Shortest Paths in a Graph

Fundamental Algorithms
The Problems

Given a directed graph $G$ with edge weights, find

- The shortest path from a given vertex $s$ to all other vertices (*Single Source Shortest Paths*)
- The shortest paths between all pairs of vertices (*All Pairs Shortest Paths*)

where the length of a path is the sum of its edge weights.
Shortest Paths: Applications

- Flying times between cities
- Distance between street corners
- Cost of doing an activity
  - Vertices are states
  - Edge weights are costs of moving between states
Shortest Paths: Algorithms

Single-Source Shortest Paths (SSSP)
  – Dijkstra’s
  – Bellman-Ford
  – DAG Shortest Paths

All-Pairs Shortest Paths (APSP)
  – Floyd-Warshall
  – Johnson’s
A Fact About Shortest Paths

**Theorem:** If $p$ is a shortest path from $u$ to $v$, then any subpath of $p$ is also a shortest path.

**Proof:** Consider a subpath of $p$ from $x$ to $y$. If there were a shorter path from $x$ to $y$, then there would be a shorter path from $u$ to $v$. 
Single-Source Shortest Paths

Given a directed graph with weighted edges, what are the shortest paths from some source vertex $s$ to all other vertices?

Note: shortest path to single destination cannot be done asymptotically faster, as far as we know.
Path Recovery

We would like to find the path itself, not just its length.

We’ll construct a shortest-paths tree:
Shortest-Paths Idea

\( \delta(u,v) \equiv \) length of the shortest path from \( u \) to \( v \).

All SSSP algorithms maintain a field \( d[u] \) for every vertex \( u \). \( d[u] \) will be an estimate of \( \delta(s,u) \). As the algorithm progresses, we will refine \( d[u] \) until, at termination, \( d[u] = \delta(s,u) \). Whenever we discover a new shortest path to \( u \), we update \( d[u] \).

In fact, \( d[u] \) will always be an overestimate of \( \delta(s,u) \):

\[
d[u] \geq \delta(s,u)
\]

We’ll use \( \pi[u] \) to point to the parent (or predecessor) of \( u \) on the shortest path from \( s \) to \( u \). We update \( \pi[u] \) when we update \( d[u] \).
SSSP Subroutine

RELAX(u, v, w)

- (Maybe) improve our estimate of the distance to v by considering a path along the edge (u, v).
- if $d[u] + w(u, v) < d[v]$ then
  - $d[v] \leftarrow d[u] + w(u, v)$ ▷ actually, DECREASE-KEY
  - $\pi[v] \leftarrow u$ ▷ remember predecessor on path
Dijkstra’s Algorithm
(pronounced “DIKE-stra”)

Assume that all edge weights are $\geq 0$.

Idea: say we have a set $K$ containing all vertices whose shortest paths from $s$ are known (i.e. $d[u] = \delta(s,u)$ for all $u$ in $K$).

Now look at the “frontier” of $K$—all vertices adjacent to a vertex in $K$. 

![Diagram of a graph with a set $K$ and vertices adjacent to $K$.]
Dijkstra’s: Theorem

At each frontier vertex \( u \), update \( d[u] \) to be the minimum from all edges from \( K \).

Now pick the frontier vertex \( u \) with the smallest value of \( d[u] \).

Claim: \( d[u] = \delta(s,u) \)
Dijkstra’s: Proof

By construction, \( d[u] \) is the length of the shortest path to \( u \) going through only vertices in \( K \).

Another path to \( u \) must leave \( K \) and go to \( v \) on the frontier.

But the length of this path is at least \( d[v] \), (assuming non-negative edge weights), which is \( \geq d[u] \). ■
Proof Explained

Why is the path through \( v \) at least \( d[v] \) in length?
We know the shortest paths to every vertex in \( K \).
We’ve set \( d[v] \) to the shortest distance from \( s \) to \( v \) via \( K \).
The additional edges from \( v \) to \( u \) cannot decrease the path length.
Dijkstra’s Algorithm, Rough Draft

\[ K \leftarrow \{s\} \]

Update \( d \) for frontier of \( K \)
\[ u \leftarrow \text{vertex with minimum } d \text{ on frontier} \]
\[ \triangleright \text{we now know } d[u] = \delta(s,u) \]
\[ K \leftarrow K \cup \{u\} \]
repeat until all vertices are in \( K \).

\[ K \]

8-ShortestPaths
A Refinement

Note: we don’t really need to keep track of the frontier.

When we add a new vertex $u$ to $K$, just update vertices adjacent to $u$. 
Example of Dijkstra’s
Code for Dijkstra’s Algorithm

```
1  DIJKSTRA(G, w, s) ▷ Graph, weights, start vertex
2     for each vertex v in V[G] do
3         d[v] ← ∞
4         π[v] ← NIL
5     d[s] ← 0
6     Q ← BUILD-PRIORITY-QUEUE(V[G])
7     ▷ Q is V[G] - K
8     while Q is not empty do
9         u = EXTRACT-MIN(Q)
10        for each vertex v in Adj[u]
11           RELAX(u, v, w)       // DECREASE_KEY
```
Running Time of Dijkstra

Initialization: $\Theta(V)$
Building priority queue: $\Theta(V)$
“while” loop done $|V|$ times
  $|V|$ calls of EXTRACT-MIN
Inner “edge” loop done $|E|$ times
  At most $|E|$ calls of DECREASE-KEY

Total time:

$$\Theta(V + V \times T_{\text{EXTRACT-MIN}} + E \times T_{\text{DECREASE-KEY}})$$
Dijkstra Running Time (cont.)

1. Priority queue is an array.
   EXTRACT-MIN in $\Theta(n)$ time, DECREASE-KEY in $\Theta(1)$
   Total time: $\Theta(V + VV + E) = \Theta(V^2)$

2. ("Modified Dijkstra")
   Priority queue is a binary (standard) heap.
   EXTRACT-MIN in $\Theta(lgn)$ time, also DECREASE-KEY
   Total time: $\Theta(Vlgn + Elgn)$

3. Priority queue is Fibonacci heap. (Of theoretical interest only.)
   EXTRACT-MIN in $\Theta(lgn)$,
   DECREASE-KEY in $\Theta(1)$ (amortized)
   Total time: $\Theta(Vlgn + E)$
The Bellman-Ford Algorithm

Handles negative edge weights

Detects negative cycles

Is slower than Dijkstra

[Diagram showing a network with negative weights and a negative cycle]

a negative cycle
Bellman-Ford: Idea

Repeatedly update $d$ for all pairs of vertices connected by an edge.

**Theorem:** If $u$ and $v$ are two vertices with an edge from $u$ to $v$, and $s \Rightarrow u \rightarrow v$ is a shortest path, and $d[u] = \delta(s,u)$,

then $d[u] + w(u,v)$ is the length of a shortest path to $v$.

**Proof:** Since $s \Rightarrow u \rightarrow v$ is a shortest path, its length is $\delta(s,u) + w(u,v) = d[u] + w(u,v)$. ■
Why Bellman-Ford Works

• On the first pass, we find $\delta(s,u)$ for all vertices whose shortest paths have one edge.

• On the second pass, the $d[u]$ values computed for the one-edge-away vertices are correct ($= \delta(s,u)$), so they are used to compute the correct $d$ values for vertices whose shortest paths have two edges.

• Since no shortest path can have more than $|V[G]|$-1 edges, after that many passes all $d$ values are correct.

• Note: all vertices not reachable from $s$ will have their original values of infinity. (Same, by the way, for Dijkstra).
Bellman-Ford: Algorithm

BELLMAN-FORD(G, w, s)
1   foreach vertex v ∈ V[G] do //INIT_SINGLE_SOURCE
2       d[v] ← ∞
3       π[v] ← NIL
4   d[s] ← 0
5   for i ← 1 to |V[G]|-1 do ▷ each iteration is a “pass”
6       for each edge (u,v) in E[G] do
7           RELAX(u, v, w)
8   ▷ check for negative cycles
9       for each edge (u,v) in E[G] do
10          if d[v] > d[u] + w(u,v) then
11             return FALSE
12   return TRUE

Running time: Θ(VE)
What if there is a negative-weight cycle reachable from s?
Assume:  
- $d[u] \leq d[x]+4$
- $d[v] \leq d[u]+5$
- $d[x] \leq d[v]-10$

Adding:
- $d[u]+d[v]+d[x] \leq d[x]+d[u]+d[v]-1$

Because it’s a cycle, vertices on left are same as those on right. Thus we get $0 \leq -1$; a contradiction. So for at least one edge $(u,v)$,

- $d[v] > d[u] + w(u,v)$

This is exactly what Bellman-Ford checks for.
SSSP in a DAG

Recall: a dag is a directed acyclic graph.

If we update the edges in topologically sorted order, we correctly compute the shortest paths.

Reason: the only paths to a vertex come from vertices before it in the topological sort.
SSSP in a DAG Theorem

**Theorem:** For any vertex $u$ in a dag, if all the vertices before $u$ in a topological sort of the dag have been updated, then $d[u] = \delta(s,u)$.

**Proof:** By induction on the position of a vertex in the topological sort.

Base case: $d[s]$ is initialized to 0.

Inductive case: Assume all vertices before $u$ have been updated, and for all such vertices $v$, $d[v] = \delta(s,v)$. (continued)
Proof, Continued

Some edge \((v,u)\) where \(v\) is before \(u\), must be on the shortest path to \(u\), since there are no other paths to \(u\).

When \(v\) was updated, we set \(d[u]\) to

\[
d[u] = d[v] + w(v,u)
\]

\[
= \delta(s,v) + w(v,u)
\]

\[
= \delta(s,u) \square
\]
SSSP-DAG Algorithm

DAG-SHORTEST-PATHS(G,w,s)
1 topologically sort the vertices of G
2 initialize d and $\pi$ as in previous algorithms
3 for each vertex u in topological sort order do
4 for each vertex v in Adj[u] do
5 RELAX(u, v, w)

Running time: $\theta(V+E)$, same as topological sort