1 Review

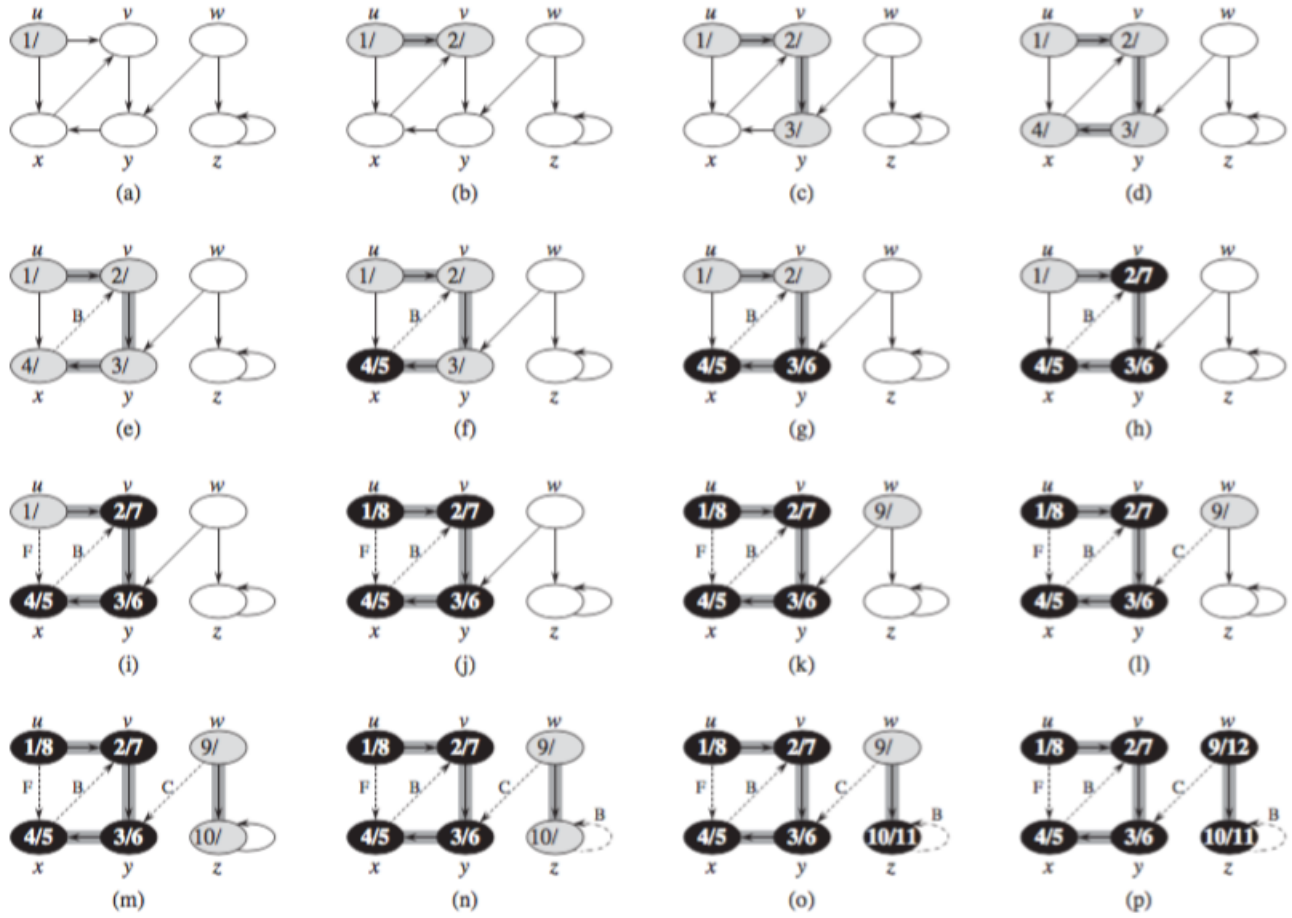


Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1            // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]      // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK            // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

Something I did not emphasize enough last time is that during the execution of depth-first-search, we construct **depth-first-search trees**. One graph may have multiple depth-first-search trees, and each tree contains the vertices that are reachable from a given start node. The edges in the depth-first-search tree are a subset of the edges in the graph, and an edge gets created from u to v in the depth-first-search tree if we are examining u's neighbors and discover that its neighbor v has not been visited yet. (Note that if v has already been reached through other means, the edge is omitted from the tree.)

Most of the time, when I referred to the "parent" of a node, I meant the parent in the depth-first-search tree. Each node only has one parent (so it's not just any node that points to it, it refers to a very specific node). Also, when I referred to the "descendants" of a node, I meant the nodes that are reachable from that node if you follow paths in the depth-first-search tree.

Parenthesis theorem: The descendants in a depth-first-search tree have an interesting property. If v is a descendant of u, then the discovery time of v is later than the discovery time of u. However, the finishing time of v is earlier than the finishing time of u. You can see this using the recursive call structure of DFS-Visit – first we call DFS-Visit on u, and then we recurse on its descendants, and the inner recursion must start after the outer recursion, but finish before the outer recursion.

If v is not a descendant of u and u is not a descendant of v, then the interval $[u.d, u.f]$ is disjoint from the interval $[v.d, v.f]$. Presumably when we are dealing with the first vertex, we are too busy looking at its descendants to even begin thinking about other parts of the graph until we are done.

White path theorem: Recall that v is a descendant of u in the depth-first-search tree if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

2 Strongly connected components

Definition: An undirected graph is connected if every vertex is reachable from all other vertices. That is, for every pair of vertices $u, v \in V$, there is a path from u to v (and therefore, a path from v to u).

Definition: A directed graph G is strongly connected if every two vertices are reachable from each other. That is, for every pair of vertices $u, v \in V$, there is both a path from u to v and a path from v to u.

Definition: A strongly connected component of a directed graph $G = (V, E)$ is a maximal part of the graph that is strongly connected. Formally, it is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, there is both a path from u to v and a path from v to u.

(By "maximal" I mean that no proper superset of the vertices forms a strongly connected component. However, note that a single graph may have multiple strongly connected components of different sizes.)

Definition: A connected component of an undirected graph is a maximal set of vertices where every vertex in the set is reachable from every other vertex in the set. That is, for every pair of vertices u, v in the connected component C, there is a path from u to v.

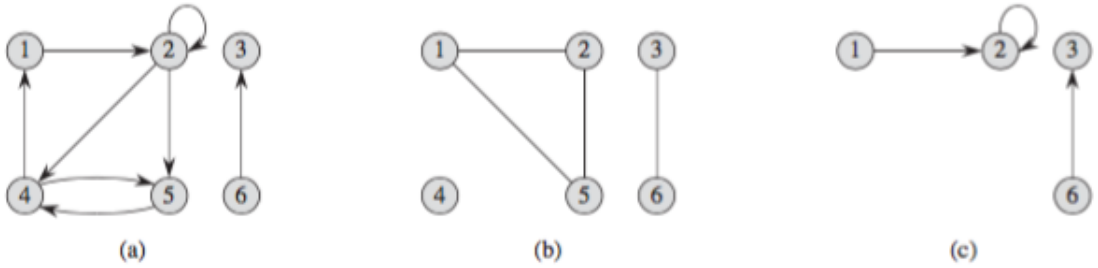


Figure B.2 Directed and undirected graphs. **(a)** A directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. The edge $(2, 2)$ is a self-loop. **(b)** An undirected graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. The vertex 4 is isolated. **(c)** The subgraph of the graph in part (a) induced by the vertex set $\{1, 2, 3, 6\}$.

Example: The graph in Figure B.2(a) has three strongly connected components: $\{1, 2, 4, 5\}$, $\{3\}$, and $\{6\}$. The graph in Figure B.2(b) has three connected components: $\{1, 2, 5\}$, $\{3, 6\}$, and $\{4\}$.

If we wanted to find the connected components in an undirected graph, we could simply run depth first search. Each depth first search tree consists of the vertices that are reachable from a given start node, and those nodes are all reachable from each other. However, finding the strongly connected components in a directed graph is not as easy, because u can be reachable from v without v being reachable from u.

2.1 Algorithm for finding strongly connected components

Suppose we decompose a graph into its strongly connected components, and build a new graph, where each strongly connected component is a "giant vertex." Then this new graph forms a directed acyclic graph, as in the figure.

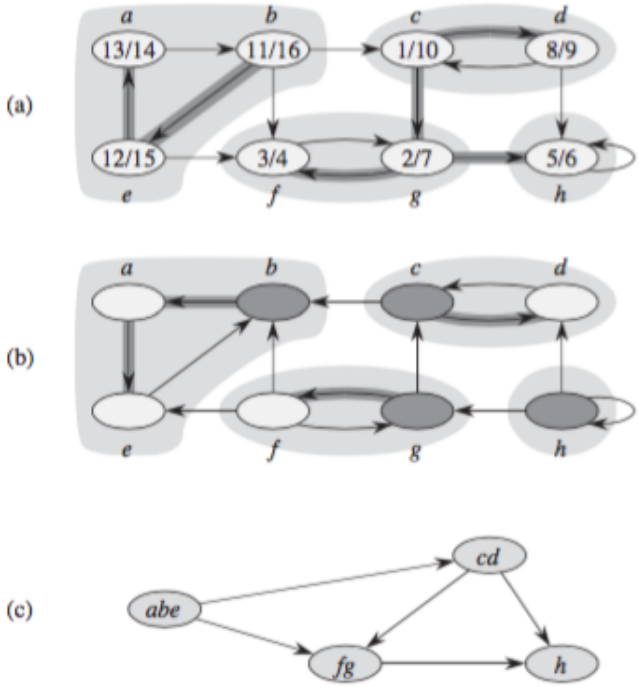


Figure 22.9 **(a)** A directed graph G. Each shaded region is a strongly connected component of G. Each vertex is labeled with its discovery and finishing times in a depth-first search, and tree edges are shaded. **(b)** The graph G^{T}, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^{T}. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

To see why the graph must be acyclic, suppose there was a cycle in the strongly connected component graph, $C_{i_1} \to C_{i_2} \to \cdots \to C_{i_k} \to C_{i_1}$. Then because

- Any vertex in C_{i_1} can be reached from any other vertex in C_{i_1}
- Any vertex in C_{i_2} can be reached from any other vertex in C_{i_2}
- There is an edge from some vertex in C_{i_1} to some vertex in C_{i_2}

It follows that any vertex in the component C_{i_2} can be reached from any other vertex in the component C_{i_1}. Similarly, any vertex in any of the components $C_{i_1}, \ldots, C_{i_k}$ can be reached from any other vertex in $C_{i_1}, \ldots, C_{i_k}$. So $C_{i_1} \cup \cdots \cup C_{i_k}$ is a strongly connected superset of C_{i_1}, which violates the assumption that C_{i_1} is a maximal strongly connected set of vertices.

2.1.1 Idea

So if we compute the finishing times of each vertex (using depth first search), and then examine the vertices in reverse order of finishing time, we will be examining the components in topologically sorted order. (The vertices themselves can't be topologically sorted, but the strongly connected components can.) Note: this is not obvious, and it's proven in the textbook.

Now suppose we reverse all the edges (which doesn't change the strongly connected components), and perform a (second) depth first search on the new graph, examining the vertices in reverse order of finishing time. Then every time we finish a strongly connected component, the recursion will abruptly halt, because by reversing the edges, we have created a "barrier" between that strongly connected component and the next one. Each depth-first-search tree produces exactly one strongly connected component. We can exploit this to return the strongly connected components.

2.1.2 Algorithm

STRONGLY-CONNECTED-COMPONENTS(G)
1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

2.1.3 Correctness

We are just proving a lemma here. For the full correctness proof, see the textbook.

If C is a component, we let $d(C) = \min_{u \in C}(u.d)$ be the earliest discovery time out of any vertex in C, and $f(C) = \max_{u \in C}(u.f)$ be the latest finishing time out of any vertex in C.

Lemma 22.14: Let C and C' be distinct strongly connected components in a directed graph $G = (V, E)$. Suppose there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

This lemma implies that if vertices are visited in reverse order of finishing time, then the components will be visited in topologically sorted order. That is, if there is an edge from component 1 to component 2, then component 1 will be visited before component 2.

Proof: Consider two cases.

1. $d(C) < d(C')$ (i.e. component C was discovered first). Let x be the first vertex discovered in C. Then at time $x.d$, all vertices in C and C' are white, so there is a path from x to each vertex in C consisting only of white vertices. There is also a path from

x to each vertex in C' consisting of only white vertices (because you can follow the edge (u, v)). By the white path theorem, all vertices in C and C' become descendants of x in the depth first search tree.

Now x has the latest finishing time out of any of its descendants, due to the "parenthesis theorem." However, you can see this if you look at the structure of the recursive calls – we call DFS-Visit on x, and then recurse on the nodes that will become its descendants in the depth-first-search tree. So the inner recursion has to finish before the outer recursion, which means x has the latest finishing time out of all the nodes in C and C'. Therefore, $x.f = f(C) > f(C')$.

2. $d(C) > d(C')$ (i.e. component C' was discovered first). Let y be the first vertex discovered in C'. At time $y.d$, all vertices in C' are white, and there is a path from y to each vertex in C' consisting only of white vertices. By the white path theorem, all vertices in C' are descendants of y in the depth-first-search tree, so $y.f = f(C')$.

   However, there is no path from C' to C (because we already have an edge from C to C', and if there was a path in the other direction, then C and C' would just be one component). Now all of the vertices in C are white at time $y.d$ (because $d(C) > d(C')$). And since no vertex in C is reachable from y, all the vertices in C must still be white at time $y.f$. Therefore, the finishing time of any vertex in C is greater than $y.f$, and $f(C) > f(C')$.

3 Breadth-first search

Breadth-first search is a method for searching through all the nodes in a graph that also takes $\Theta(m+n)$ time. It may also be used to find the shortest path (in an unweighted graph) from a given node to every other node. The idea is to start at a node, and then visit the neighbors of the node, and then visit the neighbors of its neighbors, etc until the entire graph is explored. All vertices close to the node are visited before vertices far from the node.

When we first discover a node, we keep track of its **parent** vertex, i.e. which vertex triggered the initial visit to that node. These parent-child relations form a **breadth-first search tree**, which contains a subset of the edges in the graph, and all the vertices reachable from the start node. The root is the start node, and the neighbors of the root become the root's children in the breadth-first-search-tree, and the neighbors of the neighbors become the children of the neighbors, etc. Some of the edges in the graph are not included in the breadth first search tree, because they point to vertices that were already seen on higher levels of the tree.

Using the breadth-first-search tree, we find paths from the start node to every other node in the graph. (We can prove that those paths are the shortest paths possible, and we will do this in class today.) We also keep track of the distance from the start node to every other node, which is basically how many levels away you are in the breadth-first-search tree.

3.0.1 Brief algorithm

In breadth-first search, we give each vertex a color: white ("unvisited"), grey ("discovered"), or black ("finished"). We start by placing the start node onto a queue (and marking it grey). On each iteration of the whole loop, we remove an element from the queue (marking it black), and add its neighbors to the queue (marking them grey). We do this until the queue has been exhausted. Note that we only add white ("unvisited") neighbors to the queue, so we only dequeue each vertex once.

Briefly, we

```
Mark all vertices white, except the start node s, which is grey.
Add s to an empty queue Q.
while Q is nonempty:
    node = Dequeue(Q)
    for each neighbor in Adj[node]:
        if neighbor.color is white:
            neighbor.color = gray
            Enqueue(Q, neighbor)
    node.color = black
```

You'll notice this pseudocode doesn't really do anything, aside from coloring the vertices. To use breadth-first search in real life, you would probably want to modify this code to do something every time you processed a vertex.

For example, if you wanted to find the average number of followers of users on Twitter, you could start at a user, and use breadth-first search to traverse the graph. Every time you mark a user black, you can count how many followers they have, and add this number to a database. (This actually is technically possible using the Twitter API, but it would take you forever because Twitter restricts the rate at which you can get data from their servers.)

You might also wonder what is the point of marking users black, because black and grey users are pretty much equivalent in the pseudocode. The main point is to make the correctness proof easier. Having three colors more closely mirrors the three states a vertex can be in, which makes things easier to reason about.

3.0.2 Full algorithm

But I haven't gotten to the part where we compute the shortest paths. To find the shortest paths, we use the full algorithm, which is shown here.

The full algorithm keeps two additional attributes for each node: d (which is the distance from the start node to that node), and π (which is the "parent" node, i.e., the node right before it on the shortest path). Whenever BFS discovers a new node v by way of a node u, we can prove that this was the "best way" of discovering v, so v's parent is u, and v's distance from the source node is one plus u's distance from the source node.

Note: in depth first search there is also a d attribute, but it means something different (in DFS it refers to the discovery time of a vertex, whereas in BFS it refers to the distance).

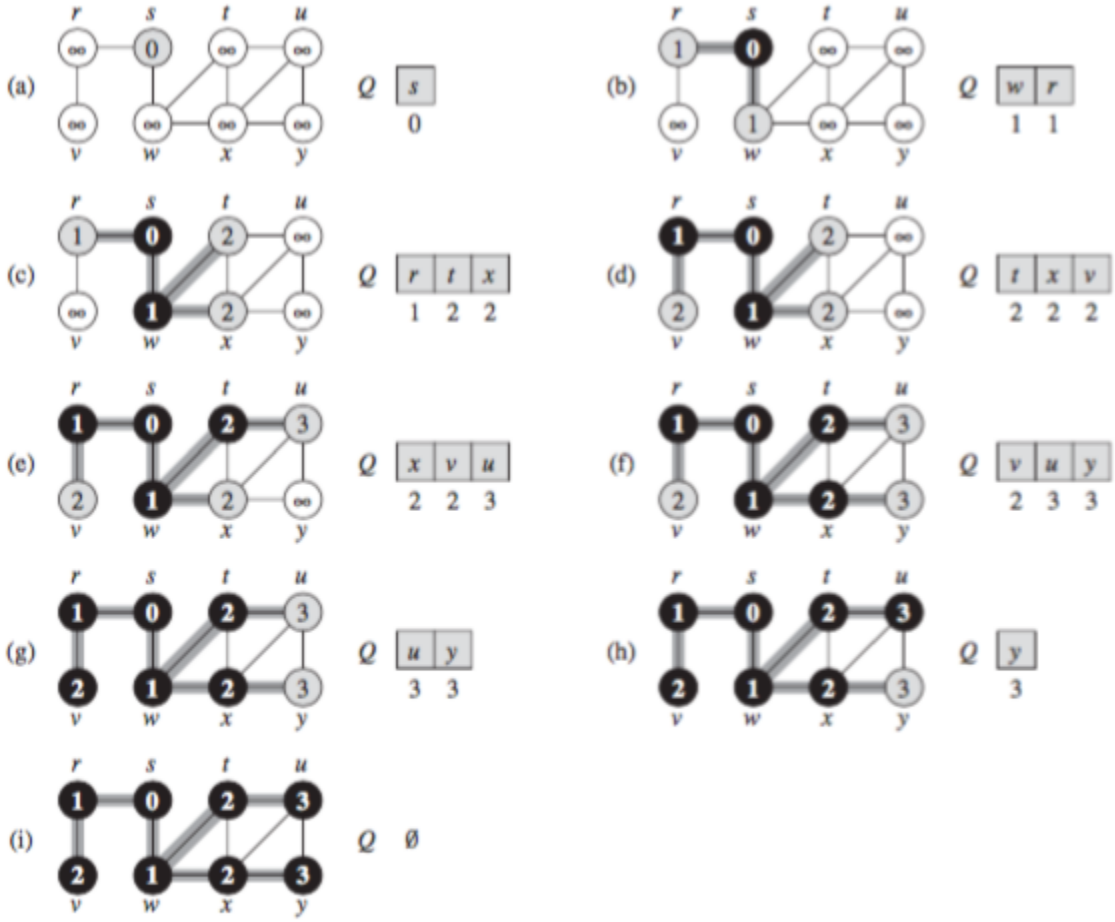


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex u. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the queue.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

3.0.3 Runtime

Breadth-first-search runs in $O(m + n)$. Because we only add white vertices to the queue (and mark them grey when they are added), and because a grey/black vertex never becomes white again, it follows that we only add a vertex to the queue at most once during the entire algorithm. So the total time taken by the queue operations is $O(n)$. The total number of iterations of the for loop is $O(m)$, because each adjacency list is iterated through at most once (i.e. when the corresponding vertex is removed from the queue), and there are m entries total if you aggregate over all of the adjacency lists.

4 Dijkstra's algorithm

While breadth-first-search computes shortest paths in an unweighted graph, Dijkstra's algorithm is a way of computing shortest paths in a weighted graph. Specifically Dijkstra's computes the shortest paths from a source node s to every other node in the graph.

The idea is that we keep "distance estimates" for every node in the graph (which are always greater than the true distance from the start node). On each iteration of the algorithm we process the (unprocessed) vertex with the smallest distance estimate. (We can prove that by the time we get around to processing a vertex, its distance estimate reflects the true distance to that vertex. This is nontrivial and must be proven.)

Whenever we process a vertex, we update the distance estimates of its neighbors, to account for the possibility that we may be reaching those neighbors through that vertex. Specifically, if we are processing u, and there is an edge from $u \to v$ with weight w, we change v's distance estimate $v.d$ to be the minimum of its current value and $u.d+w$. (It's possible that $v.d$ doesn't change at all, for example, if the shortest path from s to v was through a different vertex.)

If we did lower the estimate $v.d$, we set v's parent to be u, to signify that (we think) the best way to reach v is through u. The parent may change multiple times through the course of the algorithm, and at the end, the parent-child relations form a shortest path tree, where the path (along tree edges) from s to any node in the tree is a shortest path to that node. Note that the shortest path tree is very much like the breadth first search tree.

Important: Dijkstra's algorithm does not handle graphs with negative edge weights! For that you would need to use a different algorithm, such as Bellman-Ford.

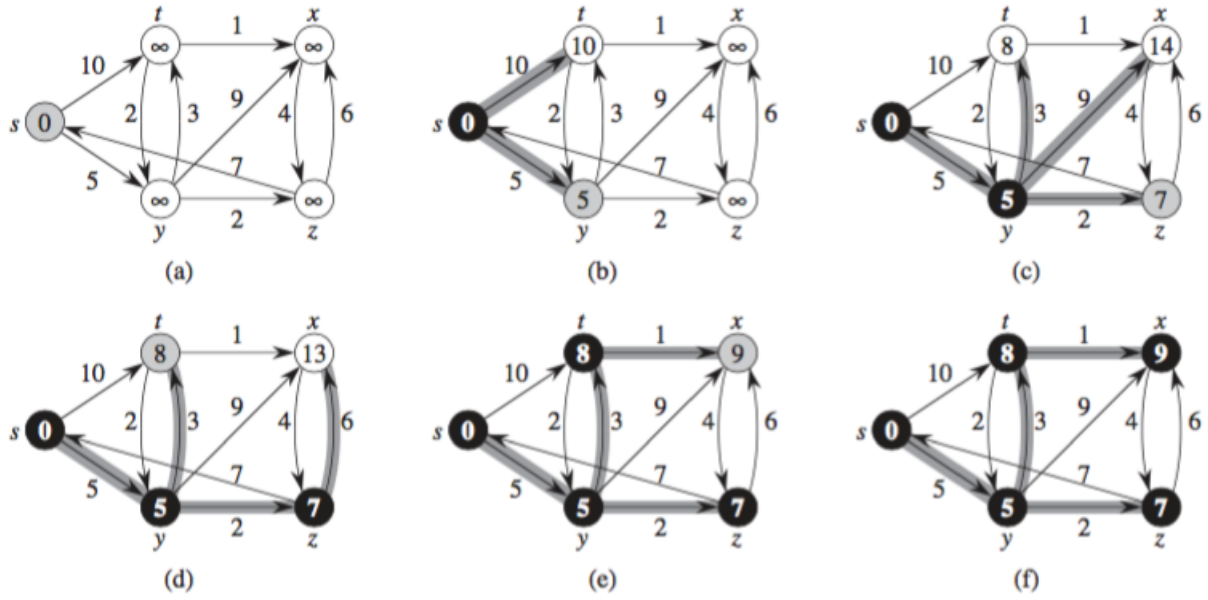


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

4.0.1 Runtime

The runtime of Dijkstra's algorithm depends on how we implement the priority queue Q. The bound you want to remember is $O(m + n \log n)$, which is what you get if Q is implemented as a Fibonacci heap.

The priority queue implements three operations: Insert (which happens when we build the queue), ExtractMin (which happens when we process the element with the lowest distance element), and DecreaseKey (which happens in the Relax function).

Insert gets called n times (as the queue is being built), ExtractMin is called n times (since each vertex is dequeued exactly once), and DecreaseKey is called m times (since the total number of edges in all the adjacency lists is m).

If you want a naive implementation, and do not want to bother with Fibonacci heaps, you can simply store the distance estimates of the vertices in an array. Assuming the vertices are labeled 1 to n, we can store $v.d$ in the vth entry of an array. Then Insert and DecreaseKey would take $O(1)$ time, and ExtractMin would take $O(n)$ time (since we are searching through the entire array. This produces a runtime of $O(n^2 + m) = O(n^2)$.

You can also implement the priority queue in a normal heap, which gives us ExtractMin and DecreaseKey in $O(\log n)$. The time to build the heap is $O(n)$. So the total runtime would be $O((n + m) \log n)$. This beats the naive implementation if the graph is sufficiently sparse.

In the Fibonacci heap, the amortized cost of each of the ExtractMin operations is $O(\log n)$, and each DecreaseKey operation takes $O(1)$ amortized time.

4.1 Correctness

A big thank you to Virginia Williams from last quarter for supplying this correctness proof so we don't have to use the one in the textbook.

Let s be the start node/source node, $v.d$ be the "distance estimate" of a vertex v, and $\delta(u, v)$ be the true distance from u to v. We want to prove two statements:

1. At any point in time, $v.d \geq \delta(s, v)$.

2. When v is extracted from the queue, $v.d = \delta(s, v)$. (Distance estimates never increase, so once $v.d = \delta(s, v)$, it stays that way.)

4.1.1 $v.d \geq \delta(s, v)$

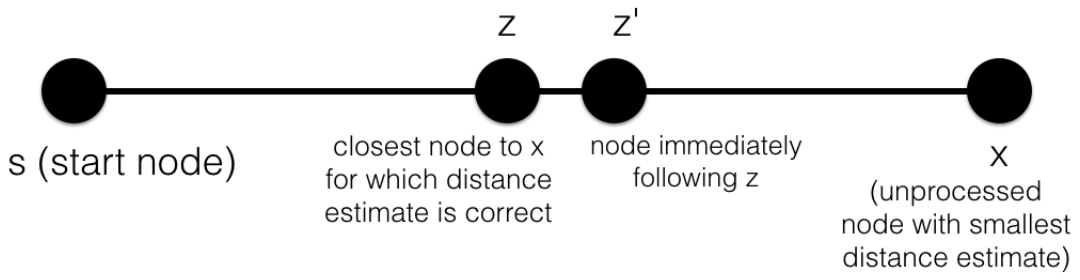
We want to show that at any point in time, if $v.d < \infty$ then $v.d$ is the weight of some path from s to v (not necessarily the shortest path). Since $\delta(s, v)$ is the weight of the shortest path from s to v, the conclusion will follow immediately.

We induct on the number of Relax operations. As our base case, we know that $s.d = 0 = \delta(s, s)$, and all other distance estimates are ∞, which is greater than or equal to their true distance.

As our inductive step, assume that at some point in time, every distance estimate corresponds to the weight of some path from s to that vertex. Now when a Relax operation is performed, the distance estimate of some neighbor $u.d$ may be changed to $x.d + w(x, u)$, for some vertex x. We know $x.d$ is the weight of some path from s to x. If we add the edge (x, u) at the end of that path, the weight of the resulting path is $x.d + w(x, u)$, which is $u.d$.

Alternatively, the distance estimate $u.d$ may not change at all during the Relax step. In that case we already know (from the inductive hypothesis) that $u.d$ is the weight of some path from s to u, so the inductive step is still satisfied.

4.1.2 $v.d = \delta(s, v)$ when v is extracted from the queue



We induct on the order in which we add nodes to S. For the base case, s is added to S when $s.d = \delta(s, s) = 0$, so the claim holds.

For the inductive step, assume that the claim holds for all nodes that are currently in S, and let x be the node in Q that currently has the minimum distance estimate. (This is the node that is about to be extracted from the queue.) We will show that $x.d = \delta(s, x)$.

Suppose p is a shortest path from s to x. Suppose z is the node on p closest to x for which $z.d = \delta(s, z)$. (We know z exists because there is at least one such node, namely s, where $s.d = \delta(s, s)$.) This means for every node y on the path p between z (not inclusive) and x (inclusive), we have $y.d > \delta(s, y)$.

If $z = x$, then $x.d = \delta(s, x)$, so we are done.

So suppose $z \neq x$. Then there is a node z' after z on p (which might equal x). We argue that $z.d = \delta(s, z) \leq \delta(s, x) \leq x.d$.

- $\delta(s, x) \leq x.d$ from the previous lemma, and $z.d = \delta(s, z)$ by assumption.
- $\delta(s, z) \leq \delta(s, x)$ because subpaths of shortest paths are also shortest paths. That is, if $s \to \cdots \to z \to \cdots \to x$ is a shortest path to x, then the subpath $s \to \cdots \to z$ is a shortest path to z. This is because, suppose there were an alternate path from s to z that was shorter. Then we could "glue that path into" the path from s to x, producing a shorter path and contradicting the fact that the s-to-x path was a shortest path.

Now we want to collapse these inequalities into equalities, by proving that $z.d = x.d$. Assume (by way of contradiction) that $z.d < x.d$. Because $x.d$ has the minimum distance estimate out of all the unprocessed vertices, it follows that z has already been added to S. This means that all of the edges coming out of z have already been relaxed by our algorithm, which means that $z'.d \leq \delta(s, z) + w(z, z') = \delta(s, z')$. (This last equality holds because z precedes z' on the shortest path to x, so z is on the shortest path to z'.)

However, this contradicts the assumption that z is the closest node on the path to x with a correct distance estimate. Thus, $z.d = x.d$.