Shortest Paths

Given a directed graph $G$, edges have costs $\text{cost}(u \rightarrow v)$.

Path length = sum of individual edge costs.

Want to find shortest paths in $G$.

Shortest paths from a source $s$ form a tree:

![Graph diagram]

If shortest $s \rightarrow t$ path is $s = u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_{k-1} \rightarrow u_k = t$, then shortest $s \rightarrow u_{k-1}$ path is also $s = u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_{k-2} \rightarrow u_{k-1}$.\]
Single-Source Shortest Paths (SSSP):
Find shortest path tree from S.

Point-to-point:
Find shortest S→T path
Algorithm = run SSSP from S
Find T in the tree
[Nothing better known in general!]

All Pairs Shortest Paths (APSP):
Find all shortest paths
Algorithm = run SSSP for all S.

So how to solve SSSP?

Let $C^*(u)$ = true shortest path length to u

Triangle Inequality says:

For all (u→v) edges,

\[ C^*(v) \leq C^*(u) + \text{cost} \ (u\rightarrow v). \]

[Can get to v by taking this edge.]

Generic algorithm:
Start with upper bound $C(\cdot)$ on $C^*(\cdot)$
Repeatedly pick edges synchronously and apply triangle inequality.
More formally:

**Generic \((G, s)\):**

Set \( c(s) = 0 \), \( c(u) = \infty \ \forall u \neq s \)

Repeatedly pick edges \((u,v)\) somehow:

\[
    c(v) \leftarrow \min \left( c(v), \frac{c(u)}{c(u) + \text{cost}(u\rightarrow v)} \right)
\]

"Relax\((u,v)\)"

Image: edge is spring of given length, go from stretched to relaxed

**Lemma:** No matter how edges are picked, \( c(v) \geq c^*(v) \ \forall v \) at all times.

PE starts true.

If it's true at any point, updates have

\[
    c(v) \leftarrow \min \left( c(v), \frac{c(u)}{c(u) + \text{cost}(u\rightarrow v)} \right) \geq c^*(v)
\]

\[
    \geq \min \left( c^*(v), \frac{c^*(u)}{c^*(u) + \text{cost}(u\rightarrow v)} \right) \geq c^*(v)
\]

by the triangle inequality.

Hence it remains true.

When does it get to the true answer?
Lemma

If \( S = u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_k \)
is a shortest \( S \rightarrow u_k \) path, and relax is called
on \( (u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k) \)
in order — possibly with intervening calls before, between and after
— then \( c(u_k) = c^*(u_k) \).

**Proof** We induct on \( k \).

\( k = 1 \Rightarrow u_1 = S, \) so \( c(u_1) = 0 = c^*(u_1) \) to start,
and it never increases.

Otherwise, by induction
\( c(u_{k-1}) = c^*(u_{k-1}) \)
when relax is called on \( (u_{k-1}, u_k) \).
Then this call sets
\[
c(u_k) = \min(c(u_k), c(u_{k-1}) + \text{cost}(u_{k-1} \rightarrow u_k))
\leq c(u_{k-1}) + \text{cost}(u_{k-1} \rightarrow u_k)
= c^*(u_{k-1}) + \text{cost}(u_{k-1} \rightarrow u_k)
= c^*(u_k).
\]
and later calls cannot increase it. \( \square \)
So we need to relax every edge of the path in order.

**Bellman-Ford Algorithm:**

N-1 times:

relax every edge.

The i<sup>th</sup> iteration relaxes every edge

\[ \Rightarrow \text{relaxes the } i\text{th edge on path} \]

Paths have \( \leq n-1 \) edges

(assuming no negative cycles!)

\[ \Rightarrow \text{relaxed all edges in order after } n-1 \text{ iterations} \]

\[ \Rightarrow \text{correct.} \]

**Running time** \( O(mn) \)

Best known algorithm for general graphs!

Can do better if edge lengths nonnegative by Dijkstra's algorithm.
Dijkstra's Algorithm

Bellman-Ford relaxes every edge $n-1$ times. Inefficient.

Dijkstra relaxes each edge once. Works harder to find the right edge to relax.

In each round, Dijkstra "visits" a vertex, relaxing all edges out of the vertex. Chooses the unvisited vertex closest to $S$. [But we don't know all distances yet! So it picks the vertex of minimum $c(s)$]

Dijkstra $(G, s)$:
- $c(u) = \infty \forall u$
- $c(s) = 0$
- $S = \emptyset$

While $S \neq V$:
- Find $u \in (V - S)$ minimizing $c(u)$.
- $S \leftarrow S + \{u\}$
- For each edge $(u,v)$ from $u$ in $E$:
  - $c(v) \leftarrow \min(c(v), c(u) + \text{cost}(u \rightarrow v))$
- "Visit $u" \\
  \{ \text{relax}(u,v) \}
Correctness

For simplicity, suppose \( c^*(u) \) all unique.

[Full proof on Piazza]

Can order \( U \subseteq V \) by distance flows:
\[
S = u_1, u_2, \ldots, u_n
\]
\( c(u_1) < c(u_2) < c(u_3) < \ldots < c(u_n) \)

Lemma:
The \( k^{th} \) node visited = \( u_k \)
and \( c(u_k) = c^*(u_k) \) when it is visited.

Proof trivial for \( k = 1 \).
If true for all \( k < k' \)
then consider the state just before
choosing the \( k^{th} \) node to visit.

Claim: \( c(u_k) = c^*(u_k) \).
Let \( u' = \text{pred}(u_k) = \text{previous node to } u_k \)
in shortest \( S \rightarrow u_k \) path.
\[
c^*(u') = c^*(u_k) - \text{cost}(u' \rightarrow u_k)
\]
\[
\leq c^*(u_k)
\]
because nonnegative edges.
Uniqueness assumption \( \Rightarrow c^*(u') < c^*(u_k) \)
\( \Rightarrow \) \( u' \) before \( u_k \) in order
\( \Rightarrow \) already visited \( u' \) and
\( \Rightarrow \) when visited \( u' \) we set
\[
\frac{c(u_k) \leq \min(c(u_k), c^*(u') + \text{cost}(u' \rightarrow u_k))}{c^*(u_k)}
\]
\( = c^*(u_k) \).
So we have \( c(U_k) = c^*(U_k) \)
when deciding on the \( k \)-th node to visit.
We've already visited \( U_s \) for \( s < k \),
and all other \( i \) have
\[
c(U_i) > c^*(U_i) > c^*(U_k) = c(U_k).
\]

Hence Dijkstra will choose \( U_k \) in the \( k \)-th round,
with
\[
c(U_k) = c^*(U_k).
\]

Since we visit every node, and \( c(U) = c^*(U) \)
when it is visited, Dijkstra eventually gets each \( c(U) = c^*(U) \),
proving correctness.

---

**Running time**

\[
\text{Time} = O(\text{time to relax } M \text{ edges} + \text{time to find } N \text{ vertices to visit})
\]

**Simplest approach:**

Look through all \( V \) to decide
node to visit,
\[
\Rightarrow O(1) \text{ relax, } O(n) \text{ time to find each } U.
\]
\[
\Rightarrow O(m+n^2) = O(n^2) \text{ running time.}
\]

Better than Bellman-Ford!
Better approach: store unvisited vertices, \( \mathcal{V} \setminus \mathcal{S} \), in a binary heap keyed by \( c() \).

Node to visit = delete-min on heap
= \( O(\log n) \) time.

but now \( \text{relax}(I) \) changes a \( c() \)
\( \Rightarrow \) need to bubble up that node in heap
"decrease-key operation"
= \( O(\log n) \) time.

\( \Rightarrow \) total time = \( O(m\log n + n\log n) \)
= \( O(m\log n) \).

Fanciest approach: use a Fibonacci heap

\( \text{delete-min}: O(\log n) \)
\( \text{decrease-key}: O(1) \) (amortized)

\( \Rightarrow \) \( O(m + n\log n) \) time.

[In practice, use a binary heap]