Shortest Paths—Definitions

- For the next few lectures, we will have a directed graph $G = (V, E)$, and a weight function $w : E \to \mathbb{R}$. 
- The weight of a path $P = \{v_0, v_1, \ldots, v_k\}$ is simply the weight of the edges taken on the sequence of nodes:
  \[
  w(P) = \sum_{i=1}^{k} w_{v_{i-1}, v_i}.
  \]
- We are interested in finding the shortest-path weights from $u$ to $v$, which we will denote $\delta(u, v)$.
- We use the convention that $\delta(u, v) = \infty$ if there is no path from $u$ to $v$ in $G$.

Example

- The example (hopefully) makes it clear that shortest paths are organized as a tree.
- Many algorithms work like a generalization of BFS to weighted graphs.
Shortest Path Variants

- **Single-Source**: Find the shortest path from $s \in V$ to every vertex $v \in V$
- **Single-Destination**: Find the shortest path from every vertex $v \in V$ to a given destination vertex $t \in V$
- **Single-Pair**: Find the shortest path from given $s \in V$ to given $t \in V$. There is now way known that is better (in the worst case) that solving the single-source version.
- **All-Pairs**: Find the shortest path from every $u \in V$ to every vertex $v \in V$

Negative Weight Edges

- In Minimum Spanning Tree, negative weight edges posed no significant challenge to the algorithms. However, for shortest path, this is not the case
- If we have a negative weight cycle, we can just keep going around it, and $\delta(s, v) = -\infty$ for all $v$ on the cycle.
- Some algorithms work only if there are no negative weight-edges in the graph

Just Like DP

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

**Proof.** (Same as DP)

**Lemma**
Shortest paths can’t contain cycles

- (Single Source) shortest-path algorithms produce a label: $d[v] = \delta(s, v)$.
- Initially $d[v] = \infty$, reduces as the algorithm goes, so always $d[v] \geq \delta(s, v)$
- Also produce labels $\pi[v]$, predecessor of $v$ on a shortest path from $s$. 

Initializing

**INIT-SINGLE-SOURCE**($V, s$)

1. for each $v$ in $V$
2. do $d[v] \leftarrow \infty$
3. $\pi[v] \leftarrow \text{NIL}$
4. $d[s] \leftarrow 0$
Relax!

- The algorithms work by improving (lowering) the shortest path estimate $d[v]$.
- This operation is called relaxing an edge $(u, v)$.
- Can we improve the shortest-path estimate for $v$ by going through $u$ and taking $(u, v)$?

**Relax** $(u, v, w)$
1. if $d[v] > d[u] + w_{uv}$
2. then $d[v] = d[u] + w_{uv}$
3. $\pi[v] \leftarrow u$

More Lemmas, (Lemma?)

**Path Relaxation Property**
Let $P = \{v_0, v_1, \ldots v_k\}$ be a shortest path from $s = v_0$ to $v_k$. If the edges $(v_0, v_1), (v_1, v_2), (v_{k-1}, v_k)$ are relaxed in that order, (there can be other relaxations in-between), then $d[v_k] = \delta(s, v_k)$

- **Proof.** Induction. (True for $i = 0$, since $d[s] = 0$). Assume $d[v_{i-1}] = \delta(s, v_{i-1})$, by calling RELAX($v_{i-1}, v_i$), then $d[v_i] = \delta(s, v_i)$ must be a shortest path to $v_i$, and the label can never change.
Bellman-Ford Algorithm

- Works with Negative-Weight Edges
- Returns `true` if there are no negative-weight cycles reachable from `s`, `false` otherwise

**Bellman-Ford**(\(V, E, w, s\))
1. \(\text{Init-Single-Source}(V, s)\)
2. for \(i \leftarrow 1\) to \(|V| - 1\)
3. do for each \((u, v)\) in \(E\)
4. do \(\text{Relax}(u, v, w)\)
5. for each \((u, v)\) in \(E\)
6. do if \(d[v] > d[u] + w_{uv}\)
7. then return `False`
8. return `True`

Analysis

- Here I’ll show Example and Code
- Analysis
  - \(\Theta(|V||E|)\)
- Correctness?
  - Let \(v\) be reachable from \(s\), and let \(P = \{v_0, v_1, \ldots v_k\}\) be a shortest path to \(v\). Each iteration of the for loop relaxes all edges. The first iteration relaxes \((v_0, v_1)\), the next \((v_1, v_2)\), the \(k\)th iteration relaxes \((v_{k-1}, v_k)\), by the path relaxation Lemma, \(d[v] = \delta(s, v)\),

Single Source Shortest Path on a DAG

**DAG-Shortest-Paths**(\(V, E, s, w\))
1. \(\text{Init-Single-Source}(V, s)\)
2. topologically sort the vertices (HOW)
3. for each \(u\) in topologically sorted \(V\)
4. do for each \(v \in \text{Adj}[u]\)
5. do \(\text{RELAX}(u, v, w)\)

SSSP-DAG Analysis and Correctness

- Correctness
  - Since vertices are processed in topologically sorted order, edges of *any* path are relaxed in order of appearance on the path.
  - Thus, edges on any shortest path are relaxed in order.
  - Thus, by the path-relaxation lemma, the algorithm is correct.
- Analysis
Dijkstra’s Algorithm

- Works only if the graph has no negative-weight edges
- This is essentially a weighted version of BFS
  - Instead of a FIFO Queue (like you used for BFS in the lab), use a priority queue
  - Keys (in PQ) are the shortest path weight estimates ($d[v]$)
- In Dijkstra’s Algorithm, we have two sets of vertices
  - $S$: Vertices whose final shortest path weights are determined
  - $Q$: Priority queue: $V \setminus S$

Dijkstra’s Algorithm

```
Dijkstra(V, E, w, s)
1 INIT-SINGLE-SOURCE(V, s)
2 S ← ∅
3 Q ← V
4 while Q ≠ ∅
5 do u ← EXTRACT-MIN(Q)
6 S ← S ∪ {u}
7 for each v ∈ Adj[u]
8 do RELAX(u, v, w)
```

Next Time

- Note: Looks a lot like Prim’s algorithm, but computing $d[v]$, and using the shortest path weights as keys
- Dijkstra’s Algorithm is greedy, since it always chooses the “lightest” vertex in $V \setminus S$ to add to $S$
- Analysis: Like Prim’s Algorithm, depends on the time it takes to perform priority queue operations.
- Suppose we use a binary heap.
  - How many times is whole loop called: $O(|E|)$
  - Inside loop, takes: $(O(\log V))$ to EXTRACT-MIN.
- Dijkstra’s Algorithm runs in $O(E \log V)$, with a binary heap implementation.
- Better heap implementations get it down to $O(V \log V + E)$.

- Today in Lab: A “very” related problem... Traveling Salesman.
- Next Time: More Shortest Paths