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 $u \rightarrow v$ 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.

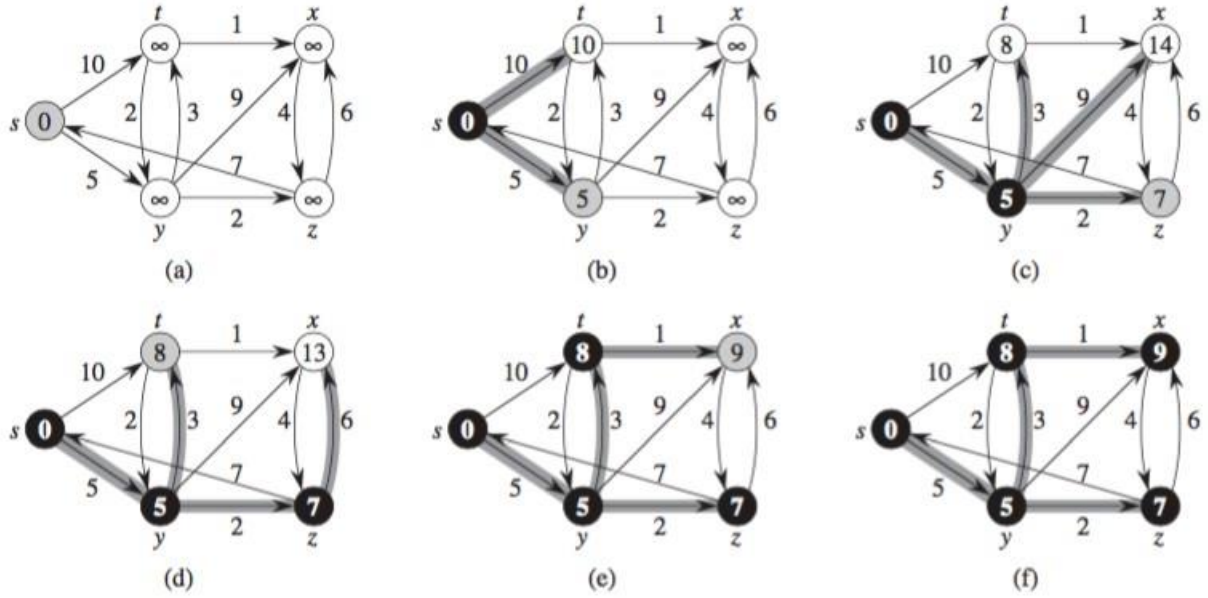


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation 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 $v.d \geq \delta(s,v)$

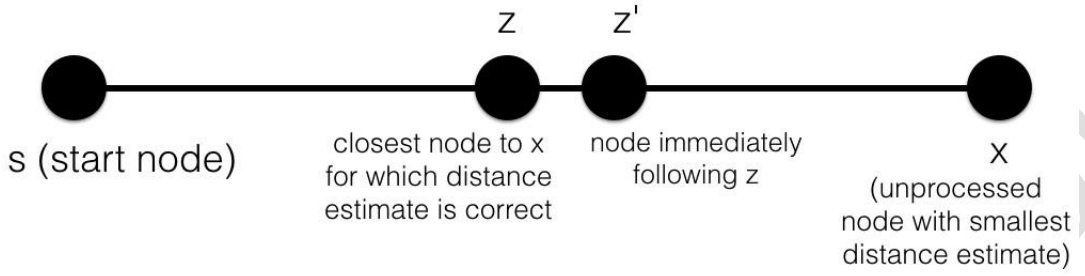
We want to show that at any point in time, if $v.d < \infty$ then $v.d$ is the weight of some path from s to v (not necessarily the shortest path). Since $\delta(s,v)$ is the weight of the shortest path from s to v, the conclusion will follow immediately.

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 6= x. Then there is a node z^0 after *z* on *p* (which might equal *x*). We argue that $z.d = \delta(s,z) \le \delta(s,x) \le x.d$.

- $\delta(s,x) \le x.d$ from the previous lemma, and $z.d = \delta(s,z)$ by assumption.
- $\delta(s,z) \le \delta(s,x)$ 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 $z^0.d \le \delta(s,z) + w(z,z^0) = \delta(s,z^0)$. (This last equality holds because *z* precedes z^0 on the shortest path to *x*, so *z* is on the shortest path to z^0.)

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