The single-source shortest path problem (SSSP) is the problem of finding a minimum weight (shortest) path from a specific source node $s$ in a directed graph $G = (V, E)$ with edge weights to every other node in $V$.

**Input:**
- A directed graph $G = (V, E)$ with edge weights.
- A specific source node $s$.

**Goal:**
- Find a minimum weight (shortest) path from $s$ to every other node in $V$.

It turns out that, in the worst case, finding the shortest path to a single node $t$ is no easier than finding the shortest paths to all other nodes.

**Single-Source Shortest Paths (Ch. 24)**

Weights in SSSP algorithms include distances, times, hops, cost, etc. Used in lots of routing applications.

Note: BFS finds the shortest paths for the special case when all edge weights are 1. Running time = $O(V + E)$

**Dijkstra’s SSSP Algorithm**

1. $Q = \emptyset$ //** $Q$ is a priority queue.
2. $d[s] = 0$ and insert $s$ into $Q$.
3. for each $v \neq s$.
4. \[ d[v] = \infty \]
5. insert $v$ into $Q$ with key $d[v]$.
6. while $Q \neq \emptyset$.
7. $u = \text{extract-min}(Q)$.
8. for each outgoing neighbor of $u$.
9. \[ d[v] = \min\{d[v], d[u] + w(u,v)\} \]
10. endfor.
11. endwhile.

**Procedure relax (s, y)**

\[ d[y] = \min\{d[y], d[s] + w(s,y)\} \]

**Tree nodes are nodes extracted from $Q$ (in $S$, the set of shortest paths).**

**Fringe nodes are nodes in $Q$ with $d[v] < \infty$**

**Unseen nodes are nodes in $Q$ with $d[v] = \infty$**

**Negative-weight cycles**

Some graphs may have negative-weight cycles and these are a problem for SSSP algorithms.

What is the shortest path from $a$ to $d$?

- path $a \rightarrow c \rightarrow e \rightarrow d = -12 + 3 - 2 = -11$
- path $a \rightarrow c \rightarrow e \rightarrow d \rightarrow a \rightarrow c \rightarrow e \rightarrow d = -12 + 3 - 2 + 10 - 12 + 3 - 2 = -13$

If we keep going around the cycle $(d \rightarrow a \rightarrow c \rightarrow e \rightarrow d)$, we keep shortening the weight of the path. So the shortest path has weight $-\infty$.

To avoid this problem, we require that the graph has no negative weight cycles, otherwise the solution does not exist.

**Question:** Can a shortest path contain a cycle?

**No.**

Suppose we have a shortest path $p = \langle v_1, v_2, ..., v_k \rangle$ and $c = \langle v_i, v_{i+1}, ..., v_j \rangle$ is a positive-weight cycle on $p$ so that $v_i = v_j$ and $w(c) > 0$. Then the path (obtained from splicing out $c$)

$p' = \langle v_1, v_2, ..., v_{i-1}, v_{i+1}, v_{i+2}, ..., v_j, v_{j+1}, ..., v_k \rangle$

has $w(p') = w(p) - w(c) < w(p)$. So $p'$ can’t be a shortest path.

Therefore, we can assume, wlog, that shortest paths have no cycles.

**Tree nodes are nodes extracted from $Q$ (in $S$, the set of shortest paths).**

**Fringe nodes are nodes in $Q$ with $d[v] < \infty$**

**Unseen nodes are nodes in $Q$ with $d[v] = \infty$**

**Algorithm SSSP-Dijkstra (G, s)**

1. $Q = \emptyset$ //** $Q$ is a priority queue.
2. \[ d[s] = 0 \] and insert $s$ into $Q$.
3. **for each** $v \neq s$.
4. \[ d[v] = \infty \]
5. **insert** $v$ into $Q$ with key $d[v]$.
6. **while** $Q \neq \emptyset$.
7. **$u = \text{extract-min}(Q)$**.
8. **for each** (outgoing) neighbor of $u$.
9. \[ d[v] = \min\{d[v], d[u] + w(u,v)\} \]
10. **endfor**.
11. **endwhile**.

**Trace execution of Dijkstra’s algorithm on graph below.**
Algorithm SSSP - Dijkstra

1. \( Q = \emptyset \) \hfill (** Q is a priority queue \\
2. \( d[s] = 0 \) and insert \( s \) into \( Q \) \\
3. for each \( v \neq s \) \\
4. \( d[v] = \infty \) \\
5. insert \( v \) into \( Q \) with key \( d[v] \) \\
6. while \( Q \neq \emptyset \) \\
7. \( u = \text{extract-min}(Q) \) \\
8. for each neighbor of \( u \) \\
9. \( d[v] = \min\{d[v], d[u] + w(t, u)\} \)

Running Time of Dijkstra’s SSSP Alg

Steps 1-5: \( O(V) \) time \\
Steps 6-11: \( V \) iterations \\
Suppose extract-min takes \( O(X) \) time. \\
Total: \( O(VX + E) \)

Exercise:

1. Give a simple example of a directed graph with negative weight edges (but no negative weight cycles) for which Dijkstra’s alg produces incorrect answers.
2. Suppose we change line 6 to “while \(|Q| > 1\)” This change causes the while loop to execute \(|V| - 1\) times instead of \(|V|\) times. Is this proposed algorithm correct? Why or why not?
Correctness of Dijkstra’s SSSP Alg

**Lemma:** For all nodes $x \in V$:
(a) if $x \in S$ ($x \notin Q$), then the shortest $s$ to $x$ path only uses nodes in $S$ and $d[x]$ is its weight.
(b) if $x \notin S$ ($x \in Q$), then $d[x]$ is weight of the shortest $s$ to $x$ path, all of whose intermediate nodes are in $S$.

**Proof:**
By induction on $i$, the number of iterations of the while loop.

**Basis:** $i = 1$, $S = \{s\}$, and $d[x] = \infty$ if $x$ is not a neighbor of $s$, and otherwise $d[x] = wt(s, x)$. So both (a) and (b) hold.

**Inductive Hypothesis (IHOP):** Assume true for iteration $i - 1$.

**Lemma:** For all nodes $x \in V$:
(a) if $x \in S$ ($x \notin Q$), then the shortest $s$ to $x$ path only uses nodes in $S$ and $d[x]$ is its weight.
(b) if $x \notin S$ ($x \in Q$), then $d[x]$ is weight of the shortest $s$ to $x$ path, all of whose intermediate nodes are in $S$.

**Induction Step:** Show Lemma is true for iteration $i$.

Let $u$ be the node selected in the $i$th iteration (the node in $Q$ with minimum $d[u]$ value).

**Proof of (a): (Assume $x$ is in $S$)**

**Case 1:** $x \neq u$. Then $x$ was in $S$ before iteration $i$, and by the IHOP, we already had the best $s$ to $x$ path.

**Case 2:** $x = u$. Suppose in contradiction the shortest $s$ to $u$ path uses some node $r$ not in $S$ after iteration $i$.

- $d[u]$ is wt of shortest $s$ to $u$ path with all internal nodes in $S$ (IHOP)
- $d[r]$ is wt of shortest $s$ to $r$ path with all internal nodes in $S$ (IHOP)
- $d[u] \leq d[r]$ since alg picks $u = x$ in iteration $i$.

So...the shortest $s$ to $u$ path can’t go through $r$ since there are no negative weight edges.

**Proof of (b) (cont):** Choose $x$ that is not in $S$ after iteration $i$ ($x \neq u$):

**Case 2:** $x$ is a neighbor of $u$.

The algorithm checks to see if it is better to go from $s$ to $x$ via $u$, or to choose an edge from some other node in $S$.

**Exercise:** Why doesn’t the proof of correctness for Dijkstra’s algorithm go through when negative edge weights are allowed?
The Path-relaxation property

- If \( p = v_0, v_1, \ldots, v_k \) is a shortest path from \( s = v_0 \) to \( v_k \), and the edges of \( p \) are relaxed in the order \( (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \), then \( d[v_k] = \delta(s, v_k) \). This property holds regardless of any other relaxation steps that occur, even if these other steps are intermixed with relaxations of the edges of \( p \).

Algorithm SSSP-Dijkstra \((G, s)\)

1. \( Q = \emptyset \) //** \( Q \) is a priority queue
2. \( d[s] = 0 \) and insert \( s \) into \( Q \)
3. for each \( v \neq s \)
4. \( d[v] = \infty \)
5. insert \( v \) into \( Q \) with key \( d[v] \)
6. while \( Q \neq \emptyset \)
7. \( u = \text{extract-min}(Q) \)
8. for each neighbor \( v \) of \( u \)
9. \( \text{relax}(u, v) \)
10. endfor
11. endwhile

Procedure relax \((x,y)\)

\[ d[y] = \min\{d[y], d[x] + \text{wt}(x,y)\} \]

Note: Dijkstra's algorithm does not ensure that the edges are examined in any particular order, but it does ensure that, at the time \( u \) is extracted from \( Q \), \( d[u] = \delta(s, u) \).

Bellman-Ford SSSP Algorithm

- Computes single-source shortest paths even when some edges have negative weight.
- Can detect if there are any negative-weight cycles in the graph.

Algorithm SSSP-Bellman-Ford \((G, s)\)

1. \( d[s] = 0 \)
2. for each \( v \neq s \)
3. \( d[v] = \infty \)
4. for \( i = 1 \) to \(|V| - 1\)
5. for each edge \((u, v) \in E\)
6. \( d[v] = \min\{d[v], d[u] + \text{wt}(u,v)\} \)
7. for each edge \((u, v) \in E\)
8. if \( d[v] > d[u] + \text{wt}(u,v) \)
9. return false
10. return true

Procedure relax \((x,y)\)

\[ d[y] = \min\{d[y], d[x] + \text{wt}(x,y)\} \]

Bellman-Ford SSSP Algorithm

The algorithm has 2 parts:
Part 1: Computing shortest paths tree:
- \(|V| - 1\) iterations.
- Iteration \( i \) computes the shortest path from \( s \) using paths of up to \( i \) edges.

Part 2: Checking for negative-weight cycles.

Correctness of Bellman-Ford Algorithm

- **Theorem:** Suppose there are no negative-weight cycles in \( G \). After \(|V| - 1\) iterations of the for loop, \( d[v] = \delta(s,v) \) for all vertices \( v \) that are reachable from \( s \).
- **Proof:**
  - If \( v \) is reachable from \( s \), then there is an acyclic path from \( s \) to \( v \), say \( s = u_0, u_1, u_2, \ldots, u_k = v \), where \( k < |V| \).
  - There are \( k \) edges in this path.
  - By the path relaxation property, after the first pass, \( u_0, u_1 \) is a shortest path; after the second pass, \( u_0, u_1, u_2 \) is a shortest path; after \( k \) passes, \( u_0, u_1, u_2, \ldots, u_k \) is a shortest path.
Complexity of Bellman-Ford Algorithm

- Initialization = $O(V)$
- decrease-key is called $(|V| - 1) \times |E|$ times
- Test for any negative-weight cycle = $O(E)$
- Total: $O(VE)$ -- so more expensive than Dijkstra’s, but also more general, since it works for graphs with negative edge weights.

SSSPs in DAGs

- If the graph is a DAG, we can use a topological sort on the vertices and compute the shortest path from a single source in $O(V + E)$ time

Alternate topological sort

- The in-degree of vertex $u$ is the number of incoming edges incident on $u$. The out-degree of vertex $u$ is the number of outgoing edges incident on $u$.

Property of a DAG

- Why does the previous algorithm work?
- Claim: a DAG G must have some vertex with no incoming edges. Why?

Suppose, in contradiction, that every vertex in G has at least one incoming edge. Choose a vertex $v_0$. Trace the edge incoming at $v_0$ to its source, $v_1$. Since $v_1$ must have an incoming edge, we can follow that edge to its source, $v_2$. If we continue backtracking in this fashion, since there are a finite number of vertices, we will eventually return to a previously visited vertex. At this point, we will have discovered a cycle, which is a contradiction to our assumption that G is a DAG.

Therefore, a DAG has at least one vertex with no incoming edge (a similar argument holds for outgoing edges).

SSSPs in DAGs

Note: This algorithm will work with negative weight edges in the DAG.

DAG-Shortest-Paths (G, s)

1. Topological-Sort(G)
2. $d[s] = 0$
3. for each $v \neq s$
   4. $d[v] = \infty$
5. for each vertex $u$, taken in topologically sorted order
6. for each vertex $v$ adjacent to $u$
7. relax($u$, $v$)

Note: This algorithm will work with negative weight edges in the DAG.

DAG-Shortest-Paths (G, s)

1. Topological-Sort(G)
2. $d[s] = 0$
3. for each $v \neq s$
   4. $d[v] = \infty$
5. for each vertex $u$, taken in topologically sorted order
6. for each vertex $v$ adjacent to $u$
7. relax($u$, $v$)

SSSPs in DAGs

Note: This algorithm will work with negative weight edges in the DAG.

DAG-Shortest-Paths (G, s)

1. Topological-Sort(G)
2. $d[s] = 0$
3. for each $v \neq s$
   4. $d[v] = \infty$
5. for each vertex $u$, taken in topologically sorted order
6. for each vertex $v$ adjacent to $u$
7. relax($u$, $v$)

SSSPs in DAGs

- In a topologically sorted list of vertices, all edges will go from left to right. Once all outgoing edges at a node u have been relaxed, u will never be revisited. Since we process the nodes in topologically-sorted order, the nodes at 1 hop from s will be finished before those at 2 hops, etc. By the path-relaxation property, all shortest paths will be found.

List the shortest path distance from s to every other node in the above graph.