Dijkstra's Algorithm: Correctness

Suppose we add vertices $v_1, v_2, \ldots, v_n$ to S, in that order.

- After the kth iteration of the loop, $S = \{v_1, \ldots, v_k\}$.

We prove (by induction on k) that after the k iteration of the loop:

1. $d[v_1] \leq d[v_2] \leq \ldots \leq d[v_k] \leq d[v']$ for $v' \notin S$
   - We add vertices to S in order of distance.
2. $d[v] = \delta(s, v)$ for every element in S.
   - i.e., for $v_1, \ldots, v_k$

Base case–$k = 1$: $v_1 = s$, so 1 and 2 are trivial.

Suppose $k = k' + 1$ and result holds for k'.

Key observation: if t is one of the k closest vertices to s and $p = (s, v_1, \ldots, v_m, t)$ is a shortest path from s to t, then $s, v_1, \ldots, v_m \in S$.

- The only vertices that can precede v on the path are ones that are strictly closer to s.
  - By induction hyp., closer vertices are in S
- Also, must have $\delta(s, t) = \delta(s, v_m) + w(v_m, t)$
  - In general, have only $\delta(s, t) \leq \delta(s, v_m) + w(v_m, t)$
- This depends on distances being *nonnegative.*

Conclusions:

- before kth iteration, the vertex t with minimum d in $S - V$ is one of the kth closest (there may be ties).
- For vertex t, $\delta(s, t) = d[t]$ (induction hypothesis)
- Thus, the vertex added at kth iteration of the algorithm is one of the kth closest.
  - Therefore properties 1 and 2 in induction hold

The Bellman-Ford Algorithm

Bellman-Ford solves single-source shortest-path problems even with negative edge weights.

- It also detects negative-weight cycles

BELLMAN-FORD(G, w, s)

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

Example:

Bellman-Ford: Running Time

- Initialization takes $O(|V|)$
- Go through outer loop (lines 2–4) $|V| - 1$ times
- Go through inner loop (lines 3–4) $|E|$ times
- Total time in loop is $O(|V||E|)$
- Go through loop in lines 5–7 $|E|$ times
- Total running time: $O(|V||E|)$

In general, Bellman-Ford is worse than Dijkstra.

- Dijkstra takes $O(|V| \lg |V| + |E|)$ or $O((|V| + |E|) \lg |V|)$ or $O(|V^2|)$, depending on how we implement priority queues

This is the price we have to pay to deal with negative edge weights.

Bellman-Ford: Correctness

Theorem: If there is no path from s to t with a negative-weight cycle, then $d[t] = \delta(s,t)$ after running Bellman-Ford. Bellman-Ford returns TRUE if there are no negative-weight cycles in G reachable from s; otherwise it returns FALSE.

Proof: Suppose there are no negative-weight cycles on a path from s to t and $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from s to t ($s = v_0$, $t = v_k$).

- This means that $(v_0, \ldots, v_j)$ is a shortest path from s to v_j, and there are no negative-weight cycles on any path from s to v_j

We prove by induction on j that after the jth pass through the loop, $d[v_i] = \delta(s, v_i)$ for $i = 0, \ldots, j$.

Base case: $j = 0$ – initially, $d[s] = 0$, so OK.

Inductive step: Suppose $j = j'+1$. Notice that $\delta(s, v_j) = \delta(s, v_{j'}) + w(u, v)$.
By induction, $\delta(s, v_{j'}) = d[v_{j'}]$ after we go through the loop j' times.

After doing $\textsc{Relax}(v_{j'}, v_j, w)$, get

$$d[v_j] \leq d[v_{j'}] + w(v_{j'}, v_j) = \delta(s, v_j)$$

By Relaxation Property,

$$d[v_j] \geq \delta(s, v_j)$$

Conclusion: $d[v_j] = \delta(s, v_j)$ after jth iteration.

If there are no negative-weight cycles on any path between s and t, the shortest path must have at most $V[G]$ vertices (including s and t).

- no vertex is repeated

Thus, $d[t] = \delta(s, t)$ after Bellman-Ford.

If there are no negative-weight cycles reachable from s, then $d[t] = \delta(s, t)$ for all vertices t.

- Thus, $d[v] \leq d[u] + w(u, v)$ for each edge (u, v)

Therefore, Bellman-Ford returns TRUE.

If there is a negative-weight cycle $(v_0, \ldots, v_k)$ with $v_0 = v_k$ reachable from s, then

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

If Bellman-Ford returns TRUE, then $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$. That means

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

Since v_i is reachable, $i = 0, \ldots, k$:

- $d[v_i] < \infty$,
- $\Sigma_{i=1}^{k} d[v_i] < \infty$

Since $v_0 = v_k$,

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

Conclusion:

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

Contradiction!

Therefore, Bellman-Ford returns FALSE if a negative-weight cycle is reachable.

Single-Source Shortest Paths in Dags

There is a better algorithm for single-source shortest paths in dags.

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

Don't have to worry about negative-weight cycles.

- There are none!

Running time is $O(|V| + |E|)$

- Initialization takes $O(|V|)$
- Topological sort takes $O(|V| + |E|)$
  - assuming adjacency-list representation.
- We go through the loop at most $|E|$ times
  - Once for each edge
- Since we don't have to update the priority queue, each iteration through the loop takes $O(1)$ time

Dag Shortest Path: Correctness

Want to show that $d[v] = \delta(s, v)$ after running DAG-SHORTEST-PATH(G, w, s)

If $\delta(s, v) = \infty$, v is not reachable from s, and this is clearly true (since $d[v] \geq \delta(s, v)$).

If $\delta(s, v) < \infty$, let $p = (v_0, \ldots, v_k)$ be a shortest path from s to v ($v_0 = s$, $v_k = v$).

Notice v_{i-1} precedes v_i in the topological sort
(since (v_{i-1}, v_i) is an edge).

- Thus we relax the edges in the order (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k).

We prove that $d[v_i] = \delta(s, v_i)$ when you relax (v_i, v_{i+1}) by induction on i:

- OK if $i = 0$ (since $v_0 = s$)
- Note that $\delta(s, v_{i+1}) = \delta(s, v_i) + w(v_i, v_{i+1})$, so RELAX$(v_i, v_{i+1},$ guarantees that $d[v_{i+1}] = \delta(s, v_{i+1})$.

An Application: Finding Longest Paths

In job scheduling, the vertices represent jobs and the edges represent necessary precedence

- there is an edge from u to v if job u must be completed before job v can begin
- the weight of (u, v) is the amount of time required to do u.

The longest path in the graph is the *critical path*.

- This gives you the time required to perform the longest sequence of jobs, so the total running time of the process.
  - It may make more sense to put the weight on the vertex, not the edge.

If the graph is a dag, we can find the longest path easily:

- replace each weight w by $-w$, and find the shortest path

Minimum Spanning Trees

A *spanning tree* of a graph $G(V, E)$ is a connected acyclic subgraph of G, which includes all the vertices in V and some edges from E.

A *minimum spanning tree (MST)* is a spanning tree is a spanning tree that uses the least number of edges among all spanning trees.

- more generally, we assume that edges have weights, and we want a spanning tree of minimum total weight
  - This assumes are given a graph $G = (V, E)$ and a weight function $w : E \to \mathbf{R}$
  - minimum spanning tree is actually short for "minimum-weight spanning tree"
- A graph may have more than one MST

Think of a MST as a "backbone"; a minimal set of edges that will let you get everywhere in a graph.

MSTs come up all the time:

- E.g., finding a minimal wiring of a set of pins.
- Find a minimal number of messages you have to send to get a message to everyone.

A Generic Algorithm for Building MSTs

We're going to build the spanning tree step by step, adding one edge at a time.

- Invariant: at all times, we have a subgraph of some MST

If A is a set of edges contained in some MST, $(u, v) \in E$ is *safe* for A if $A \cup \{(u, v)\}$ is also a subset of some MST.

Generic-MST$(G(V, E), w)$

$A \leftarrow \emptyset$
while A is not a spanning tree
    do find an edge $(u, v) \notin A$ safe for A
        $A \leftarrow A \cup \{(u, v)\}$
return A

This will clearly work:

- A is always a subset of some MST.
- If A is not a MST, then there must always be some edge $(u, v) \notin A$ safe for A
  - The hard part is finding it!

Recognizing Safe Edges

A *cut* $(S, V - S)$ of an undirected graph $G(V, E)$ is a way of splitting it into two parts.

- An edge (u, v) *crosses the cut* if one of its endpoints is in S, the other in $V - S$
- A cut *respects* a set A of edges if no edge in A crosses the cut
- A *light edge* is an edge of minimum weight crossing a cut
  - There may be more than one light edge

Theorem: If A is included in a MST for $G(V, E)$ and $(S, V - S)$ is a cut that respects A, then any light edge (u, v) crossing $(S, V - S)$ is safe for A.

Proof: Let T be a MST containing A.

- If T contains (u, v), we are done
- If not, construct MST T' containing $A \cup \{u, v\}$

Since T is a MST, there must be a path in T from u to v. Adding (u, v) gives us a cycle.

Since (u, v) crosses from S to $V - S$, there must be another edge (x, y) on the cycle that also crosses from S to $V - S$.

- (x, y) can't be in A, since A respects the cut.
- $T' = T - \{(x, y)\} \cup \{(u, v)\}$ must be a spanning tree.
- Since (u, v) is light, we must have $w(u, v) \leq w(x, y)$.
- Therefore T' is a MST that contains $A \cup \{u, v\}$.
- Therefore (u, v) is safe for A.