Lecture Notes for Chapter 21: Minimum Spanning Trees

Chapter 21 overview

Problem

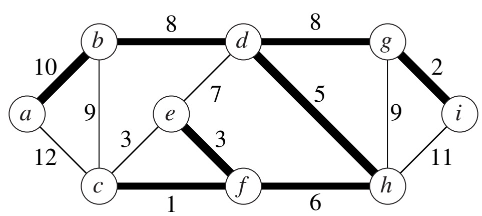
- A town has a set of houses and a set of roads.
- A road connects 2 and only 2 houses.
- A road connecting houses u and v has a repair cost $w(u, v)$.
- ***Goal:*** Repair enough (and no more) roads such that
  1. everyone stays connected: can reach every house from all other houses, and
  2. total repair cost is minimum.

Model as a graph:

- Undirected graph $G = (V, E)$.
- ***Weight*** $w(u, v)$ on each edge $(u, v) \in E$.
- Find $T \subseteq E$ such that
  1. T connects all vertices (T is a ***spanning tree***), and
  2. $w(T) = \sum_{(u,v) \in T} w(u, v)$ is minimized.

A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***.

Example of such a graph *[Differs from Figure 21.1 in the textbook. Edges in the MST are drawn with heavy lines.]* :



In this example, there is more than one MST. Replace edge (e, f) in the MST by (c, e). Get a different spanning tree with the same weight.

Growing a minimum spanning tree

Some properties of an MST:

- It has $|V| - 1$ edges.
- It has no cycles.
- It might not be unique.

Building up the solution

- Build a set A of edges.
- Initially, A has no edges.
- As edges are added to A, maintain a loop invariant:

  Loop invariant: A is a subset of some MST.

- Add only edges that maintain the invariant. If A is a subset of some MST, an edge (u, v) is ***safe*** for A if and only if $A \cup \{(u, v)\}$ is also a subset of some MST. So add only safe edges.

Generic MST algorithm

```
GENERIC-MST(G, w)
  A = ∅
  while A does not form a spanning tree
      find an edge (u, v) that is safe for A
      A = A ∪ {(u, v)}
  return A
```

Use the loop invariant to show that this generic algorithm works.

Initialization: The empty set trivially satisfies the loop invariant.

Maintenance: Since only safe edges are added, A remains a subset of some MST.

Termination: The loop must terminate by the time it considers all edges. All edges added to A are in an MST, so upon termination. A is a spanning tree that is also an MST.

Finding a safe edge

How to find safe edges?

Let's look at the example. Edge (c, f) has the lowest weight of any edge in the graph. Is it safe for $A = \emptyset$?

Intuitively: Let $S \subset V$ be any proper subset of vertices that includes c but not f (so that f is in $V - S$). In any MST, there has to be one edge (at least) that connects S with $V - S$. Why not choose the edge with minimum weight? (Which would be (c, f) in this case.)

Some definitions: Let $S \subset V$ and $A \subseteq E$.

- A ***cut*** $(S, V - S)$ is a partition of vertices into disjoint sets V and $S - V$.
- Edge $(u, v) \in E$ ***crosses*** cut $(S, V - S)$ if one endpoint is in S and the other is in $V - S$.
- A cut ***respects*** A if and only if no edge in A crosses the cut.
- An edge is a ***light edge*** crossing a cut if and only if its weight is minimum over all edges crossing the cut. For a given cut, there can be > 1 light edge crossing it.

Theorem

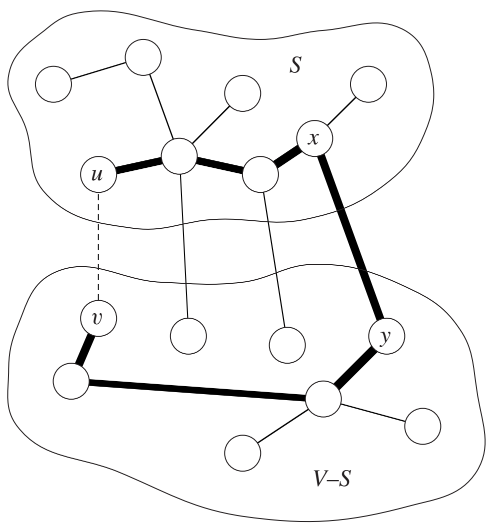
Let A be a subset of some MST, $(S, V - S)$ be a cut that respects A, and (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Proof Let T be an MST that includes A.

If T contains (u, v), done.

So now assume that T does not contain (u, v). Construct a different MST T' that includes $A \cup \{(u, v)\}$.

Recall: a tree has unique path between each pair of vertices. Since T is an MST, it contains a unique path p between u and v. Path p must cross the cut$(S, V - S)$ at least once. Let (x, y) be an edge of p that crosses the cut. From how we chose (u, v), must have $w(u, v) \le w(x, y)$.



[Except for the dashed edge (u, v), all edges shown are in T. A is some subset of the edges of T, but A cannot contain any edges that cross the cut $(S, V - S)$, since this cut respects A. Edges with heavy lines are the path p.]

Since the cut respects A, edge (x, y) is not in A.

To form T' from T:

- Remove (x, y). Breaks T into two components.
- Add (u, v). Reconnects.

So $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

T' is a spanning tree.

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\le w(T) , \end{aligned}$$

since $w(u, v) \le w(x, y)$. Since T' is a spanning tree, $w(T') \le w(T)$, and T is an MST, then T' must be an MST.

Need to show that (u, v) is safe for A:

- $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$.
- $A \cup \{(u, v)\} \subseteq T'$.
- Since T' is an MST, (u, v) is safe for A. ■ (theorem)

So, in GENERIC-MST:

- A is a forest containing connected components. Initially, each component is a single vertex.
- Any safe edge merges two of these components into one. Each component is a tree.
- Since an MST has exactly $|V| - 1$ edges, the **for** loop iterates $|V| - 1$ times. Equivalently, after adding $|V|-1$ safe edges, we're down to just one component.

Corollary
If $C = (V_C, E_C)$ is a connected component in the forest $G_A = (V, A)$ and (u, v) is a light edge connecting C to some other component in G_A (i.e., (u, v) is a light edge crossing the cut $(V_C, V - V_C)$), then (u, v) is safe for A.

Proof Set $S = V_C$ in the theorem. ■ (corollary)

This idea naturally leads to the algorithm known as Kruskal's algorithm to solve the minimum-spanning-tree problem.

Kruskal's algorithm

$G = (V, E)$ is a connected, undirected, weighted graph. $w : E \to \mathbb{R}$.

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that connects them (i.e., the light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

```
MST-KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  create a single list of the edges in G.E
  sort the list of edges into nondecreasing order by weight w
  for each edge (u, v) taken from the sorted list in order
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

Run through the above example to see how Kruskal's algorithm works on it:

(c, f) : safe
(g, i) : safe
(e, f) : safe
(c, e) : reject
(d, h) : safe
(f, h) : safe
(e, d) : reject
(b, d) : safe
(d, g) : safe
(b, c) : reject
(g, h) : reject
(a, b) : safe

At this point, there is only one component, so that all other edges will be rejected. *[Could add a test to the main loop of* KRUSKAL *to stop once* $|V| - 1$ *edges have been added to* A*.]*

Get the heavy edges shown in the figure.

Suppose (c, e) had been examined *before* (e, f). Then would have found (c, e) safe and would have rejected (e, f).

Analysis

Initialize A:	$O(1)$
First **for** loop:	$\|V\|$ MAKE-SETs
Sort E:	$O(E \lg E)$
Second **for** loop:	$O(E)$ FIND-SETs and UNIONs

- Assuming the implementation of disjoint-set data structure, already seen in Chapter 19, that uses union by rank and path compression:

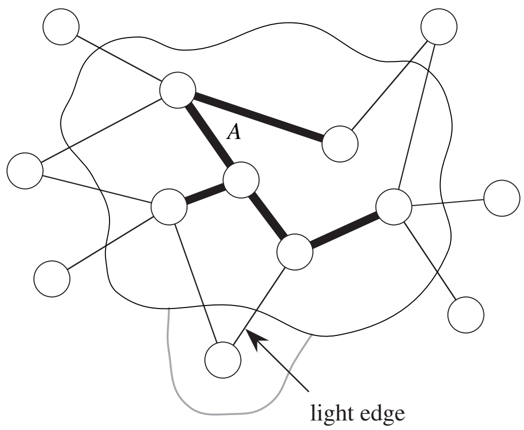
  $$O((V + E)\,\alpha(V)) + O(E \lg E)\ .$$

- Since G is connected, $|E| \geq |V| - 1 \Rightarrow O(E\,\alpha(V)) + O(E \lg E)$.
- $\alpha(|V|) = O(\lg V) = O(\lg E)$.
- Therefore, total time is $O(E \lg E)$.

- $|E| \le |V|^2 \Rightarrow \lg |E| = O(2 \lg V) = O(\lg V)$.
- Therefore, $O(E \lg V)$ time. (If edges are already sorted, $O(E\ \alpha(V))$, which is almost linear.)

Prim's algorithm

- Builds one tree, so A is always a tree.
- Starts from an arbitrary "root" r.
- At each step, find a light edge connecting A to an isolated vertex. Such an edge must be safe for A. Add this edge to A.



[Edges of A are drawn with heavy lines.]

How to find the light edge quickly?

Use a priority queue Q:

- Each object is a vertex *not* in A.
- $v.key$ is the minimum weight of any edge connecting v to a vertex in A. $v.key = \infty$ if no such edge.
- $v.\pi$ is v's parent in A.
- Maintain A implicitly as $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
- At completion, Q is empty and the minimum spanning tree is $A = \{(v, v.\pi) : v \in V - \{r\}\}$.

```
MST-PRIM(G, w, r)
 for each vertex u ∈ G.V
     u.key = ∞
     u.π = NIL
 r.key = 0
 Q = ∅
 for each vertex u ∈ G.V
     INSERT(Q, u)
 while Q ≠ ∅
     u = EXTRACT-MIN(Q)          // add u to the tree
     for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
         if v ∈ Q and w(u, v) < v.key
             v.π = u
             v.key = w(u, v)
             DECREASE-KEY(Q, v, w(u, v))
```

Loop invariant: Prior to each iteration of the **while** loop,

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $v.\pi \neq$ NIL, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

Do example from the graph on page 21-1. *[Let a student pick the root.]*

Analysis

Depends on how the priority queue is implemented:

- Suppose Q is a binary heap.

Initialize Q and first **for** loop:	$O(V \lg V)$
Decrease key of r:	$O(\lg V)$
while loop:	$\lvert V\rvert$ EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
	$\leq \lvert E\rvert$ DECREASE-KEY calls $\Rightarrow O(E \lg V)$
Total:	$O(E \lg V)$

- Suppose DECREASE-KEY could take $O(1)$ *amortized* time.

  Then $\leq |E|$ DECREASE-KEY calls take $O(E)$ time altogether $\Rightarrow$ total time becomes $O(V \lg V + E)$.

  In fact, there is a way to perform DECREASE-KEY in $O(1)$ amortized time: Fibonacci heaps, mentioned in the introduction to Part V.

Chapter 22: Single-Source Shortest Paths

Reading: Chapter 22 intro, 22.1, 22.3, 22.5 (skip DAGs and difference constraints)

Lecture Notes for Chapter 22: Single-Source Shortest Paths

Shortest paths

How to find the shortest route between two points on a map.

Input:

- Directed graph $G = (V, E)$
- Weight function $w : E \to \mathbb{R}$

Weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$

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

$=$ sum of edge weights on path p .

Shortest-path weight u to v:

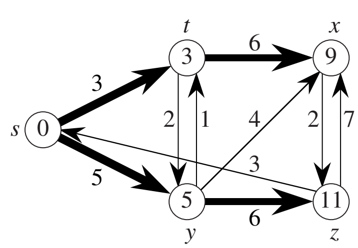
$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there exists a path } u \leadsto v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

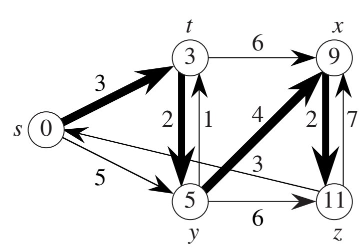
Shortest path u to v is any path p such that $w(p) = \delta(u, v)$.

Example

shortest paths from s

[δ values appear inside vertices. Heavy edges show shortest paths.]





This example shows that a shortest path might not be unique.

It also shows that when we look at shortest paths from one vertex to all other vertices, the shortest paths are organized as a tree.

Can think of weights as representing any measure that

- accumulates linearly along a path, and
- we want to minimize.

Examples: time, cost, penalties, loss.

Generalization of breadth-first search to weighted graphs.

Variants

- ***Single-source:*** Find shortest paths from a given ***source*** vertex $s \in V$ to every vertex $v \in V$.
- ***Single-destination:*** Find shortest paths to a given destination vertex.
- ***Single-pair:*** Find shortest path from u to v. No way known that's better in worst case than solving single-source.
- ***All-pairs:*** Find shortest path from u to v for all $u, v \in V$. We'll see algorithms for all-pairs in the next chapter.

Negative-weight edges

OK, as long as no negative-weight cycles are reachable from the source.

- If we have a negative-weight cycle, we can just keep going around it, and get $w(s, v) = -\infty$ for all v on the cycle.
- But OK if the negative-weight cycle is not reachable from the source.
- Some algorithms work only if there are no negative-weight edges in the graph. We'll be clear when they're allowed and not allowed.

Optimal substructure

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

Proof Cut-and-paste.

$$u \overset{p_{ux}}{\leadsto} x \overset{p_{xy}}{\leadsto} y \overset{p_{yv}}{\leadsto} v$$

Suppose this path p is a shortest path from u to v.

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

Now suppose there exists a shorter path $x \overset{p'_{xy}}{\leadsto} y$.

Then $w(p'_{xy}) < w(p_{xy})$.

Construct p':

$$u \overset{p_{ux}}{\leadsto} x \overset{p'_{xy}}{\leadsto} y \overset{p_{yv}}{\leadsto} v$$

Then

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

Contradicts the assumption that p is a shortest path. ■ (lemma)

Cycles

Shortest paths can't contain cycles:

- Already ruled out negative-weight cycles.
- Positive-weight $\Rightarrow$ we can get a shorter path by omitting the cycle.
- 0-weight: no reason to use them $\Rightarrow$ assume that our solutions won't use them.

Output of single-source shortest-path algorithm

For each vertex $v \in V$:

- $v.d = \delta(s, v)$.
  - Initially, $v.d = \infty$.
  - Reduces as algorithms progress. But always maintain $v.d \geq \delta(s, v)$.
  - Call $v.d$ a ***shortest-path estimate***.
- $v.\pi$ = predecessor of v on a shortest path from s.
  - If no predecessor, $v.\pi$ = NIL.
  - π induces a tree—***shortest-path tree***.
  - We won't prove properties of π in lecture—see text.

Initialization

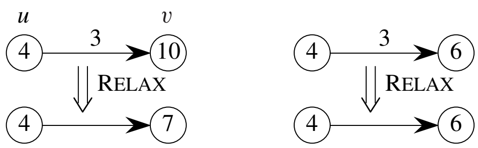
All the shortest-paths algorithms start with INITIALIZE-SINGLE-SOURCE.

```
INITIALIZE-SINGLE-SOURCE(G, s)
  for each vertex v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```

Relaxing an edge (u, v)

Can the shortest-path estimate for v be improved by going through u and taking (u, v)?

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



For all the single-source shortest-paths algorithms we'll look at,

- start by calling INITIALIZE-SINGLE-SOURCE,
- then relax edges.

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

Shortest-paths properties

[The textbook states these properties in the chapter introduction and proves them in a later section. You might elect to just state these properties at first and prove them later.]

Based on calling INITIALIZE-SINGLE-SOURCE once and then calling RELAX zero or more times.

Triangle inequality: For all $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Proof Weight of shortest path $s \leadsto v$ is $\le$ weight of any path $s \leadsto v$. Path $s \leadsto u \to v$ is a path $s \leadsto v$, and if we use a shortest path $s \leadsto u$, its weight is $\delta(s, u) + w(u, v)$. ■

Upper-bound property: Always have $v.d \ge \delta(s, v)$ for all v. Once $v.d$ gets down to $\delta(s, v)$, it never changes.

Proof Initially true.

Suppose there exists a vertex such that $v.d < \delta(s, v)$.

Without loss of generality, v is first vertex for which this happens.

Let u be the vertex that causes $v.d$ to change.

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

So,

$$
\begin{aligned}
v.d &< \delta(s, v) \\
&\le \delta(s, u) + w(u, v) \quad \text{(triangle inequality)} \\
&\le u.d + w(u, v) \quad\quad (v \text{ is first violation)} \\
\Rightarrow v.d &< u.d + w(u, v) .
\end{aligned}
$$

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

Once $v.d$ reaches $\delta(s, v)$, it never goes lower. It never goes up, since relaxations only lower shortest-path estimates. ■

No-path property: If $\delta(s, v) = \infty$, then $v.d = \infty$ always.

Proof $v.d \geq \delta(s, v) = \infty \Rightarrow v.d = \infty$. ■

Convergence property: If $s \leadsto u \to v$ is a shortest path, $u.d = \delta(s, u)$, and edge (u, v) is relaxed, then $v.d = \delta(s, v)$ afterward.

Proof After relaxation:

$$\begin{aligned} v.d &\leq u.d + w(u, v) && \text{(RELAX code)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && \text{(lemma—optimal substructure)} \end{aligned}$$

Since $v.d \geq \delta(s, v)$, must have $v.d = \delta(s, v)$. ■

Path-relaxation property: Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If the edges of p are relaxed, *in the order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $v_k.d = \delta(s, v_k)$.

Proof Induction to show that $v_i.d = \delta(s, v_i)$ after (v_{i-1}, v_i) is relaxed.

Basis: $i = 0$. Initially, $v_0.d = 0 = \delta(s, v_0) = \delta(s, s)$.

Inductive step: Assume $v_{i-1}.d = \delta(s, v_{i-1})$. Relax (v_{i-1}, v_i). By convergence property, $v_i.d = \delta(s, v_i)$ afterward and $v_i.d$ never changes. ■

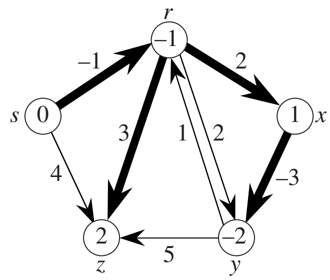
The Bellman-Ford algorithm

- Allows negative-weight edges.
- Computes $v.d$ and $v.\pi$ for all $v \in V$.
- Returns TRUE if no negative-weight cycles reachable from s, FALSE otherwise.

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

Time: $O(V^2 + VE)$. The first **for** loop makes $|V| - 1$ passes over the edges, and each pass takes $\Theta(V + E)$ time. We use O rather than Θ because sometimes $< |V| - 1$ passes are enough (Exercise 22.1-3).

Example



Values you get on each pass and how quickly it converges depends on order of relaxation.

But guaranteed to converge after $|V| - 1$ passes, assuming no negative-weight cycles.

Proof Use path-relaxation property.

Let v be reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from s to v, where $v_0 = s$ and $v_k = v$. Since p is acyclic, it has $\le |V| - 1$ edges, so that $k \le |V| - 1$.

Each iteration of the **for** loop relaxes all edges:

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

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

How about the TRUE/FALSE return value?

- Suppose there is no negative-weight cycle reachable from s.

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

  $$\begin{aligned} v.d &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \quad \text{(triangle inequality)} \\ &= u.d + w(u, v) \ . \end{aligned}$$

  So BELLMAN-FORD returns TRUE.

- Now suppose there exists negative-weight cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$, reachable from s.

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

  Suppose (for contradiction) that BELLMAN-FORD returns TRUE.

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

  Sum around c:

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

Each vertex appears once in each summation $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d \Rightarrow$

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

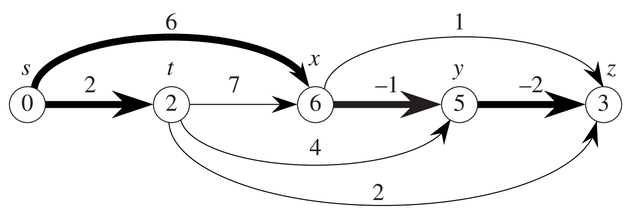
Contradicts c being a negative-weight cycle. ■

Single-source shortest paths in a directed acyclic graph

Since a dag, we're guaranteed no negative-weight cycles.

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

Example



Time

$\Theta(V + E)$.

Correctness

Because vertices are processed in topologically sorted order, edges of *any* path must be relaxed in order of appearance in the path.
$\Rightarrow$ Edges on any shortest path are relaxed in order.
$\Rightarrow$ By path-relaxation property, correct. ■

Dijkstra's algorithm

No negative-weight *edges*.

Essentially a weighted version of breadth-first search.

- Instead of a FIFO queue, uses a priority queue.
- Keys are shortest-path weights ($v.d$).
- Can think of waves, like BFS.

- A wave emanates from the source.
- The first time that a wave arrives at a vertex, a new wave emanates from that vertex.
- The time it takes for the wave to arrive at a neighboring vertex equals the weight of the edge. (In BFS, each wave takes unit time to arrive at each neighbor.)

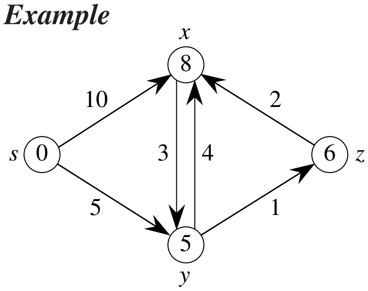
Have two sets of vertices:

- $S =$ vertices whose final shortest-path weights are determined,
- $Q =$ priority queue $= V - S$.

```
DIJKSTRA(G, w, s)
 INITIALIZE-SINGLE-SOURCE(G, s)
 S = ∅
 Q = ∅
 for each vertex u ∈ G.V
     INSERT(Q, u)
 while Q ≠ ∅
     u = EXTRACT-MIN(Q)
     S = S ∪ {u}
     for each vertex v in G.Adj[u]
         RELAX(u, v, w)
         if the call of RELAX decreased v.d
             DECREASE-KEY(Q, v, v.d)
```

- Looks a lot like Prim's algorithm, but computing $v.d$, and using shortest-path weights as keys.
- Dijkstra's algorithm can be viewed as greedy, since it always chooses the "lightest" ("closest"?) vertex in $V - S$ to add to S.

Example



Order of adding to S: s, y, z, x.

Correctness

We will show that at the start of each iteration of the **while** loop, $v.d = \delta(s, v)$ for all $v \in S$. The algorithm terminates when $S = V$, so that $v.d = \delta(s, v)$ for all $v \in V$.

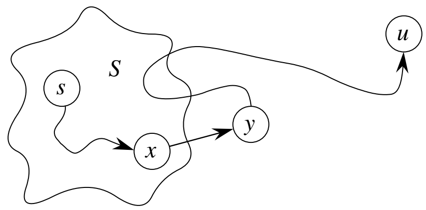
The proof is by induction on the number of iterations of the **while** loop, i.e., on $|S|$. The bases are for $|S| = 0$, so that $S = \emptyset$ and the claim is trivially true, and for $|S| = 1$, so that $S = \{s\}$ and $s.d = \delta(s, s) = 0$.

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

Inductive step: The algorithm extracts vertex u from $V - S$. Because the algorithm adds u into S, we need to show that $u.d = \delta(s, u)$ at that time. If there is no path from s to u, then we are done, by the no-path property.

If there is a path from s to u:

- Let y be the first vertex on a shortest path from s to u that is *not* in S.
- Let $x \in S$ be the predecessor of y on that shortest path.
- Could have $y = u$ or $x = s$.



- y appears no later than u on the shortest path and all edge weights are nonnegative $\Rightarrow \delta(s, y) \le \delta(s, u)$.
- How we chose $u \Rightarrow u.d \le y.d$ at the time u is extracted from $V - S$.
- Upper-bound property $\Rightarrow \delta(s, u) \le u.d$.
- $x \in S \Rightarrow x.d = \delta(s, x)$. Edge (x, y) was relaxed when x was added into S. Convergence property $\Rightarrow$ set $y.d = \delta(s, y)$ at that time.
- Thus, we have $\delta(s, y) \le \delta(s, u) \le u.d \le y.d$ and $y.d = \delta(s, y) \Rightarrow \delta(s, y) = \delta(s, u) = u.d = y.d$.
- Hence, $u.d = \delta(s, u)$. Upper-bound property $\Rightarrow u.d$ doesn't change afterward. ■

Analysis

$|V|$ INSERT and EXTRACT-MIN operations.
$\le |E|$ DECREASE-KEY operations.

Like Prim's algorithm, depends on implementation of priority queue.

- If binary heap, each operation takes $O(\lg V)$ time $\Rightarrow O(E \lg V)$.
- If a Fibonacci heap:
  - Each EXTRACT-MIN takes $O(1)$ amortized time.
  - There are $\Theta(V)$ INSERT and EXTRACT-MIN operations, taking $O(\lg V)$ amortized time each.
  - Therefore, time is $O(V \lg V + E)$.

Difference constraints

Special case of linear programming.

Given a set of inequalities of the form $x_j - x_i \le b_k$.