ICS 311 #18: Single Source Shortest Paths

Outline

Today's Theme: Relax!

1. Shortest Paths Problems
2. Bellman-Ford Algorithm
3. Shortest Paths in a DAG
4. Dijkstra's Algorithm

Readings and Screencasts

- Required: CLRS 3rd Ed., Sections 24.1-24.3.
- See also: Sedgewick (1984) Chapter 31 for light conceptual introduction (in Laulima), or Sedgewick & Wayne (2001) Algorithms Chapter 4 for code and application examples.
- Screencasts 18 A Intro, 18 B Bellman-Ford , 18 C Dijkstra.

Shortest Paths Problems

or how to get there from here ...

Definition

Input is a directed graph $G = (V, E)$ and a ***weight function*** $w: E \to \Re$.

Define the ***path weight w(p)*** of path $p = \langle v_0, v_1, \ldots v_k \rangle$ to be the sum of edge weights on the path:

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

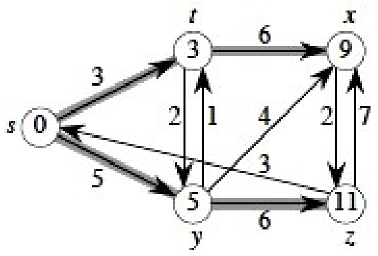
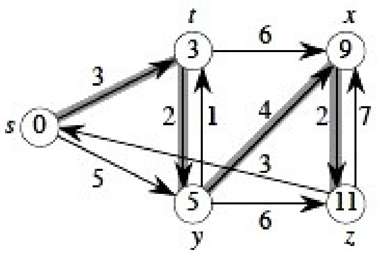
Then the ***shortest path weight*** from u to v is:

$$\delta(u, v) = \begin{cases} \min\left\{w(p) : u \overset{p}{\leadsto} v\right\} & \text{if there exists a path } u \leadsto v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

A **shortest path** from u to v is any path such that $w(p) = \delta(u, v)$.

Examples

In our examples the shortest paths will always start from s, the ***source***. The δ values will appear inside the vertices, and shaded edges show the shortest paths.

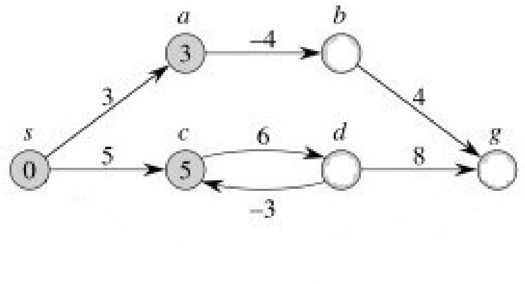
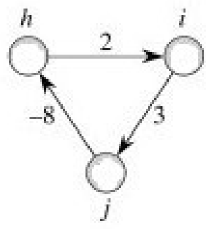
As can be seen, shortest paths are not unique.

Variations

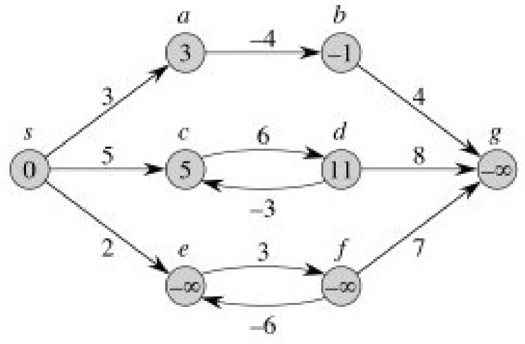
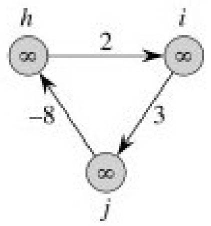
- ***Single-Source:*** from *s* to every *v* ∈ *V* (the version we consider)
- ***Single-Destination:*** from every *v* ∈ *V* to some *d*. (Solve by reversing the links and solving single source.)
- ***Single-Pair:*** from some *u* to some *v*. Every known algorithm takes just as long as solving Single-Source. (*Why might that be the case?*)
- ***All-Pairs:*** for every pair *u*, *v* ∈ *V*. Next lecture.

Negative Weight Edges

These are OK as long as no negative-weight cycles are reachable from the source *s*. Fill in the blanks:

If a negative-weight cycle is accessible, it can be iterated to make *w*(*s*, *v*) arbitarily small for all *v* on the cycle:

Some algorithms can detect negative-weight cycles and others cannot, but when they are present shortest paths are not well defined.

Cycles

Shortest paths cannot contain cycles.

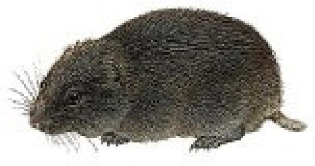
- We already ruled out negative-weight cycles.

- If there is a positive-weight cycle we can get a shorter path by omitting the cycle, so it can't be a shortest path with the cycle.
- If there is a zero-weight cycle, it does not affect the cost to omit them, so we will assume that solutions won't use them.

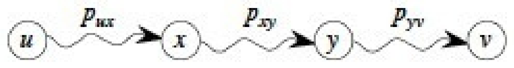
Optimal Substructure

The shortest paths problem exhibits ***optimal substructure***, suggesting that greedy algorithms and dynamic programming may apply. Turns out we will see examples of both (Dijkstra's algorithm in this chapter, and Floyd-Warshall in the next chapter, respectively).

Lemma: **Any subpath of a shortest path is a shortest path.**



Proof is by cut and paste. Let path p_{uv} be a shortest path from u to v, and that it includes subpath p_{xy} (this represents subproblems):

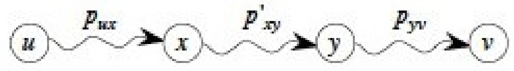


$$\begin{tikzcd} u \arrow[r, squiggly, "p_{ux}"] & x \arrow[r, squiggly, "p_{xy}"] & y \arrow[r, squiggly, "p_{yv}"] & v \end{tikzcd}$$

Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

Now, for proof by contradiction, suppose that substructure is not optimal, meaning that for some choice of these paths there exists a shorter path p'_{xy} from x to y that is shorter than p_{xy}. Then $w(p'_{xy}) < w(p_{xy})$.

From this, we can construct p':



$$\begin{tikzcd} u \arrow[r, squiggly, "p_{ux}"] & x \arrow[r, squiggly, "p'_{xy}"] & y \arrow[r, squiggly, "p_{yv}"] & v \end{tikzcd}$$

Then

$$\begin{aligned} w(p') &= w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) \\ &< w(p_{ux}) + w(p_{xy}) + w(p_{yv}) \\ &= w(p)\,. \end{aligned}$$

which contradicts the assumption that p_{uv} is a shortest path.

Algorithms

All the algorithms we consider will have the following in common.

Output

For each vertex $v \in V$, we maintain these attributes:

v.d is called the ***shortest path estimate***.

- Initially, $v.d = \infty$
- $v.d$ may be reduced as the algorithm progresses, but $v.d \geq \delta(s, v)$ is always true.
- We want to show that at the conclusion of our algorithms, $v.d = \delta(s, v)$.

v.π = the predecessor of v by which it was reached on the shortest path known so far.

- If there is no predecessor, $v.\pi$ = NIL.
- We want to show that at the conclusion of our algorithms, $v.\pi$ = the predecessor of v on the shortest path from s.
- If that is true, π induces a ***shortest path tree*** on G. (See text for proofs of properties of π.)

Initialization

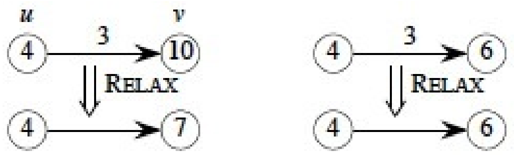
All the shortest-paths algorithms start with this:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex $v \in G.V$
2      $v.d = \infty$
3      $v.\pi = \text{NIL}$
4  $s.d = 0$
```

Relaxation

They all apply the relaxation procedure, which essentially asks: can we improve the current shortest-path estimate for v by going through u and taking (u, v)?

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



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

Shortest Paths Properties

All but the first of these properties assume that `INIT-SINGLE-SOURCE` has been called once, and then `RELAX` is called zero or more times.

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 $v.d \geq \delta(s, v)$ 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 $s \leadsto u \to v$ 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.

Proofs are available in the text. Try to explain informally why these are correct.

Bellman-Ford Algorithm

Essentially a **brute force strategy**: relax systematically enough times that you can be sure you are done.

- Allows negative-weight edges
- Computes $v.d$ and $v.\pi$ for all $v \in V$.
- Returns True (and a solution embedded in the graph) if no negative-weight cycles are reachable from s, and False otherwise.

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

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

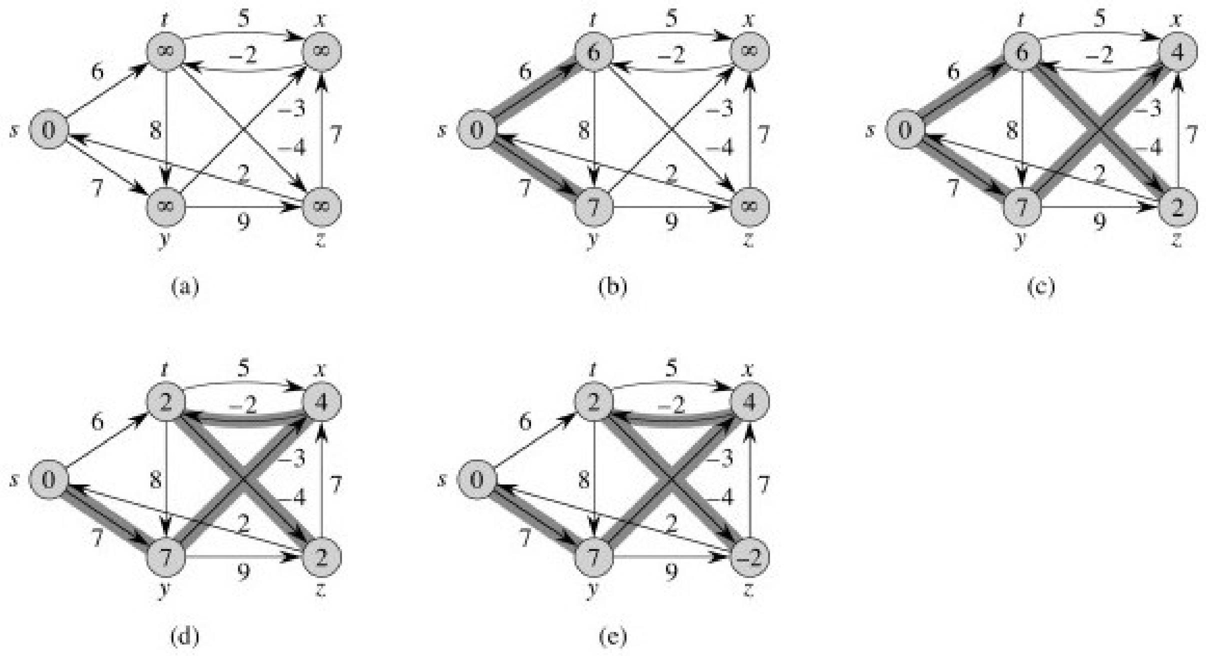
The first `for` loops do the work of relaxation. *How does the last `for` loop help -- how does it work?*

Analysis:

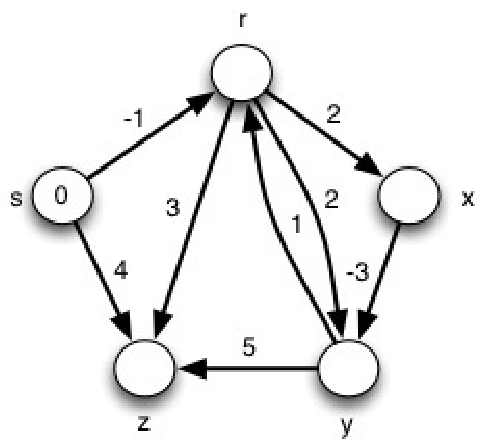
`RELAX` is O(1), and the nested `for` loops relax all edges $|V|$ - 1 times, so `BELLMAN-FORD` is $\Theta(V E)$.

Examples:

Example from the text, relaxed in order (t,x), (t,y), (t,z), (x,t), (y,x) (y,z), (z,x), (z,s), (s,t), (s,y):



Try this other example (click for answer):



```
BELLMAN-FORD($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX($u, v, w$)
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

Correctness

The values for $v.d$ and $v.\pi$ are guaranteed to converge on shortest paths after $|V| - 1$ passes, assuming no negative-weight cycles.

This can be proven with the path-relaxation property, which states that if we relax the edges of a shortest path $\langle v_0, v_1, \ldots v_k \rangle$ in order, even if interleaved with other edges, then $v_k.d = \delta(s,v_k)$ after v_k is relaxed.

Since the list of edges is relaxed as many times as the longest possible shortest path ($|V|$- 1), it must converge by this property.

- First iteration relaxes (v_0, v_1)
- Second iteration relaxes (v_1, v_2)
- ...
- kth iteration relaxes (v_{k-1}, v_k)

We also must show that the True/False values are correct.

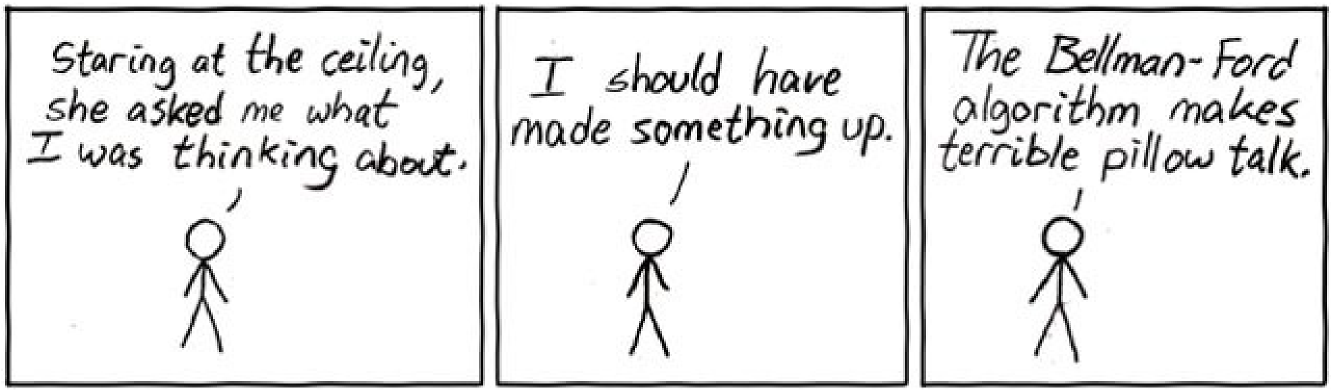
```
BELLMAN-FORD($G, w, s$)
INITIALIZE-SINGLE-SOURCE($G, s$)
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX($u, v, w$)
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE
```

Informally, we can see that if *v.d* is still getting smaller after it should have converged (see above), then there must be a negative weight cycle that continues to decrement the path.

The full proof of correctness may be found in the text.

The values computed on each pass and how quickly it converges depends on order of relaxation: it may converge earlier.

How can we use this fact to speed the algorithm up a bit?


Staring at the ceiling, she asked me what I was thinking about.
I should have made something up.
The Bellman-Ford algorithm makes terrible pillow talk.

Shortest Paths in a DAG

Life is easy when you are a DAG ...

There are no cycles in a Directed Acyclic Graph. Thus, negative weights are not a problem. Also, vertices must occur on shortest paths in an order consistent with a topological sort.

We can do something like Bellman-Ford, but don't need to do it as many times, and don't need to check for negative weight cycles:

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

Analysis:

Given that topological sort is $\Theta(V + E)$, what's the complexity of `DAG-SHORTEST-PATHS`? *This one's on you: what's the run-time complexity?* Use aggregate analysis ...

Correctness:

Because we process vertices in topologically sorted order, edges of *any* path must be relaxed in order of appearance in the path.

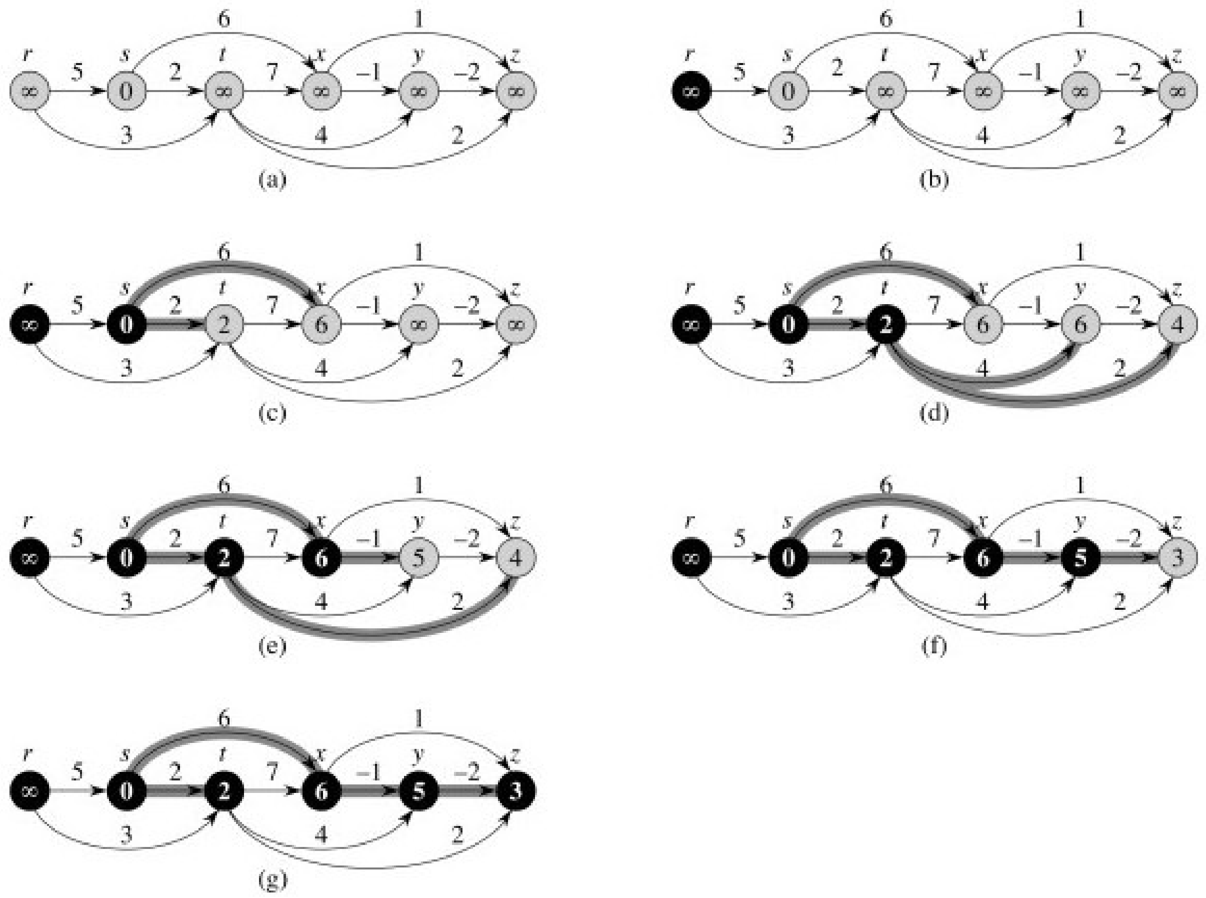
Therefore edges on any shortest path are relaxed in order.

Therefore, by the path-relaxation property, the algorithm terminates with correct values.

Examples

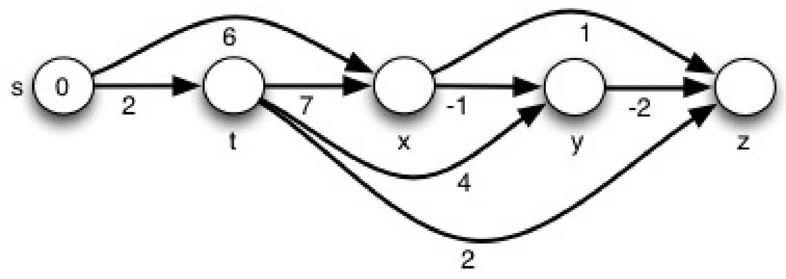
From the text:

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```



Notice we could not reach *r*!

Let's try another example (click for answer):



Dijkstra's Algorithm

The algorithm is essentially a weighted version of breadth-first search: BFS uses a FIFO queue; while this version of Dijkstra's algorithm uses a priority queue.

It also has similarities to Prim's algorithm, being greedy, and with similar iteration.

Assumes there are no negative-weight edges.

Algorithm

- S = set of vertices whose final shortest-path weights are determined.
- $Q = V - S$ is the priority queue.
- Priority queue keys are shortest path estimates $v.d$.

Here is the algorithm as given by CLRS, with Prim on the right for comparison:

```
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)
```

```
MST-PRIM(G, w, r)
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)
```

Dijkstra's algorithm is greedy in choosing the closest vertex in $V - S$ to add to S each iteration. The difference is that

- For Prim "close" means the cost to take one step to include the next cheapest vertex:
  `if w(u,v) < v.key`
- for Dijkstra "close" means the cost from the source vertex s to v: this is in the RELAX code
  `if v.d > u.d + w(u,v).`

The above specification of the algorithm can be improved. Relax(u,v,w) updates the shortest path estimates $v.d$ of the vertices that are in the priority queue. To make sure the keys of the priority queue are updated properly, we must call `DecreaseKey` on the vertex being updated. We can only know whether to do this if Relax tells us whether there was a change. Therefore the following modifications are needed:

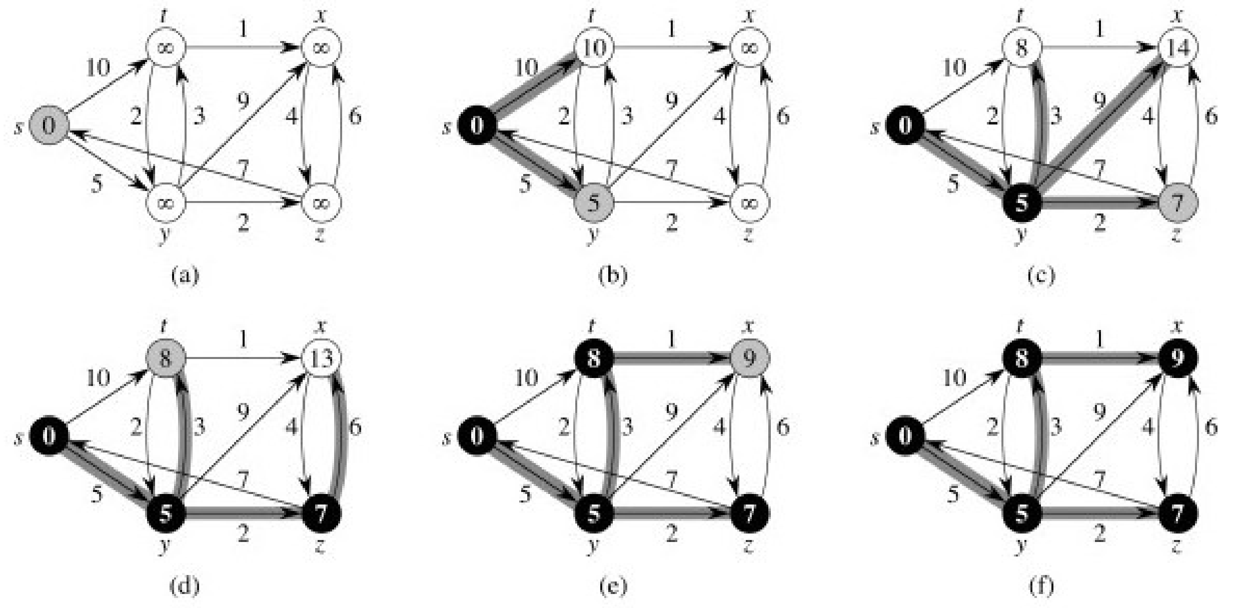
```
Relax(u,v,w)
1 if v.d > u.d + w(u,v)
2     v.d = u.d + w(u,v)
3     v.π = u
4     return TRUE
5 else
6     return FALSE
```

Then change Dijkstra as follows:

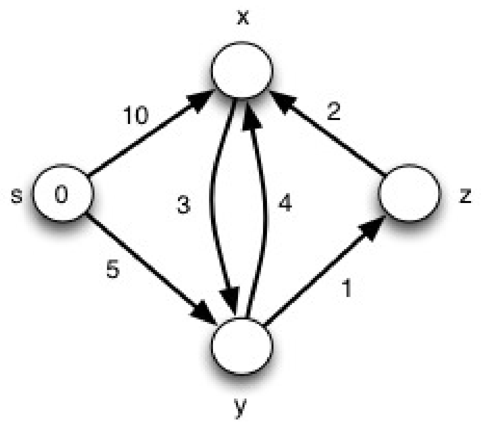
```
8         if Relax(u, v, w)
9             DecreaseKey(Q, v, v.d)
```

Examples

From the text (black vertices are set S; white vertices are on Q; shaded vertex is the min valued one chosen next iteration):

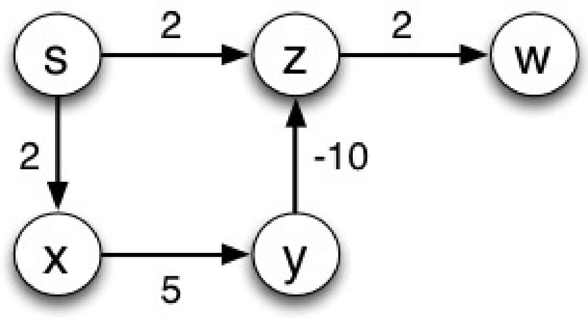


Let's try another example (click for answer):



```
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)
```

Here's a graph with a negative weight: try it from *s* and see what happens:



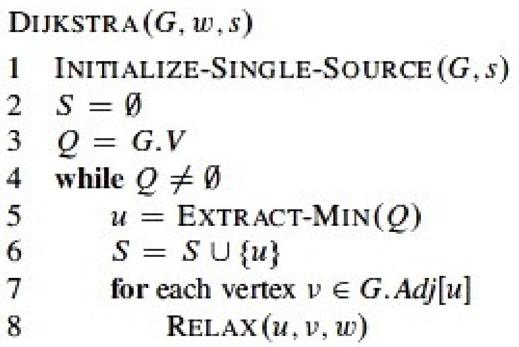
Correctness

The proof is based on the following loop invariant at the start of the `while` loop:

$v.d = \delta(s, v)$ for all $v \in S$.

Initialization: Initially $S = \emptyset$, so trivially true.

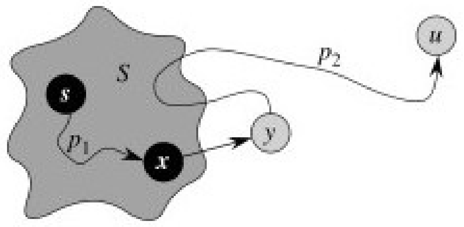
Maintenance: We just sketch this part (see text). Need to show that $u.d = \delta(s, u)$ when u is added to S in each iteration. The upper bound property says it will stay the same thereafter.



```
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)
```

Suppose (for proof by contradiction) that $\exists\, u$ such that $u.d \neq \delta(s, u)$ when added to S. Without loss of generality, let u be the first such vertex added to S.

- $u \neq s$, since $s.d = \delta(s, s) = 0$. Therefore $s \in S \neq \varnothing$.
- So there is a path from s to u. This means there must be a shortest path p from s to u.
- The proof decomposes p into a path s to x, (x, y), and a path from y to u. (Some but not all of these can be null.)
- $y.d = \delta(s, y)$ when u added to S. (By hypothesis, $x.d = \delta(s, x)$ when x was added. Relaxation of (x, y) extends this to y by the convergence property.)
- Since y appears before u on a shortest path with non-negative weights, $\delta(s,y) \leq \delta(s,u)$, and we can show that $y.d \leq u.d$ by the triangle inequality and upper-bound properties.
- But u being chosen first from Q means $u.d \leq y.d$; so must be that $u.d = y.d$.
- Therefore $y.d = \delta(s, y) = \delta(s, u) = u.d$.
- This contradicts the assumption that $u.d \neq \delta(s, u)$

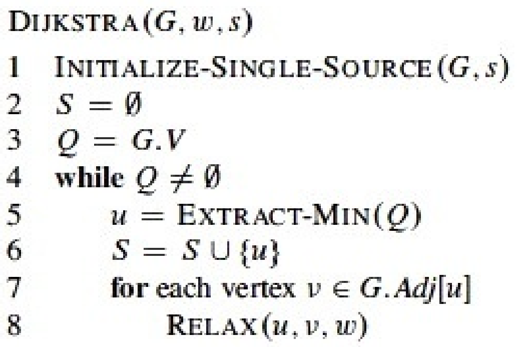


Termination: At the end, Q is empty, so $S = V$, so $v.d = \delta(s, v)$ for all $v \in V$

Analysis

The run time depends on the implementation of the priority queue.

If ***binary min-heaps*** are used:



```
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)
```

- The `EXTRACT-MIN` in line 5 and the implicit `DECREASE-KEY` operation that results from relaxation in line 8 are each O(lg V).
- The `while` loop over $|V|$ elements of Q invokes $|V|$ O(log V) `EXTRACT-MIN` operations.
- Switching to aggregate analysis for the `for` loop in lines 7-8, there is a call to `RELAX` for each of O(E) edges, and each call may result in an O(log V) `DECREASE-KEY`.
- The total is **O((V + *E*) lg V)**.
- If the graph is connected, there are at least as many edges as vertices, and this can be simplified to **O(*E* lg V)**, which is faster than `BELLMAN-FORD`'s O(E V).

With ***Fibonacci heaps*** (which were developed specifically to speed up this algorithm), O(V lg V + E) is possible. *(Do not use this result unless you are specifically using Fibonacci heaps!)*

Dan Suthers (with edits suggested by Nodari Sitchinava)
Last modified: Sun Apr 5 04:37:54 HST 2015
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition.