_I's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable. (*Hint:* Assign values to the variables as follows: repeatedly pick a sink strongly connected component of G_I. Assign value `true` to all literals in the sink, assign `false` to their negations, and delete all of these. Show that this ends up discovering a satisfying assignment.)

(f) Conclude that there is a linear-time algorithm for solving 2SAT.

3.29. Let S be a finite set. A binary *relation* on S is simply a collection R of ordered pairs $(x, y) \in S \times S$. For instance, S might be a set of people, and each such pair $(x, y) \in R$ might mean "x knows y."

An *equivalence relation* is a binary relation which satisfies three properties:

- Reflexivity: $(x, x) \in R$ for all $x \in S$
- Symmetry: if $(x, y) \in R$ then $(y, x) \in R$
- Transitivity: if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$

For instance, the binary relation "has the same birthday as" is an equivalence relation, whereas "is the father of" is not, since it violates all three properties.

Show that an equivalence relation partitions set S into disjoint groups $S_1, S_2, \ldots, S_k$ (in other words, $S = S_1 \cup S_2 \cup \cdots \cup S_k$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$) such that:

- Any two members of a group are related, that is, $(x, y) \in R$ for any $x, y \in S_i$, for any i.
- Members of different groups are not related, that is, for all $i \neq j$, for all $x \in S_i$ and $y \in S_j$, we have $(x, y) \notin R$.

(*Hint:* Represent an equivalence relation by an undirected graph.)

3.30. On page 102, we defined the binary relation "connected" on the set of vertices of a *directed* graph. Show that this is an equivalence relation (see Exercise 3.29), and conclude that it partitions the vertices into disjoint strongly connected components.

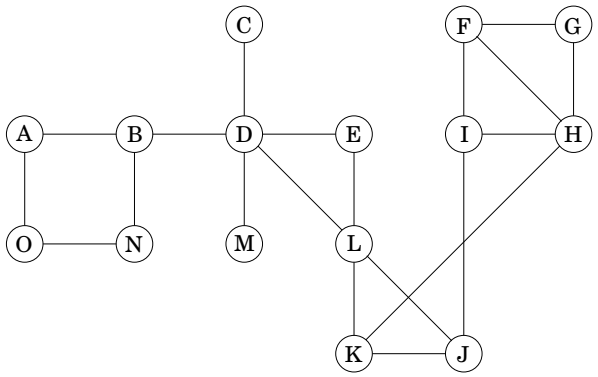
3.31. *Biconnected components* Let $G = (V, E)$ be an undirected graph. For any two edges $e, e' \in E$, we'll say $e \sim e'$ if either $e = e'$ or there is a (simple) cycle containing both e and e'.

(a) Show that $\sim$ is an equivalence relation (recall Exercise 3.29) on the edges.

The equivalence classes into which this relation partitions the edges are called the *biconnected components* of G. A *bridge* is an edge which is in a biconnected component all by itself.

A *separating vertex* is a vertex whose removal disconnects the graph.

(b) Partition the edges of the graph below into biconnected components, and identify the bridges and separating vertices.



Not only do biconnected components partition the edges of the graph, they also *almost* partition the vertices in the following sense.

(c) Associate with each biconnected component all the vertices that are endpoints of its edges. Show that the vertices corresponding to two different biconnected components are either disjoint or intersect in a single separating vertex.

(d) Collapse each biconnected component into a single meta-node, and retain individual nodes for each separating vertex. (So there are edges between each component-node and its separating vertices.) Show that the resulting graph is a tree.

DFS can be used to identify the biconnected components, bridges, and separating vertices of a graph in linear time.

(e) Show that the root of the DFS tree is a separating vertex if and only if it has more than one child in the tree.

(f) Show that a non-root vertex v of the DFS tree is a separating vertex if and only if it has a child v' none of whose descendants (including itself) has a backedge to a proper ancestor of v.

(g) For each vertex u define:

$$\texttt{low}(u) = \min \begin{cases} \texttt{pre}(u) \\ \texttt{pre}(w) \end{cases} \quad \text{where } (v, w) \text{ is a backedge for some descendant } v \text{ of } u$$

Show that the entire array of `low` values can be computed in linear time.

(h) Show how to compute all separating vertices, bridges, and biconnected components of a graph in linear time. (*Hint:* Use `low` to identify separating vertices, and run another DFS with an extra stack of edges to remove biconnected components one at a time.)

Chapter 4

Paths in graphs

4.1 Distances

Depth-first search readily identifies all the vertices of a graph that can be reached from a designated starting point. It also finds explicit paths to these vertices, summarized in its search tree (Figure 4.1). However, these paths might not be the most economical ones possible. In the figure, vertex C is reachable from S by traversing just one edge, while the DFS tree shows a path of length 3. This chapter is about algorithms for finding *shortest paths* in graphs.

Path lengths allow us to talk quantitatively about the extent to which different vertices of a graph are separated from each other:

The *distance* between two nodes is the length of the shortest path between them.

To get a concrete feel for this notion, consider a physical realization of a graph that has a ball for each vertex and a piece of string for each edge. If you lift the ball for vertex s high enough, the other balls that get pulled up along with it are precisely the vertices reachable from s. And to find their distances from s, you need only measure how far below s they hang.

Figure 4.1 (a) A simple graph and (b) its depth-first search tree.

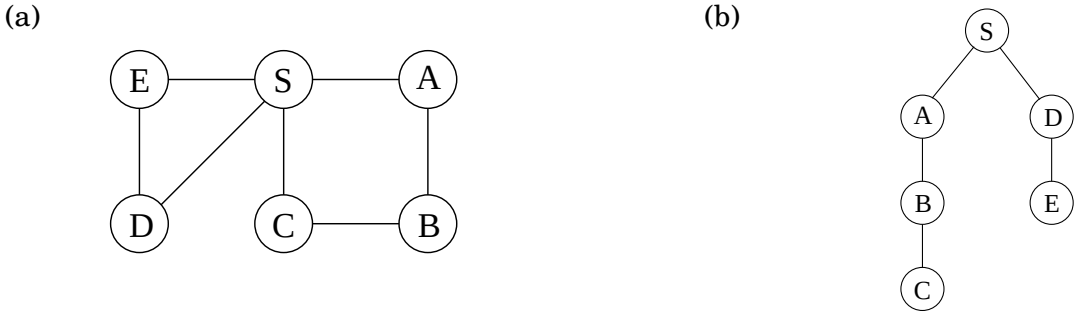
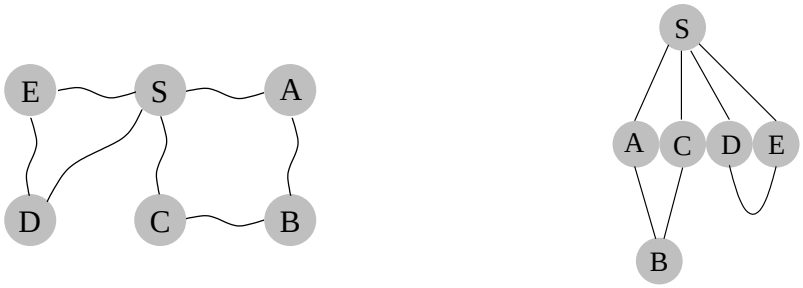


Figure 4.2 A physical model of a graph.



In Figure 4.2 for example, vertex B is at distance 2 from S, and there are two shortest paths to it. When S is held up, the strings along each of these paths become taut. On the other hand, edge (D, E) plays no role in any shortest path and therefore remains slack.

4.2 Breadth-first search

In Figure 4.2, the lifting of s partitions the graph into layers: s itself, the nodes at distance 1 from it, the nodes at distance 2 from it, and so on. A convenient way to compute distances from s to the other vertices is to proceed layer by layer. Once we have picked out the nodes at distance $0, 1, 2, \ldots, d$, the ones at $d + 1$ are easily determined: they are precisely the as-yet-unseen nodes that are adjacent to the layer at distance d. This suggests an iterative algorithm in which two layers are active at any given time: some layer d, which has been fully identified, and $d + 1$, which is being discovered by scanning the neighbors of layer d.

Breadth-first search (BFS) directly implements this simple reasoning (Figure 4.3). Initially the queue Q consists only of s, the one node at distance 0. And for each subsequent distance $d = 1, 2, 3, \ldots$, there is a point in time at which Q contains all the nodes at distance d and nothing else. As these nodes are processed (ejected off the front of the queue), their as-yet-unseen neighbors are injected into the end of the queue.

Let's try out this algorithm on our earlier example (Figure 4.1) to confirm that it does the right thing. If S is the starting point and the nodes are ordered alphabetically, they get visited in the sequence shown in Figure 4.4. The breadth-first search tree, on the right, contains the edges through which each node is initially discovered. Unlike the DFS tree we saw earlier, it has the property that all its paths from S are the shortest possible. It is therefore a *shortest-path tree*.

Correctness and efficiency

We have developed the basic intuition behind breadth-first search. In order to check that the algorithm works correctly, we need to make sure that it faithfully executes this intuition. What we expect, precisely, is that

For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$

Figure 4.3 Breadth-first search.

```
procedure bfs(G, s)
Input:    Graph G = (V, E), directed or undirected; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
   u = eject(Q)
   for all edges (u, v) ∈ E:
      if dist(v) = ∞:
         inject(Q, v)
         dist(v) = dist(u) + 1
```

from s have their distances correctly set; (2) all other nodes have their distances set to ∞; and (3) the queue contains exactly the nodes at distance d.

This has been phrased with an inductive argument in mind. We have already discussed both the base case and the inductive step. Can you fill in the details?

The overall running time of this algorithm is linear, $O(|V| + |E|)$, for exactly the same reasons as depth-first search. Each vertex is put on the queue exactly once, when it is first encountered, so there are $2|V|$ queue operations. The rest of the work is done in the algorithm's innermost loop. Over the course of execution, this loop looks at each edge once (in directed graphs) or twice (in undirected graphs), and therefore takes $O(|E|)$ time.

Now that we have both BFS and DFS before us: how do their exploration styles compare? Depth-first search makes deep incursions into a graph, retreating only when it runs out of new nodes to visit. This strategy gives it the wonderful, subtle, and extremely useful properties we saw in the Chapter 3. But it also means that DFS can end up taking a long and convoluted route to a vertex that is actually very close by, as in Figure 4.1. Breadth-first search makes sure to visit vertices in increasing order of their distance from the starting point. This is a broader, shallower search, rather like the propagation of a wave upon water. And it is achieved using almost exactly the same code as DFS—but with a queue in place of a stack.

Also notice one stylistic difference from DFS: since we are only interested in distances from s, we do not restart the search in other connected components. Nodes not reachable from s are simply ignored.

Figure 4.4 The result of breadth-first search on the graph of Figure 4.1.

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$

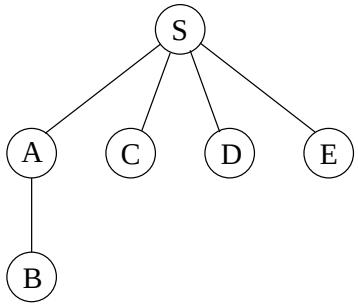
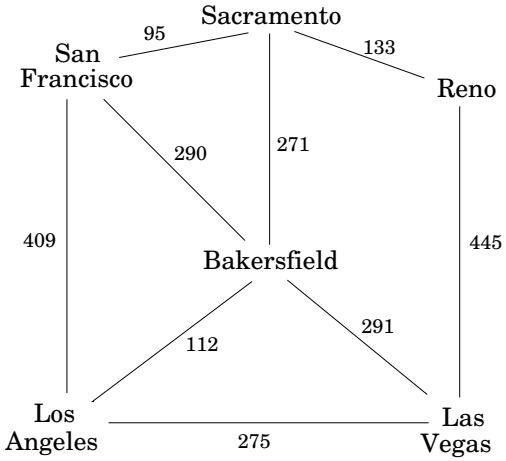


Figure 4.5 Edge lengths often matter.

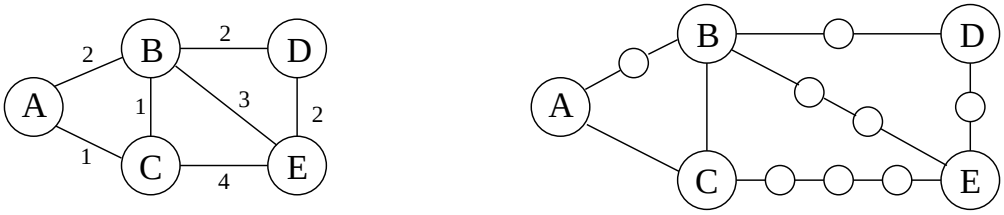


4.3 Lengths on edges

Breadth-first search treats all edges as having the same length. This is rarely true in applications where shortest paths are to be found. For instance, suppose you are driving from San Francisco to Las Vegas, and want to find the quickest route. Figure 4.5 shows the major highways you might conceivably use. Picking the right combination of them is a shortest-path problem in which the length of each edge (each stretch of highway) is important. For the remainder of this chapter, we will deal with this more general scenario, annotating every edge $e \in E$ with a length l_e. If $e = (u, v)$, we will sometimes also write $l(u, v)$ or l_{uv}.

These l_e's do not have to correspond to physical lengths. They could denote time (driving time between cities) or money (cost of taking a bus), or any other quantity that we would like to conserve. In fact, there are cases in which we need to use negative lengths, but we will briefly overlook this particular complication.

Figure 4.6 Breaking edges into unit-length pieces.



4.4 Dijkstra's algorithm

4.4.1 An adaptation of breadth-first search

Breadth-first search finds shortest paths in any graph whose edges have unit length. Can we adapt it to a more general graph $G = (V, E)$ whose edge lengths l_e are *positive integers*?

A more convenient graph

Here is a simple trick for converting G into something BFS can handle: break G's long edges into unit-length pieces, by introducing "dummy" nodes. Figure 4.6 shows an example of this transformation. To construct the new graph G',

> For any edge $e = (u, v)$ of E, replace it by l_e edges of length 1, by adding $l_e - 1$ dummy nodes between u and v.

Graph G' contains all the vertices V that interest us, and the distances between them are exactly the same as in G. Most importantly, the edges of G' all have unit length. Therefore, we can compute distances in G by running BFS on G'.

Alarm clocks

If efficiency were not an issue, we could stop here. But when G has very long edges, the G' it engenders is thickly populated with dummy nodes, and the BFS spends most of its time diligently computing distances to these nodes that we don't care about at all.

To see this more concretely, consider the graphs G and G' of Figure 4.7, and imagine that the BFS, started at node s of G', advances by one unit of distance per minute. For the first 99 minutes it tediously progresses along $S - A$ and $S - B$, an endless desert of dummy nodes. Is there some way we can snooze through these boring phases and have an alarm wake us up whenever something *interesting* is happening—specifically, whenever one of the real nodes (from the original graph G) is reached?

We do this by setting two alarms at the outset, one for node A, set to go off at time $T = 100$, and one for B, at time $T = 200$. These are *estimated times of arrival*, based upon the edges currently being traversed. We doze off and awake at $T = 100$ to find A has been discovered. At

this point, the estimated time of arrival for B is adjusted to $T = 150$ and we change its alarm accordingly.

More generally, at any given moment the breadth-first search is advancing along certain edges of G, and there is an alarm for every endpoint node toward which it is moving, set to go off at the estimated time of arrival at that node. Some of these might be overestimates because BFS may later find shortcuts, as a result of future arrivals elsewhere. In the preceding example, a quicker route to B was revealed upon arrival at A. However, *nothing interesting can possibly happen before an alarm goes off*. The sounding of the next alarm must therefore signal the arrival of the wavefront to a real node $u \in V$ by BFS. At that point, BFS might also start advancing along some new edges out of u, and alarms need to be set for their endpoints.

The following "alarm clock algorithm" faithfully simulates the execution of BFS on G'.

- Set an alarm clock for node s at time 0.
- Repeat until there are no more alarms:

  Say the next alarm goes off at time T, for node u. Then:

  - The distance from s to u is T.
  - For each neighbor v of u in G:
    - If there is no alarm yet for v, set one for time $T + l(u, v)$.
    - If v's alarm is set for later than $T + l(u, v)$, then reset it to this earlier time.

Dijkstra's algorithm. The alarm clock algorithm computes distances in any graph with positive integral edge lengths. It is almost ready for use, except that we need to somehow implement the system of alarms. The right data structure for this job is a *priority queue* (usually implemented via a *heap*), which maintains a set of elements (nodes) with associated numeric key values (alarm times) and supports the following operations:

Insert. Add a new element to the set.

Decrease-key. Accommodate the decrease in key value of a particular element.[1]

[1] The name *decrease-key* is standard but is a little misleading: the priority queue typically does not itself change key values. What this procedure really does is to notify the queue that a certain key value has been decreased.

Figure 4.7 BFS on G' is mostly uneventful. The dotted lines show some early "wavefronts."