Single-source Shortest Path

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Dec 5 & 7, 2017

Single-source Shortest Path (Chap 24)

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

Input:

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

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:

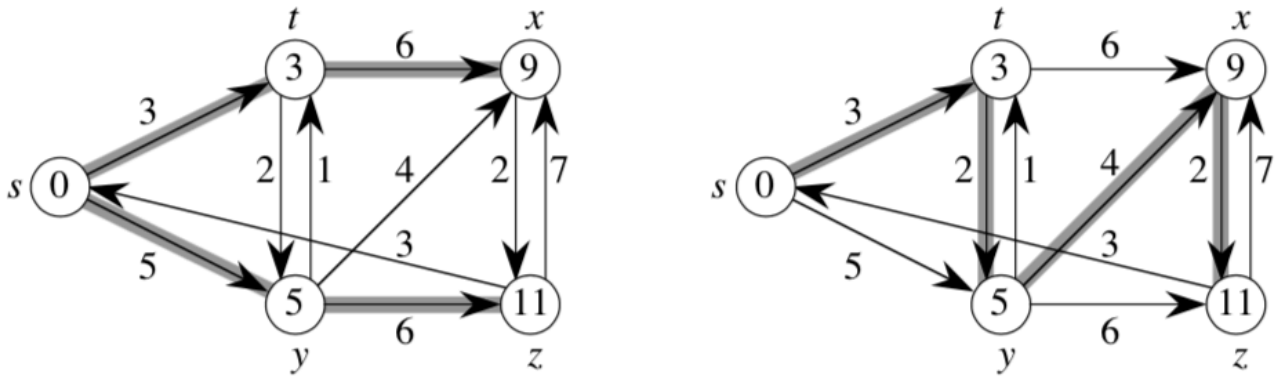
$$\delta(u, v) = \begin{cases} \min \{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there exists a path } u \rightsquigarrow v \text{ ,} \\ \infty & \text{otherwise .} \end{cases}$$

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

Example

Example: shortest paths from s

[d values appear inside vertices. Shaded edges show shortest paths.]



This example shows that the 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,
- we want to minimize.

Examples: time, cost, penalties, loss.

Generalization of breadth-first search to weighted graphs.

Variants

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.

Edge weights

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.

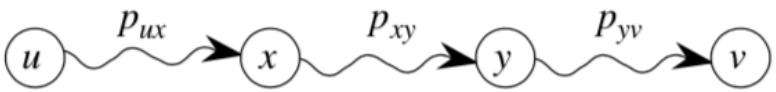
Optimal Substructure

Optimal substructure

Lemma

Any subpath of a shortest path is a shortest path.

Proof Cut-and-paste.



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

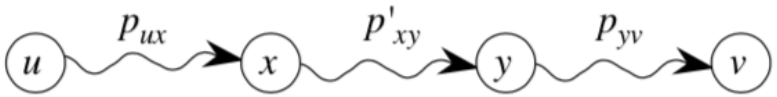
Then $\delta(u, v) = w(p) = w(p_{ux}) + w(p_{xy}) + w(p_{yv})$.

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

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

Optimal Substructure

Construct p':



Then

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

So p wasn't a shortest path after all!

■ (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.
- Zero-weight: no reason to use them $\Rightarrow$ assume that our solutions won't use them.

Algorithm

Output of single-source shortest-path algorithm

For each vertex $v \in V$:

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

Algorithm

Initialization

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

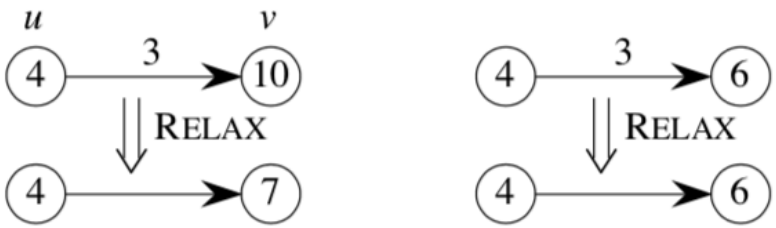
INIT-SINGLE-SOURCE(V, s)
for each $v \in V$
    do $d[v] \leftarrow \infty$
        $\pi[v] \leftarrow$ NIL
$d[s] \leftarrow 0$

Relaxing an edge (u, v)

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

Algorithm

RELAX(u, v, w)
if $d[v] > d[u] + w(u, v)$
  then $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$



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

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

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

Shortest-paths properties

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

Triangle inequality

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

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

Properties

Upper-bound property

Always have $d[v] \geq \delta(s, v)$ for all v. Once $d[v] = \delta(s, v)$, it never changes.

Proof Initially true.

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

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

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

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

So,

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

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

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

Properties

No-path property

If $\delta(s, v) = \infty$, then $d[v] = \infty$ always.

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

Convergence property

If $s \rightsquigarrow u \rightarrow v$ is a shortest path, $d[u] = \delta(s, u)$, and we call RELAX(u, v, w), then $d[v] = \delta(s, v)$ afterward.

Proof After relaxation:

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

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

Properties

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 we relax, *in order*, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $d[v_k] = \delta(s, v_k)$.

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

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

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

Bellman-Ford Algorithm

The Bellman-Ford algorithm

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

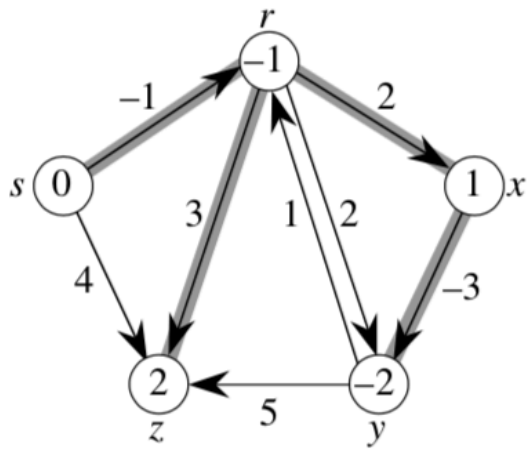
BELLMAN-FORD(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
for $i \leftarrow 1$ to $|V| - 1$
  do for each edge $(u, v) \in E$
    do RELAX(u, v, w)
for each edge $(u, v) \in E$
  do if $d[v] > d[u] + w(u, v)$
    then return FALSE
return TRUE

Core: The first **for** loop relaxes all edges $|V| - 1$ times.

Time: $\Theta(VE)$.

Bellman-Ford Algorithm

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.

Bellman-Ford Algorithm

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 $\leq |V| - 1$ edges, so $k \leq |V| - 1$.

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, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. ■

Bellman-Ford Algorithm

How about the TRUE/FALSE return value?

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

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

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

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

Bellman-Ford Algorithm

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

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

Sum around c:

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

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

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

This contradicts c being a negative-weight cycle! ■

Single-source Shortest Path on DAG

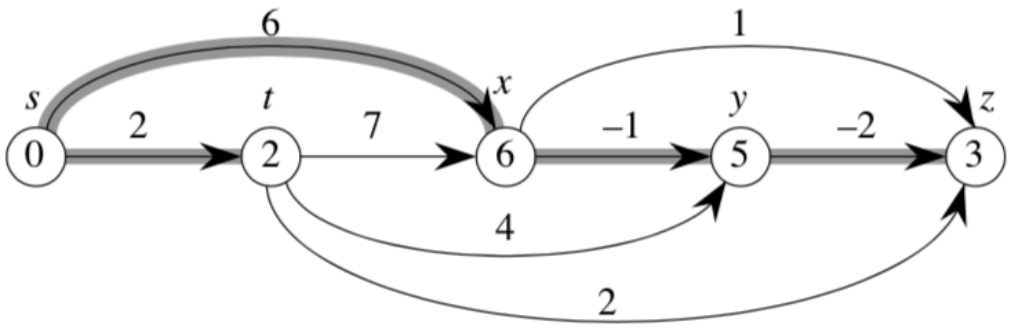
Single-source shortest paths in a directed acyclic graph

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

DAG-SHORTEST-PATHS(V, E, w, s)
topologically sort the vertices
INIT-SINGLE-SOURCE(V, s)
for each vertex u, taken in topologically sorted order
    do for each vertex $v \in Adj[u]$
        do RELAX(u, v, w)

Single-source Shortest Path on DAG

Example:



Time: $\Theta(V + E)$.

Correctness: Because we process vertices 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 ($d[v]$).

Have two sets of vertices:

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

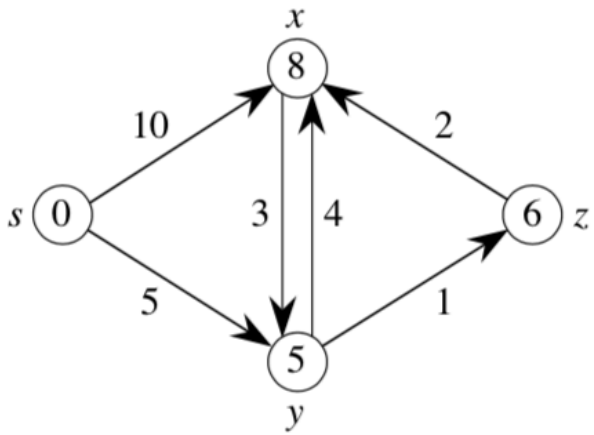
Dijkstra's Algorithm

```
DIJKSTRA(V, E, w, s)
INIT-SINGLE-SOURCE(V, s)
S ← ∅
Q ← V          ▷ i.e., insert all vertices into Q
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

- Looks a lot like Prim's algorithm, but computing $d[v]$, 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.

Dijkstra's Algorithm

Example:



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

Correctness:

> **Loop invariant:** At the start of each iteration of the **while** loop, $d[v] = \delta(s, v)$ for all $v \in S$.

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

Termination: At end, $Q = \emptyset \Rightarrow S = V \Rightarrow d[v] = \delta(s, v)$ for all $v \in V$.

Dijkstra's Algorithm

Maintenance: Need to show that $d[u] = \delta(s, u)$ when u is added to S in each iteration.

Suppose there exists u such that $d[u] \neq \delta(s, u)$. Without loss of generality, let u be the first vertex for which $d[u] \neq \delta(s, u)$ when u is added to S.

Observations:

- $u \neq s$, since $d[s] = \delta(s, s) = 0$.
- Therefore, $s \in S$, so $S \neq \emptyset$.
- There must be some path $s \rightsquigarrow u$, since otherwise $d[u] = \delta(s, u) = \infty$ by no-path property.

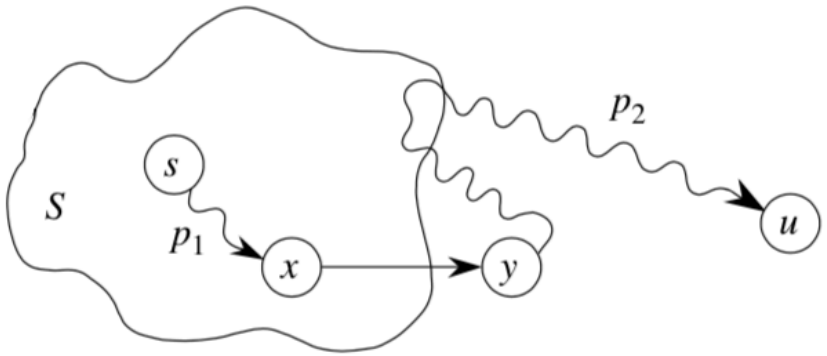
So, there's a path $s \rightsquigarrow u$.

This means there's a shortest path $s \overset{p}{\rightsquigarrow} u$.

Just before u is added to S, path p connects a vertex in S (i.e., s) to a vertex in $V - S$ (i.e., u).

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

Dijkstra's Algorithm



Decompose p into $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$. (Could have $x = s$ or $y = u$, so that p_1 or p_2 may have no edges.)

Claim

$d[y] = \delta(s, y)$ when u is added to S.

Proof $x \in S$ and u is the first vertex such that $d[u] \neq \delta(s, u)$ when u is added to $S \Rightarrow d[x] = \delta(s, x)$ when x is added to S. Relaxed (x, y) at that time, so by the convergence property, $d[y] = \delta(s, y)$. ■ (claim)

Dijkstra's Algorithm

Now can get a contradiction to $d[u] \neq \delta(s, u)$:

y is on shortest path $s \rightsquigarrow u$, and all edge weights are nonnegative
$\Rightarrow \delta(s, y) \leq \delta(s, u) \Rightarrow$

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

Also, both y and u were in Q when we chose u, so

$d[u] \leq d[y] \Rightarrow d[u] = d[y]$.

Therefore, $d[y] = \delta(s, y) = \delta(s, u) = d[u]$.

Contradicts assumption that $d[u] \neq \delta(s, u)$. Hence, Dijkstra's algorithm is correct. ■

Dijkstra's Algorithm

Analysis: 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 $O(V)$ other operations, taking $O(\lg V)$ amortized time each.
  - Therefore, time is $O(V \lg V + E)$.