COMP251: Single source shortest paths

Jérôme Waldispühl
School of Computer Science
McGill University
Based on (Cormen et al., 2002)

Problem
What is the shortest road to go from one city to another?
Example: Which road should I take to go from Montréal to Boston (MA)?
Variants:
• What is the fastest road?
• What is the cheapest road?

Modeling as graphs
Input:
• Directed graph $G = (V, E)$
• Weight function $w : E \rightarrow \mathbb{R}$
Weight of path $p = (v_0, v_1, ..., v_k)$
$= \sum_{i=0}^{k} w(v_{i-1}, v_i)$
= sum of edge weights on path $p$.
Shortest-path weight $u$ to $v$: $\delta(u,v) = \begin{cases} w(p) : \text{if there exists a path } u \rightarrow v, \\ \infty : \text{otherwise.} \end{cases}$
Shortest path $u$ to $v$ is any path $p$ such that $w(p) = \delta(u,v)$.
Generalization of breadth-first search to weighted graphs.

Example
Shortest paths are organized as a tree.
Vertices store the length of the shortest path from $s$.

Example
Shortest paths are not necessarily unique!
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 \( v \). Note: No way to known that is better in worst case than solving the single-source problem!
- **All-pairs**: Find shortest path from \( u \) to \( v \) for all \( u, v \in V \).

Negative weight edges

Negative weight edges can create issues.

**Why?** 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.

**When?** If they are reachable from the source. OK, as long as no negative-weight cycles are reachable from the source.

**Who?** Some algorithms work only if there are no negative-weight edges in the graph. We must specify when they are allowed and not.

Cycles

Shortest paths cannot contain cycles:

- **Negative-weight**: Already ruled out.
- **Positive-weight**: we can get a shorter path by omitting the cycle.
- **Zero-weight**: no reason to use them \( \Rightarrow \) assume that our solutions will not use them.

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

**Contradiction of the hypothesis that \( p \) is the shortest path!**

Optimal substructure

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

**Proof**: (cont’d)

Now suppose there exists a shorter path \( p' \), \( x \sim_{p'} y \).

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

Then \( w(p') = w(p_{ux}) + w(p'_{xy}) + w(p_{yv}) < w(p_{ux}) + w(p_{xy}) + w(p_{yv}) = w(p) \).

**Contradiction of the hypothesis that \( p \) is the shortest path!**

Customized breadth-first search

Vertices count the number of edges used to reach them.
Can we generalize BFS to use edge weights?

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, $\pi[v] = \text{NIL}$.
  - It induces a tree - **shortest-path tree** (see proof in textbook).

Algorithm structure

1. Initialization
2. Scan vertices and relax edges

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

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

Relaxing an edge

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

Triangle inequality

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

Proof:
Weight of shortest path \( u \rightarrow v \) is \( \leq \) weight of any path \( u \rightarrow v \).
Path \( u \rightarrow x \rightarrow v \) is a path \( u \rightarrow v \), and if we use a shortest path \( u \rightarrow x \) and \( x \rightarrow v \), its weight is \( \delta(u, x) + \delta(x, v) \).

| \( u \) | 3 | \( v \) |
|\( u \) | 3 | \( v \) |

Upper bound property

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

Proof:
Initially true.
Suppose there exists a vertex such that \( d[v] < \delta(s, v) \).
Assume \( v \) is first vertex for which this happens.
Let \( u \) be the vertex that causes \( d[v] \) to change.
Then \( d[v] = d[u] + \delta(u, v) \).
\( d[v] < \delta(s, v) \leq d(s, u) + \delta(u, v) \leq d(u) + \delta(u, v) \Rightarrow d[v] < d(u) + \delta(u,v) \).
(v is first violation)
Contradicts \( d[v] = d[u] + \delta(u, v) \).

| \( u \) | \( u \) | \( v \) |

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:
1. \( s \rightarrow u \rightarrow v \) is a shortest path,
2. \( d[u] = \delta(s, u) \),
3. we call \( \text{RELAX}(u, v, w) \),
then \( d[v] = \delta(s, v) \) afterward.

Proof:
After relaxation:
\[
d[v] \leq d[u] + w(u, v) \tag{\text{RELAX code}} \\
= \delta(s, u) + w(u, v) \\
= \delta(s, v) \tag{\text{lemma-optimal substructure}}
\]
Since \( d[v] \geq \delta(s, v) \), must have \( d[v] = \delta(s, v) \).
**Path-relaxation property**

Let \( p = (v_0, v_1, \ldots, v_k) \) 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.

**Single-source shortest paths in a DAG**

Since a DAG, we are guaranteed no negative-weight cycles.

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

---

**Example**

```
  6
  7
  2
  s

  1
  4
  y
  2
  z
```

**Example**

```
  6
  7
  2
  s

  1
  4
  y
  2
  z
```

**Example**

```
  6
  7
  2
  s

  1
  4
  y
  2
  z
```

**Example**

```
  6
  7
  2
  s

  1
  4
  y
  2
  z
```
Single-source shortest paths in a DAG

\[
\text{DAG-SHORTEST-PATHS}(V, E, w, s) \\
topologically sort the vertices \\
\text{INIT-SINGLE-SOURCE}(V, s) \\
\text{for each vertex } u \text{ in topological order do} \\
\quad \text{for each vertex } v \in \text{Adj}[u] \text{ do} \\
\quad \quad \text{RELAX}(u, v, w) \\
\]

Time: \((V + E)\).
Correctness: Because we process vertices in topologically sorted order, edges of any path must be relaxed in order of appearance in the path.
⇒ Edges on any shortest path are relaxed in order.
⇒ By path-relaxation property, correct.

Dijkstra’s algorithm

\[
\text{DIJKSTRA}(V, E, w, s) \\
\text{INIT-SINGLE-SOURCE}(V, s) \\
S \leftarrow \emptyset \\
Q \leftarrow V \\
\text{while } Q \neq \emptyset \text{ do} \\
\quad u \leftarrow \text{EXTRACT-MIN}(Q) \\
\quad S \leftarrow S \cup \{u\} \\
\quad \text{for each vertex } v \in \text{Adj}[u] \text{ do} \\
\quad \quad \text{RELAX}(u, v, w) \\
\]

• No negative-weight edges.
• Weighted version of BFS:
  • 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 = \(V - S\).
• Similar Prim’s algorithm, but computing \(d[v]\), and using shortest-path weights as keys.
• Greedy choice: At each step we choose the light edge.
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, $S = \varnothing$, so trivially true.

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

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

Correctness (cont’d)

Show that $d(u) = \delta(s,u)$ when $u$ is added to $S$ in each iteration. 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 is in $V - S$, and let $x \in S$ be $y$ is predecessor.

Decompose $p$ into $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$.

Correctness (cont’d)

Show that $d(u) = \delta(s,u)$ when $u$ is added to $S$ in each iteration. Now can get a contradiction to $d[u] \neq \delta(s, u)$:
$\Rightarrow \delta(s, y) \leq \delta(s, u)$
$\Rightarrow d(y) = \delta(s, y)$
$\leq \delta(s, u)$
$S[d(u)]$ (upper-bound property)

In addition, since $y$ and $u$ were in $Q$ when we chose $u$:
$d[u] \leq d[y] \Rightarrow d[u] = d[y]$ . Therefore, $d[y] = \delta(s, y) = \delta(s, u) = d[u]$.

Contradicts assumption that $d[u] \neq \delta(s, u)$.

Correctness (cont’d)

Show that $d[u] = \delta(s,u)$ when $u$ is added to $S$ in each iteration.
Suppose there exists $u$ such that $d[u] \neq \delta(s,u)$.
Let $u$ be the first vertex for which $d[u] \neq \delta(s,u)$ when $u$ is added to $S$.

- $u \neq s$, since $d[s] = \delta(s,s) = 0$.
- Therefore, $s \in S$, so $S \neq \varnothing$.

- There must be some path $s \rightsquigarrow u$. Otherwise $d[u] = \delta(s,u) = \Rightarrow$ by no-path property.
  
So, there is a path $s \rightsquigarrow u$. Thus, there is a shortest path $s \rightsquigarrow u$.

Analysis

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

If binary heap, each operation takes $O(\log V)$ time

$\Rightarrow O(E \log V)$.

Note: We can achieve $O(V \log V + E)$ with Fibonacci heaps.