Single-Source Shortest Paths

Introduction

Many problems can be modeled using graphs with weights assigned to their edges.

For example, they can be used to model an airline system.

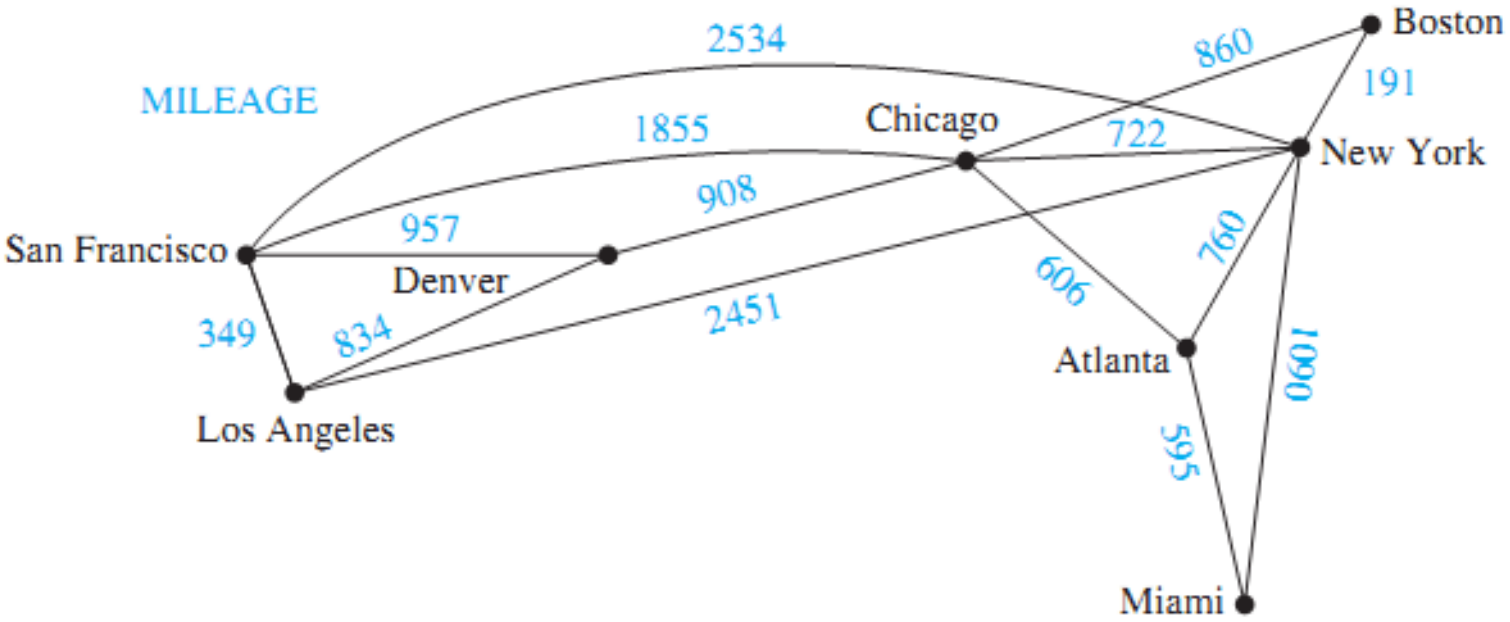
We create the basic graph model by representing cities by vertices and flights by edges.

Problems involving distances can be modeled by assigning distances between cities to the edges.

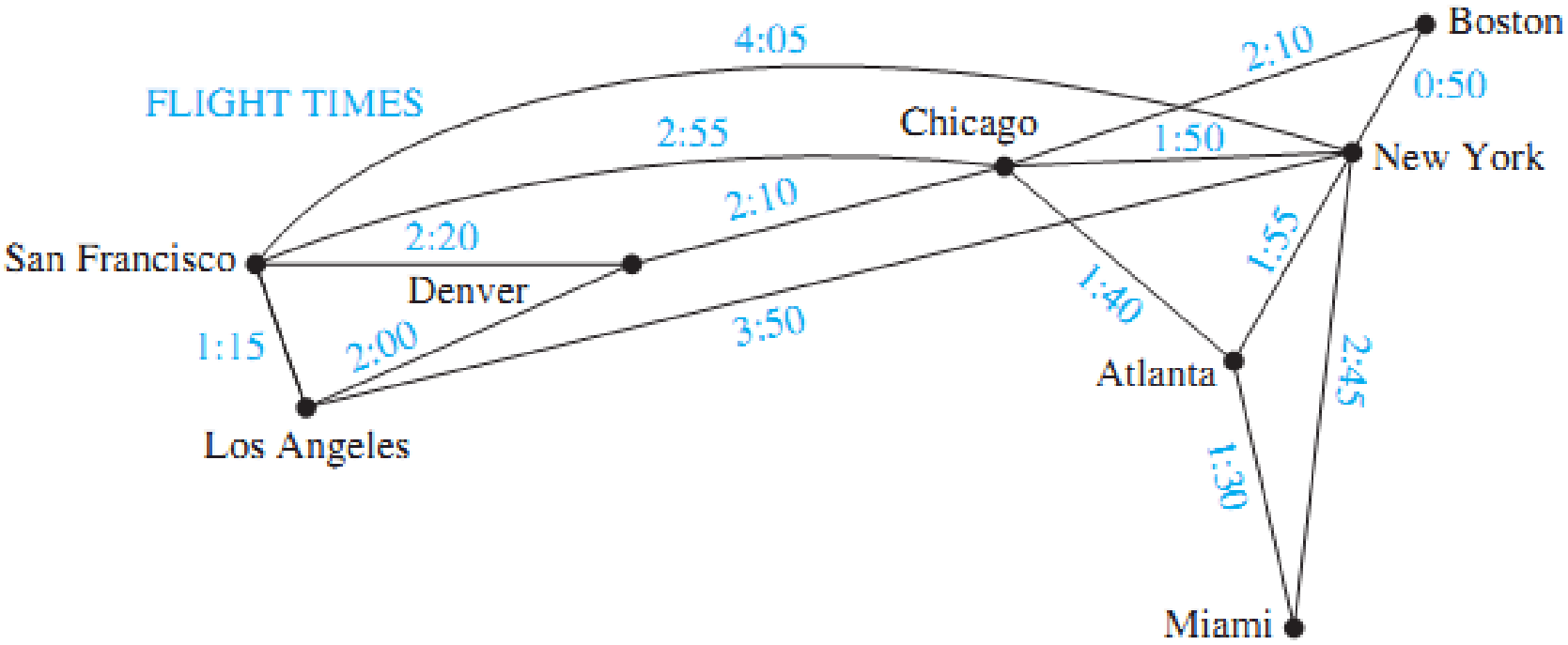
Problems involving flight time can be modeled by assigning flight times to edges.

Problems involving fares can be modeled by assigning fares to the edges.

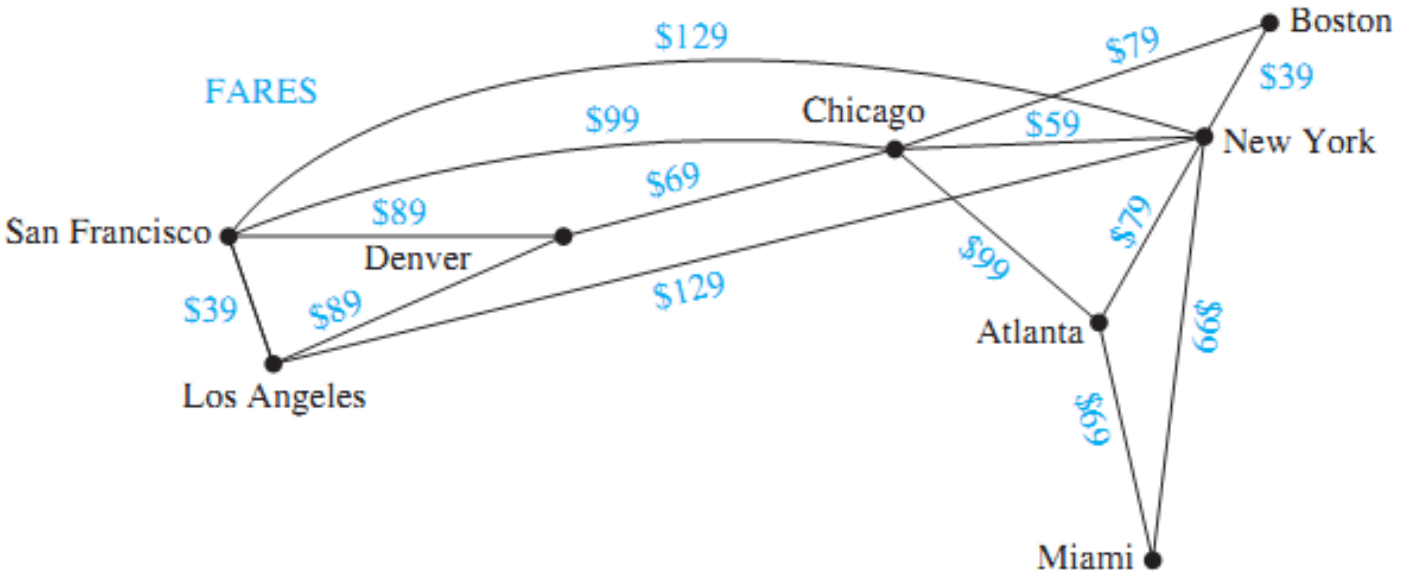
Distances between cities



Flight times



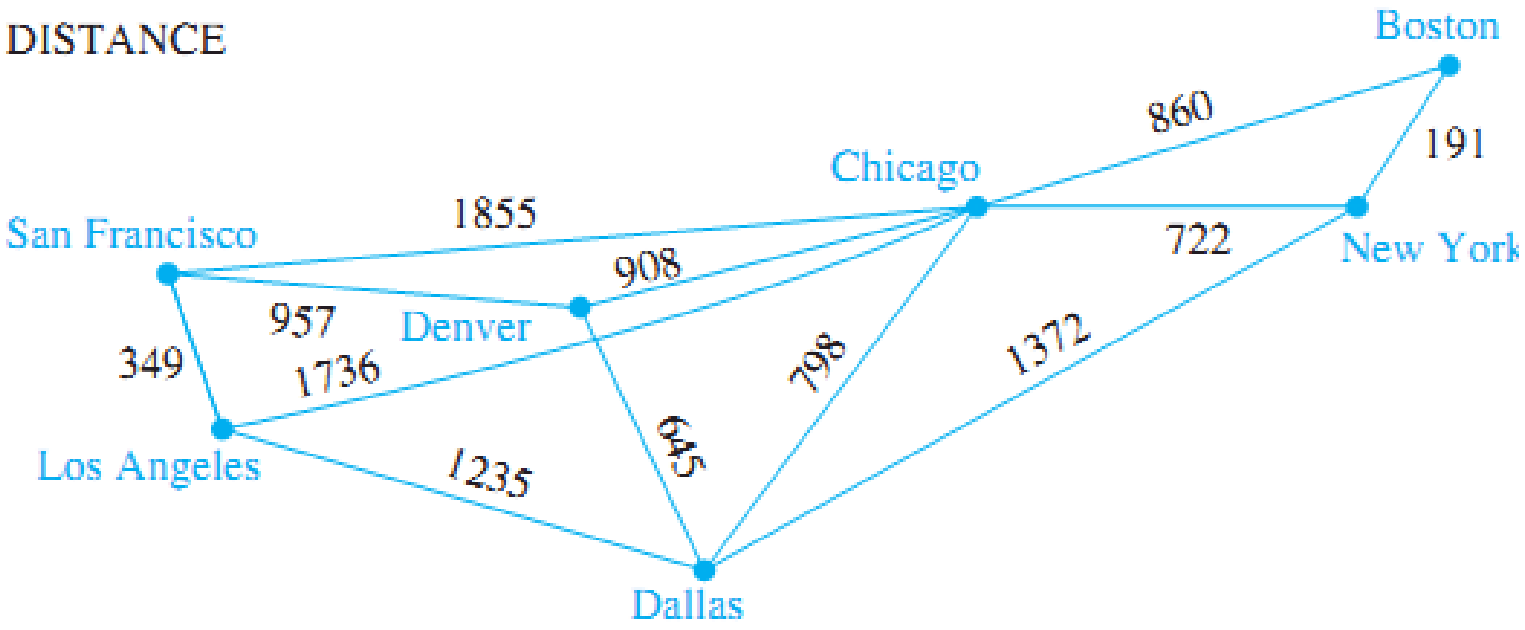
Fares


FARES
Boston
$129
$79
$39
$99
Chicago
$59
New York
$69
$89
San Francisco
Denver
$79
$99
$39
$89
$129
Atlanta
$99
$69
Los Angeles
Miami

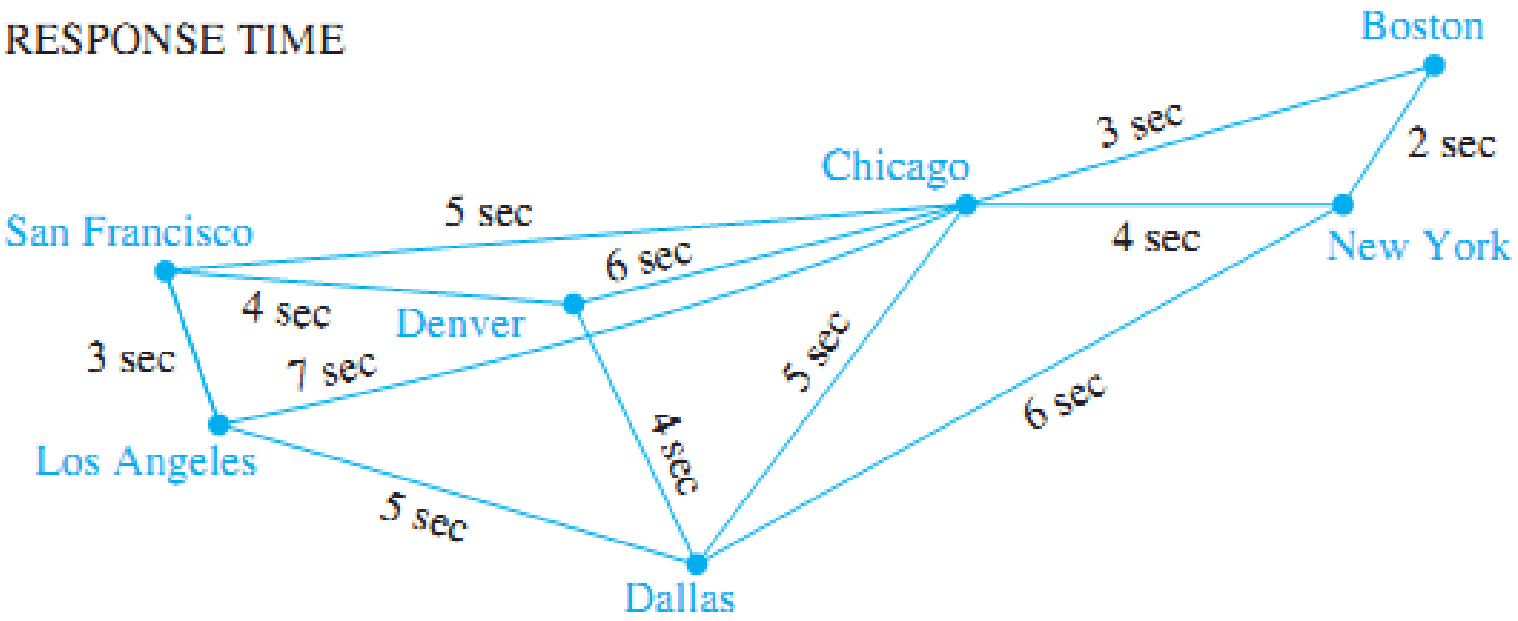
Weighted graphs can be used to model computer networks.

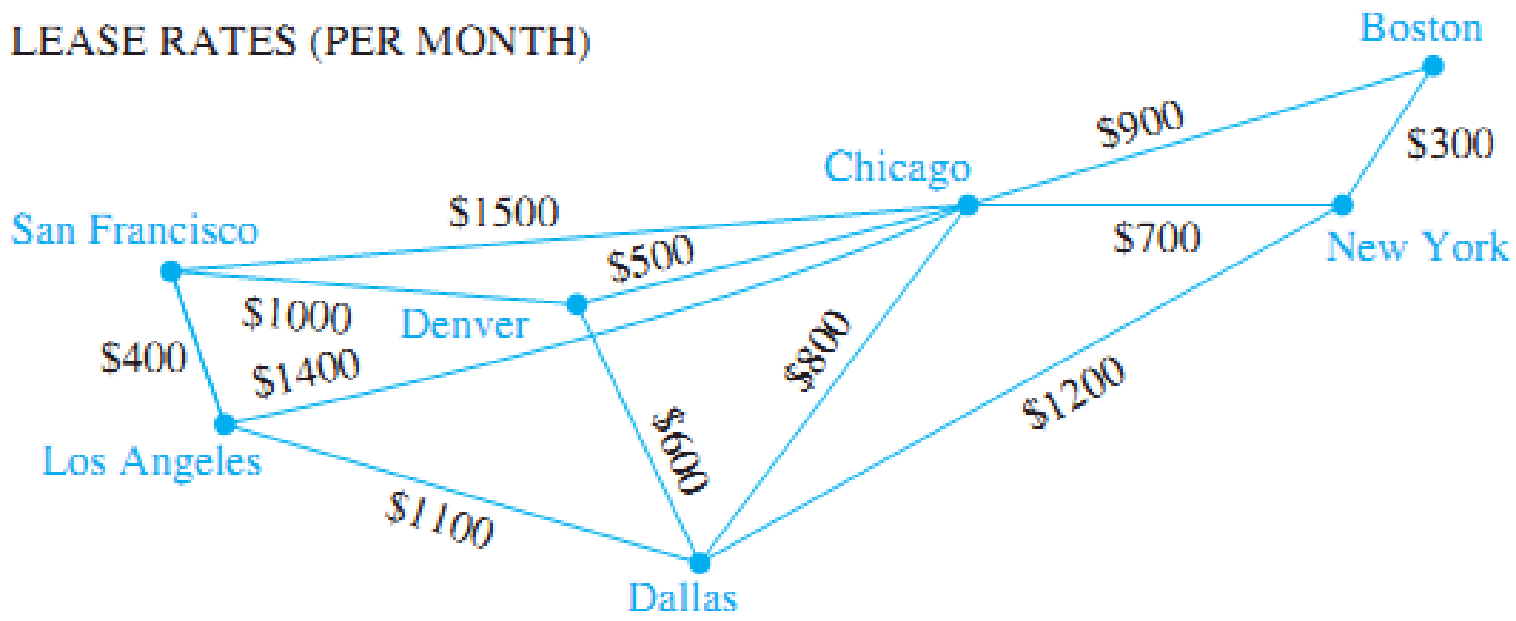
- Communications costs (such as the monthly cost of leasing a telephone line),
- the response times of the computers over these lines,
- or the distance between computers,

can all be studied using weighted graphs.


DISTANCE
Boston
860
191
Chicago
1855
San Francisco
722
New York
908
957
Denver
349
1736
798
1372
645
Los Angeles
1235
Dallas

RESPONSE TIME




LEASE RATES (PER MONTH)
Boston
$900
$300
Chicago
$1500
San Francisco
$700
New York
$500
$1000
Denver
$400
$1400
$800
$600
$1200
Los Angeles
$1100
Dallas

In a shortest-paths problem, we are given a weighted, directed graph $G = (V, E)$, with weight function $w:E \rightarrow R$ mapping edges to real-valued weights.

The weight $w(p)$ of path $p = (v_0, v_1, \ldots, v_k)$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

We define the shortest-path weight $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

A shortest path from vertex u to vertex v is then defined as any path p with weight
$w(p) = \delta(u, v)$.

Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that we would want to minimize.

Variants

We shall focus on the single-source shortest-paths problem:

given a graph $G = (V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $t \in V$.

The algorithm for the single-source problem can solve many other problems, including the following variants.

Single-destination shortest-paths problem

Find a shortest path to a given destination vertex t from each vertex v.

By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

Single-pair shortest-path problem

Find a shortest path from *u* to *v* for given vertices *u* and *v*.

If we solve the single-source problem with source vertex *u*, we solve this problem also.

Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.

All-pairs shortest-paths problem

Find a shortest path from *u* to *v* for every pair of vertices *u* and *v*.

Although we can solve this problem by running a single-source algorithm once from each vertex, we usually can solve it faster. This problem will be considered later.

Optimal substructure of a shortest path

Shortest-paths algorithms typically rely on the property that a shortest path between 2 vertices contains other shortest paths within it.

Lemma 1 (Subpaths of shortest paths are shortest paths)

Given a weighted, directed graph $G = (V, E)$ with weight function

$w : E \rightarrow R$,

let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j .

Then, p_{ij} is a shortest path from v_i to v_j .

Proof If we decompose path p into $v_0 \overset{p0i}{\sim>} v_i \overset{pij}{\sim>} v_j \overset{pjk}{\sim>} v_k$, then we have that

$w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$.

Now, assume that there is a path p_{ij}' from v_l to v_j with weight $w(p_{ij}') < w(p_{ij})$.

Then, $v_0 \overset{p0i}{\sim>} v_i \overset{p'ij}{\sim>} v_j \overset{pjk}{\sim>} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k.

Negative-weight edges

Some instances of the single-source shortest-paths problem may include edges whose weights are negative.

If the graph $G = (V, E)$ contains no negative-weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value.

If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined.

No path from s to a vertex on the cycle can be a shortest path—we can always find a path with lower weight by following the proposed "shortest" path and then traversing the negative-weight cycle.

If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

Figure 1 illustrates the effect of negative weights and negative-weight cycles on shortest-path weights.

Negative edge weights in a directed graph

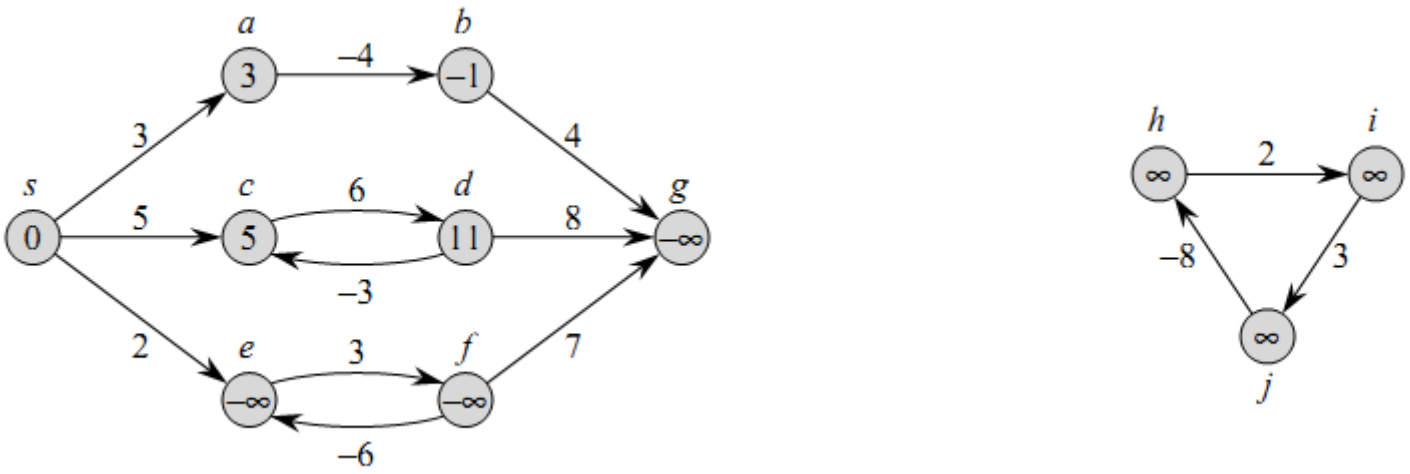


Fig. 1. The shortest-path weight from source *s* appears within each vertex.
Because vertices *e* and *f* form a negative-weight cycle reachable from *s*, they have shortest-path weights of $-\infty$.
Because vertex *g* is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$.
Vertices such as *h, i* , and *j* are not reachable from *s*, and so their shortest-path weights are ∞, even though they lie on a negative-weight cycle.

Because there is only one path from *s* to *a* (the path (*s, a*), we have $\delta(s, a) = w(s, a) = 3$. Similarly, there is only one path from *s* to *b*, and so $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$.

There are infinitely many paths from *s* to *c*: (*s, c*), (*s, c, d, c*), (*s, c, d, c, d, c*) and so on.

Because the cycle (*c, d, c*) has weight 6 + -3= 3 > 0, the shortest path from *s* to *c* is (*s, c*), with weight $\delta(s, c) = w(s, c) = 5$.

Similarly, the shortest path from *s* to *d* is (*s, c, d*),

with weight $\delta(s, d) = w(s, c) + w(c, d) = 11$.

Analogously, there are infinitely many paths from *s* to *e*: (*s, e*), (*s, e, f, e*), (*s, e, f, e, f, e*), and so on.

Because the cycle (*e, f, e*) has weight 3 +(-6)= -3 < 0, however, there is no shortest path from *s* to *e*.

By traversing the negative-weight cycle (*e, f, e*) arbitrarily many times, we can find paths from *s* to *e* with arbitrarily large negative weights, and so $\delta(s, e) = -\infty$.

Similarly, $\delta(s, f) = -\infty$.

Because *g* is reachable from *f* , we can also find paths with arbitrarily large negative weights from *s* to *g*, and so $\delta(s, g) = -\infty$.

Vertices *h, i* , and *j* also form a negative-weight cycle.

They are not reachable from s, however, and so $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative.

Others, such as the Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source.

Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

Cycles

Can a shortest path contain a cycle?

As we have just seen, it cannot contain a negative-weight cycle.

Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.

That is, if $p = (v_0, v_1, \ldots v_k)$ is a path and $c = (v_i, v_{i+1}, \ldots, v_j)$ is a positive-weight cycle on this path (so that $v_i = v_j$ and

$w(c) > 0$), then the path $p' = (v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_{ki})$ has weight

$w(p') = w(p) - w(c) < w(p)$, and so p cannot be a shortest path from v_0 to v_k.

Cycles

That leaves only 0-weight cycles.

We can remove a 0-weight cycle from any path to produce another path whose weight is the same.

Thus, if there is a shortest path from a source vertex s to a destination vertex v that contains a 0-weight cycle, then there is another shortest path from s to v without this cycle.

As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from the path until we have a shortest path that is cycle-free.

Therefore, without loss of generality we can assume that when we are finding shortest paths, they have no cycles, i.e., they are simple paths.

Since any acyclic path in a graph $G = (V, E)$ contains at most $|V|$ distinct vertices, it also contains at most $|V|-1$ edges.

Thus, we can restrict our attention to shortest paths of at most $|V|-1$ edges.

Representing shortest paths

We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well.

Given a graph $G = (V, E)$, we maintain for each vertex $v \in V$ a predecessor $v.\pi$ that is either another vertex or NIL.

The shortest-paths algorithms set the π attributes so that the chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v.

In the midst of executing a shortest-paths algorithm, however, the π values might not indicate shortest paths.

As in breadth-first search, we have considered already the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ induced by the π values.

Here again, we define the vertex set V_π to be the set of vertices of G with non-NIL predecessors, plus the source s:

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\} \ .$$

The directed edge set E_π is the set of edges induced by the π values for vertices in V_π :

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\} \ .$$

The π values produced by the single-source shortest path algorithms have the property that at termination G_π is a "shortest-paths tree"—informally, a rooted tree containing a shortest path from the source *s* to every vertex that is reachable from *s*.

A shortest-paths tree is like the breadth-first tree, but it contains shortest paths from the source defined in terms of edge weights instead of numbers of edges

A shortest-paths tree rooted at s is a directed subgraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Shortest paths are not necessarily unique, and neither are shortest-paths trees.

For example, the next slide shows a weighted, directed graph and 2 shortest-paths trees with the same root.

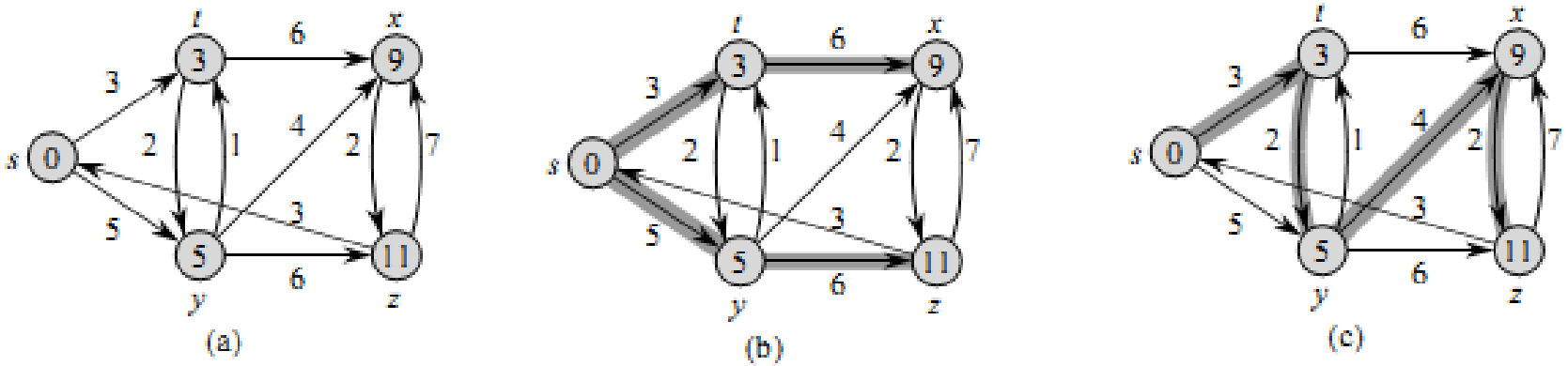


Fig. 2 a) A weighted, directed graph with shortest-path weights from source *s*.
(b) The shaded edges form a shortest-paths tree rooted at the source *s*.
(c) Another shortest-paths tree with the same root.

Initialization

For each vertex $v \in V$, we maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v.

We call $v.d$ a shortest-path estimate.

We initialize the shortest-path estimates and predecessors by the following $\Theta(V)$-time procedure:

INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** each vertex $v \in G.V$

2 $\quad v.d = \infty$

3 $\quad v.\pi$ = NIL

4 $s.d = 0$

After initialization, we have $v.\pi$ = NIL for all $v \in V$, $s.d = 0$, and $v.d = \infty$ for $v \in V - \{s\}$.

Relaxation

The process of **relaxing** an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$ and $v.\pi$.

A relaxation step may decrease the value of the shortest-path estimate $v.d$ and update v's predecessor attribute $v.\pi$.

The following code performs a relaxation step on edge (u, v) in $O(1)$ time:

```
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
```

Relaxation

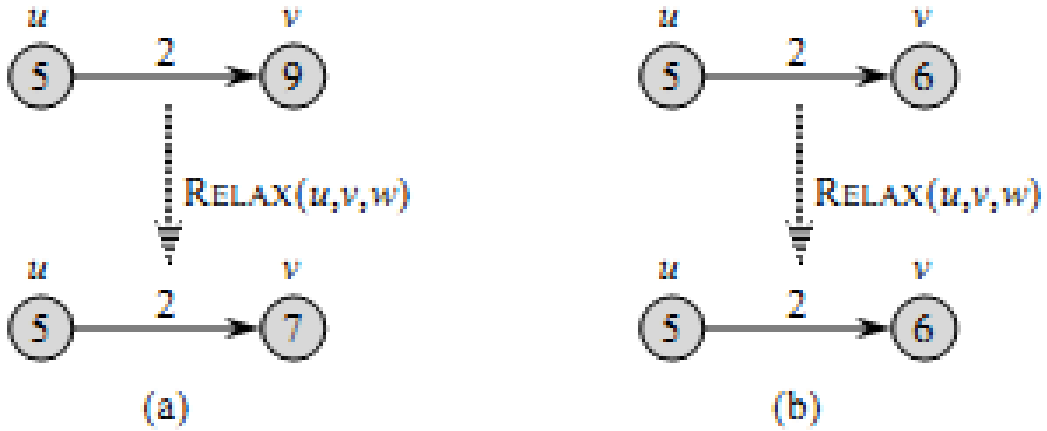


Fig. 3. Relaxing an edge (u, v) with weight $w(u, v) = 2$.
The shortest-path estimate of each vertex appears within the vertex.
(a) Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases.
(b) Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation step leaves $v.d$ unchanged.

Each considered single-source shortest path algorithm calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges.

Moreover, relaxation is the only means by which shortest-path estimates and predecessors change.

The algorithms differ in how many times they relax each edge and the order in which they relax edges.

Dijkstra's algorithm relax each edge exactly once.

The Bellman-Ford algorithm relaxes each edge $|V| - 1$ times.

Properties of shortest paths and relaxation

To prove the algorithms in this chapter correct, we shall appeal to several properties of shortest paths and relaxation.

For your reference, each property stated here includes the appropriate lemma or corollary number from Cormen (Section 24.5).

The latter 5 of these properties, which refer to shortest-path estimates or the predecessor subgraph, implicitly assume that the graph is initialized with a call to INITIALIZE-SINGLE-SOURCE(*G, s*) and that the only way that shortest-path estimates and the predecessor subgraph change are by some sequence of relaxation steps.

Triangle inequality (Lemma 24.10)

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 24.11)

We always have $v.d \geq \delta(s, u)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)

If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)

If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Predecessor-subgraph property (Lemma 24.17)

Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Arithmetic with infinities.

We shall assume that for any real number $a \neq \infty$,

$a + \infty = \infty + a = \infty$.

Also, to make our proofs hold in the presence of negative-weight cycles, we shall assume that for any real number $a \neq \infty$, we have $a + (-\infty) = (-\infty) + a = -\infty$.

All algorithms in this chapter assume that the directed graph G is stored in the adjacency-list representation.

Additionally, stored with each edge is its weight, so that as we traverse each adjacency list, we can determine the edge weights in $O(1)$ time per edge.

The Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative.

Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow R$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.

If there is such a cycle, the algorithm indicates that no solution exists.

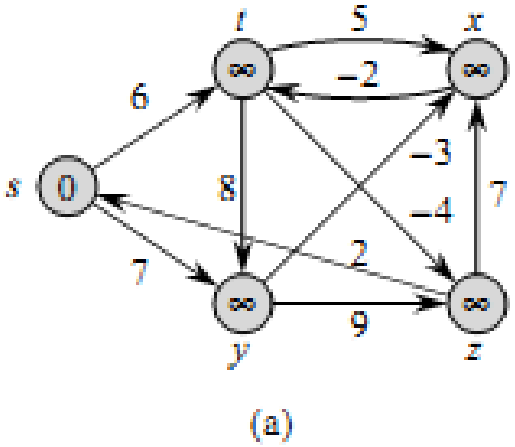
If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

The algorithm returns TRUE $\Leftrightarrow$ the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i = 1 to |G.V| - 1
3       for each edge (u, v)∈G.E
4               RELAX(u, v, w)
5 for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7               return FALSE
8 return TRUE
```

Initialization



(a)

The source is vertex *s*.

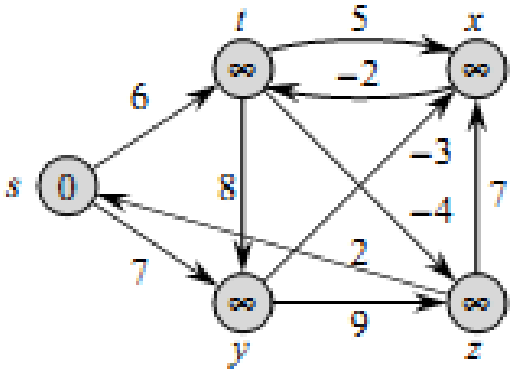
The *d* values appear within the vertices, and shaded edges indicate predecessor values: if edge (*u, v*) is shaded, then $v.\pi = u$.

In this particular example, each pass relaxes the edges in the order (*t, x*), (*t, y*), (*t, z*), (*x, t*), (y, x), (*y, z*), (*z, x*), (*z, s*), (*s, t*), (*s, y*).

(a) The situation just before the first pass over the edges.

1st iteration

before:



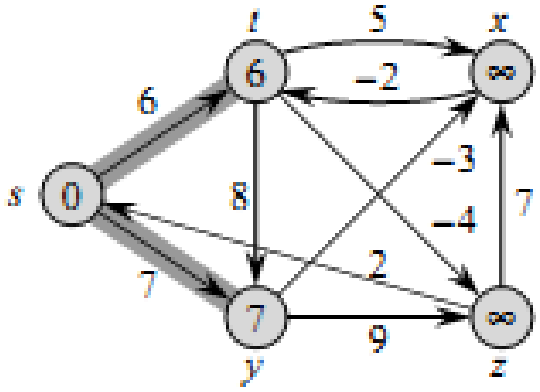
(a)

If $v.d > u.d + w(u, v)$

$v.d = u.d + w(u, v)$

$v.\pi = u$

After the 1st iteration:



(b)

(t, x): $t.d + 5 = \infty + 5 = \infty$ => $x.d = \infty$ (no change)

(t, y): $t.d + 8 = \infty + 8 = y.d = \infty$ => no change

(t, z): $t.d - 4 = \infty - 4 = z.d = \infty$ => no change

(x, t): $x.d - 2 = \infty - 2 = t.d = \infty$ => no change

(y, x): $y.d - 3 = \infty - 3 = x.d = \infty$ => no change

(y, z): $y.d + 9 = \infty + 9 = z.d = \infty$ => no change

(z, x): $z.d + 7 = \infty + 7 = x.d = \infty$ => no change

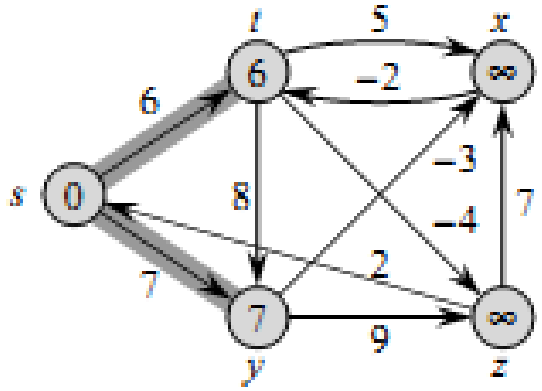
(z, s): $z.d + 2 = \infty + 2 > s.d = 0$ => no change

(s, t): $t.d = \infty$ $t.d > s.d + w(s, t) = 0 + 6$ => $t.d = 6$, $t.\pi = s$

(s, y): $y.d = \infty$ $y.d > s.d + w(s, y) = 0 + 7$ => $y.d = 7$, $y.\pi = s$

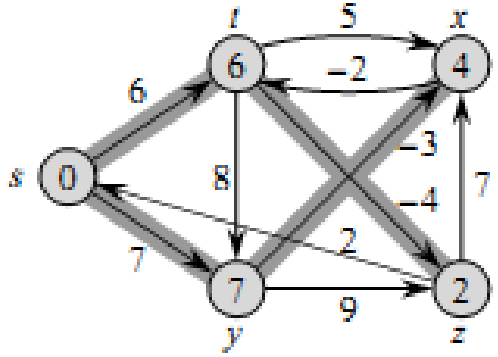
2-nd iteration:

Before:



(b)

After the 2nd iteration:

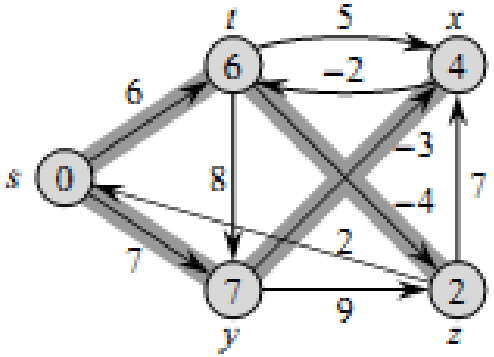


(c)

(t, x): $t.d + w(t, x) = 6 + 5 < \infty$ => $x.d = 11$, $x.\pi = t$,
(t, y): $t.d + w(t, y) = 6 + 8 > 7$ => no change
(t, z): $t.d + w(t, z) = 6 - 4 = 2 < \infty$ => $z.d = 2$, $z.\pi = t$
(x, t): $x.d + w(x, t) = 11 - 2 = 9 > 6$ => no change,
(y, x): $y.d + w(y, x) = 7 - 3 < 11$ => $x.d = 4$, $x.\pi = y$,
(y, z): $y.d + w(y, z) = 7 + 9 > 2$ => no change
(z, x): $z.d + w(z, x) = 2 + 7 > 4$ => no change
(z, s): $z.d + w(z, s) = 2 + 2 > 0$ => no change
(s, t): $s.d + w(s, t) = 0 + 6 = 6$ => no change
(s, y): $s.d + w(s, y) = 0 + 7 = 7$ => no change

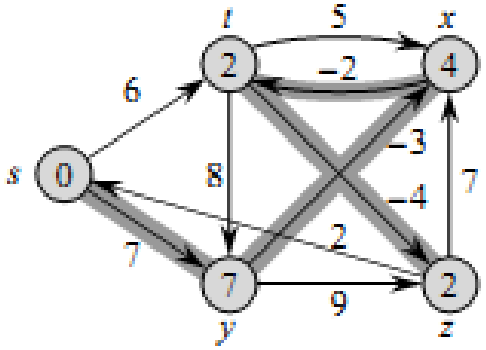
3rd iteration:

Before:



(c)

After the 3rd iteration:

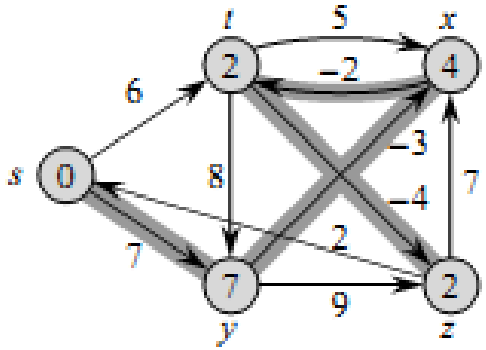


(d)

(t, x): $t.d + w(t, x) = 6 + 5 > 4$ => no change
(t, y): $t.d + w(t, y) = 6 + 8 > 7$ => no change
(t, z): $t.d + w(t, z) = 6 - 4 = 2$ => no change
(x, t): $x.d + w(x, t) = 4 - 2 < 6$ => $t.d = 2$, $t.\pi = x$
(y, x): $y.d + w(y, x) = 7 - 3 = 4$ => no change,
(y, z): $y.d + w(y, z) = 7 + 9 > 2$ => no change
(z, x): $z.d + w(z, x) = 2 + 7 > 4$ => no change
(z, s): $z.d + w(z, s) = 2 + 2 > 0$ => no change
(s, t): $s.d + w(s, t) = 0 + 6 = 6$ => no change
(s, y): $s.d + w(s, y) = 0 + 7 = 7$ => no change

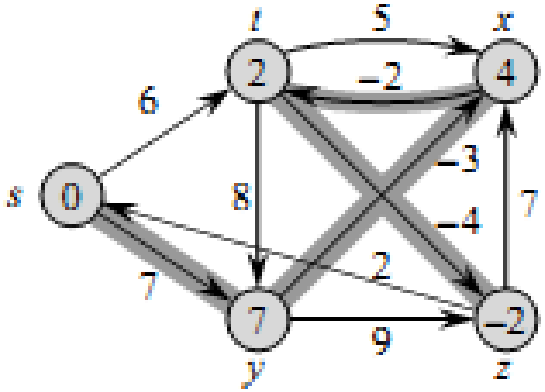
4th iteration:

Before:



(d)

After the 4th iteration:



(e)

(t, x): $t.d + w(t, x) = 2 + 5 > 4$ => no change
(t, y): $t.d + w(t, y) = 2 + 8 > 7$ => no change
(t, z): $t.d + w(t, z) = 2 - 4 < 2$ => $z.d = -2$, $z.\pi = t$
(x, t): $x.d + w(x, t) = 4 - 2 = 2$ => no change
(y, x): $y.d + w(y, x) = 7 - 3 = 4$ => no change
(y, z): y.d + w(y, z) =7 + 9 > -2 => no change
(z, x): $z.d + w(z, x) = -2 + 7 = 5 > 4$ => no change
(z, s): $z.d + w(z, s) = -2 + 2 = 0$ => no change
(s, t):$s.d + w(s, t) = 0 + 6 = 6$ => no change
(s, y):$s.d + w(s, y) = 0 + 7 = 7$ => no change

Slides 39-43 show the execution of the Bellman-Ford algorithm on a graph with 5 vertices.

After initializing the d and d π values of all vertices in line 1, the algorithm makes

$|V|$ - 1 = 4 passes over the edges of the graph.

Each pass is 1 iteration of the **for** loop of lines 2–4 and consists of relaxing each edge of the graph once.

After making $|V|$ - 1 passes, lines 5–8 check for a negative-weight cycle and return the appropriate boolean value.

The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V|$ - 1 passes over the edges in lines 2–4 takes $\Theta(E)$time, and the for loop of lines 5–7 takes $O(E)$ time.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that if there are no negative-weight cycles, the algorithm computes correct shortest-path weights for all vertices reachable from the source.

Lemma 2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight function $w: E \rightarrow R$, and assume that G contains no negative-weight cycles that are reachable from s.

Then, after the $|V| - 1$ iterations of the **for** loop of lines 2–4 of BELLMAN-FORD, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Proof We prove the lemma by appealing to the path-relaxation property. Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v.

Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \leq |V| - 1$.

Each of the $|V| - 1$ iterations of the **for** loop of lines 2–4 relaxes all $|E|$ edges.

Among the edges relaxed in the i th iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i).

By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$.

Corollary 3

Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function w: $E \rightarrow R$, and assume that G contains no negative-weight cycles that are reachable from s.

Then, for each vertex $v \in V$, there is a path from s to $v \Leftrightarrow$ BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Theorem 4 (Correctness of the Bellman-Ford algorithm)

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w: E \rightarrow R$.

If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G is a shortest-paths tree rooted at s.

If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof Suppose that graph G contains no negative-weight cycles that are reachable from the source s.

We first prove the claim that at termination, $v.d = \delta(s, v)$ for all vertices $v \in V$.

a) If vertex v is reachable from s, then Lemma 2 proves this claim.

b) If v is not reachable from s, then the claim follows from the no-path property.

Thus, the claim is proven.

c) The predecessor-subgraph property, along with the claim, implies that G_π is a shortest-paths tree.

d) Now we use the claim to show that BELLMAN-FORD returns TRUE.

At termination, we have for all edges $(u, v) \in E$,

$v.d = \delta(s, v)$

$\leq \delta(s, u) + w(u, v)$ (by the triangle inequality)

$= u.d + w(u, v)$.

and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

e) Now, suppose that graph *G* contains a negative-weight cycle that is reachable from the source *s*; let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \qquad (1)$$

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE.

Thus, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.

Summing the inequalities around cycle *c* gives us

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) . \end{aligned}$$

Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations

$$\sum_{i=1}^{k} v_i.d \quad \text{and} \quad \sum_{i=1}^{k} v_{i-1}.d,$$

And so

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d .$$

Moreover, by Corollary 3, $v_i.d$ is finite for $i = 1, 2, \ldots, k$. Thus,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) ,$$

which contradicts inequality (1).

We conclude that the Bellman-Ford algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise.

Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative.

In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.

The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

In the following implementation, we use a min-priority queue Q (очередь с приоритетами) of vertices, keyed by their d 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)
```

Dijkstra's algorithm relaxes edges as shown in Slides 54-59.

Line 1 initializes the d and π values in the usual way, and line 2 initializes the set S to the empty set.

The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4–8.

Line 3 initializes the min-priority queue Q to contain all the vertices in V ; since $S = \varnothing$; at that time, the invariant is true after line 3.

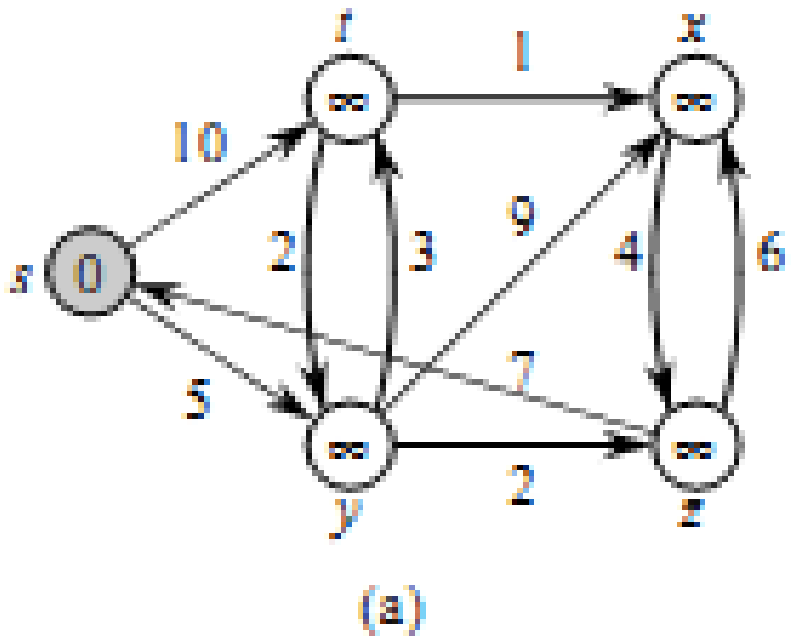
Each time through the **while** loop of lines 4–8, line 5 extracts a vertex u from $Q = V - S$ and line 6 adds it to set S, thereby maintaining the invariant.

(The first time through this loop, $u = s$.)

Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$.

Then, lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to ***v*** found so far by going through u.

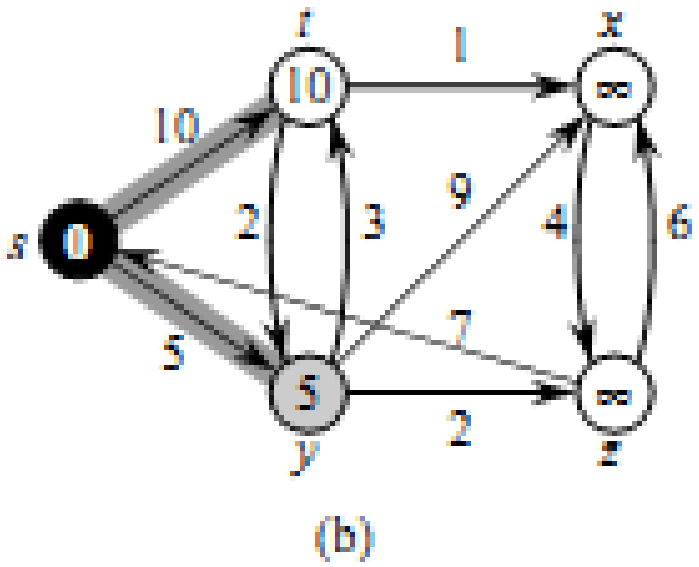
Observe that the algorithm never inserts vertices into Q after line 3 and that each vertex is extracted from Q and added to S exactly once, so that the **while** loop of lines 4–8 iterates exactly $|V|$ times.



(a)

After initialization:

$Q = \varnothing$



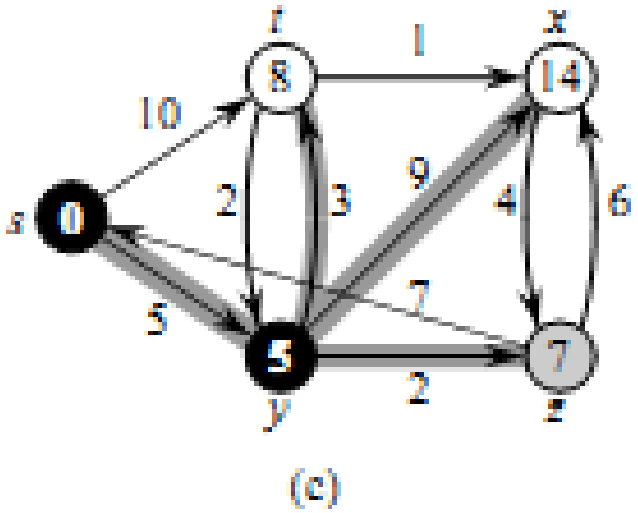
(b)

1st iteration:

$S = \{s\}$

$\{s, y\}$: $0 + 5 < \infty \Rightarrow y.d = 5, y.\pi = s$

$\{s, t\}$: $0 + 10 < \infty \Rightarrow t.d = 10, t.\pi = s$



(c)

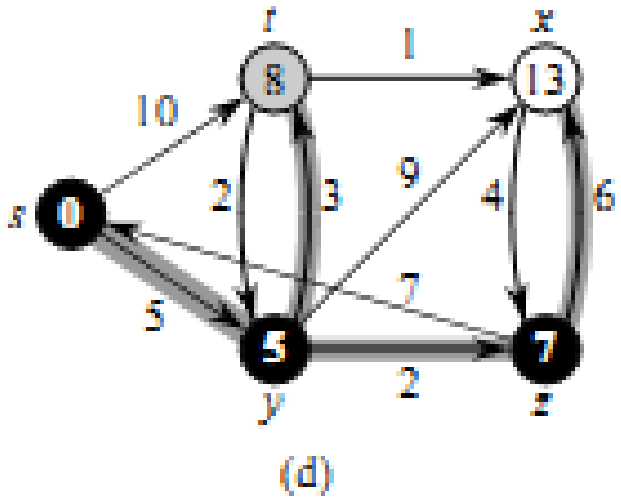
2nd iteration:

$S = \{s, y\}$

$\{y, t\}$: $5 + 3 < 10 \Rightarrow t.d = 8, t.\pi = y$

$\{y, x\}$: $5 + 9 < \infty \Rightarrow x.d = 14, x.\pi = y$

$\{y, z\}$: $5 + 2 < \infty \Rightarrow z.d = 7, z.\pi = y$

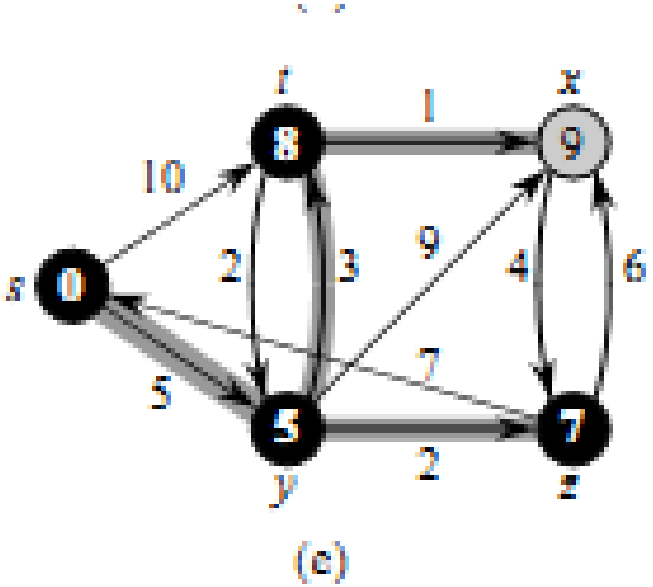


(d)

3rd iteration:

$S = \{s, y, z\}$

$\{z, x\}$: $7 + 6 < 13 \Rightarrow x.d = 13$,
$x.\pi = z$

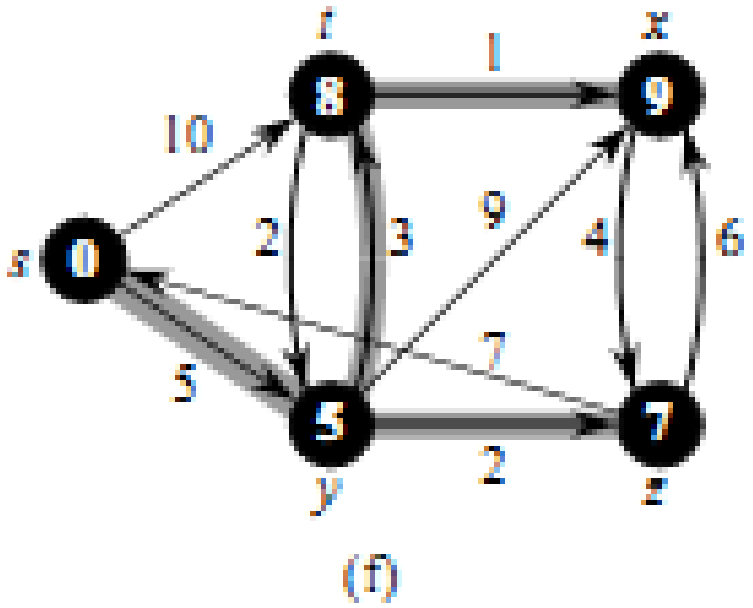


(c)

4th iteration:

$S = \{s, y, z, t\}$

$\{t, x\}$: $8 + 1 < 13 \Rightarrow x.d = 9, x.\pi = t$



(f)

5th iteration:

$S = \{s, y, z, t, x\}$

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy.

Greedy strategies do not always yield optimal results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths.

The key is to show that each time it adds a vertex u to set S, we have $u.d = \delta(s, u)$.

Theorem 5 (Correctness of Dijkstra's algorithm)

Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function w and source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof We use the following loop invariant:

At the start of each iteration of the **while** loop of lines 4–8,
$v.d = \delta(s)$ for each vertex $v \in S$.

It suffices to show for each vertex $u \in V$, we have $u.d = \delta(s, u)$ at the time when u is added to set S.

Once we show that $u.d = \delta(s, u)$, we rely on the upper-bound property to show that the equality holds at all times thereafter.

Initialization: Initially, $S = \varnothing$, and so the invariant is trivially true.

Maintenance: We wish to show that in each iteration, $u.d = \delta(s, u)$ for the vertex added to set S.

For the purpose of contradiction, let u be the first vertex for which $u.d \neq \delta(s, u)$ when it is added to set S.

We shall focus our attention on the situation at the beginning of the iteration of the **while** loop in which u is added to S and derive the contradiction that $u.d = \delta(s, u)$ at that time by examining a shortest path from s to u.

We must have $u \neq s$ because s is the first vertex added to set S and $s.d = \delta(s, s) = 0$ at that time.

Because $u \neq s$, we also have that $S \neq \varnothing$; just before u is added to S.

There must be some path from s to u, for otherwise $u.d = \delta(s, u) = \infty$ by the no-path property, which would violate our assumption that $u.d \neq \delta(s, u)$.

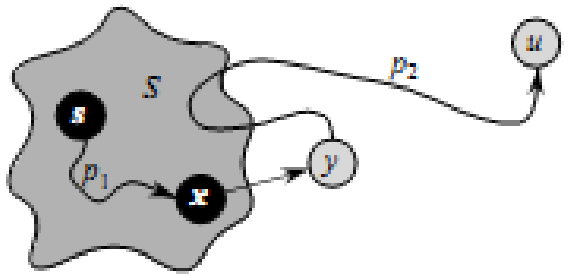
Because there is at least one path, there is a shortest path p from s to u.

Prior to adding u to S, path p connects a vertex in S, namely s, to a vertex in $V - S$, namely u.

Let us consider the first vertex y along p such that $y \in V - S$, and let $x \in S$ be y's predecessor along p.

Thus, we can decompose path p from source s to vertex u into $s \overset{p1}{\sim>} x \to y \overset{p2}{\sim>} u$.

(Either of paths $p1$ or $p2$ may have no edges.)



Set S is nonempty just before vertex u is added to it.

y is the first vertex on the path that is not in S and $x \in S$ immediately precedes y.

Vertices x and y are distinct, but we may have $s = x$ or $y = u$.

Path p_2 may or may not reenter set S.

We claim that $y.d = \delta(s, y)$ when u is added to S.

To prove this claim, observe that $x \in S$.

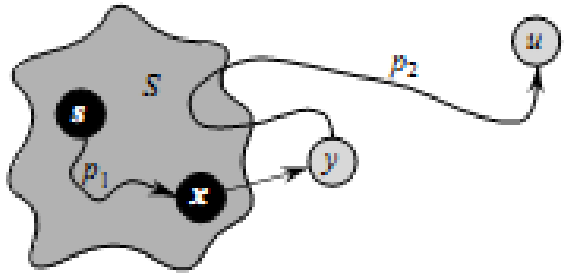
Then, because we chose u as the first vertex for which $u.d \neq \delta(s, u)$ when it is added to S, we had $x.d = \delta(s, x)$ when x was added to S.

Edge (x, y) was relaxed at that time, and the claim follows from the convergence property.

We can now obtain a contradiction to prove that $u.d = \delta(s, u)$.

Because y appears before u on a shortest path from s to u and all edge weights are nonnegative (notably those on path p_2), we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \qquad (2) \\ &\leq u.d \text{ (by the upper-bound property)} . \end{aligned}$$

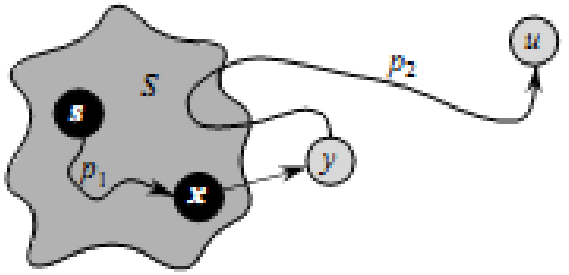


But because both vertices u and y were in V-S when u was chosen in line 5, we have $u.d \leq y.d$.

Thus, the two inequalities in (2) are in fact equalities, giving $y.d = \delta(s, y) = \delta(s, u) = u.d$.

Consequently, $u.d = \delta(s, u)$, which contradicts our choice of u.

We conclude that $u.d = \delta(s, u)$ when u is added to S, and that this equality is maintained at all times thereafter.



Termination: At termination, $Q = \varnothing$ which, along with our earlier invariant that $Q = V - S$, implies that $S = V$.

Thus, $u.d = \delta(s, u)$ for all vertices $u \in V$.

Corollary 6

If we run Dijkstra's algorithm on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source s, then at termination, the predecessor subgraph G_π is a shortest-paths tree rooted at s.

Proof Immediate from Theorem 5 and the predecessor-subgraph property.

We can now estimate the computational complexity of Dijkstra's algorithm (in terms of additions and comparisons).

The algorithm uses no more than $|V|$ iterations where $|V|$ is the number of vertices in the graph, because one vertex is added to the distinguished set at each iteration.

We are done if we can estimate the number of operations used for each iteration.

We can identify the vertex not in S with the smallest $v.d$ estimate using no more than $|V| - 1$ comparisons.

Then we use an addition and a comparison to update the label of each vertex not in S.

It follows that no more than $2(|V| - 1)$ operations are used at each iteration, because there are no more than $n - 1$ estimates to update at each iteration.

Because we use no more than $|V|$ iterations, each using no more than $2(|V| - 1)$ operations, we have Theorem 7.

THEOREM 7 Dijkstra's algorithm uses $O(|V|^2)$ operations (additions and comparisons) to solve a single source shortest paths problem in a directed weighted graph with $|V|$ vertices.