Shortest path problem
- Single destination shortest path
  find a shortest path to given destination vertex \( t \) from each vertex \( v \).
- APSP
- SSSP → Single source shortest path
  - Directed unweighted graphs
  - Directed acyclic graphs with non-negative weights
  - Given a source \( S \), find shortest path to each vertex \( v \) in \( V \).

Definitions
- Graph \( G = (V, E) \), \( V \) set of vertices
  \( E \) set of edges
- Weight \( w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \)
- Shortest path weight \( w(p) = \delta(u, v) \)
  where \( \delta(u, v) = \min(\{ w(p) : u \in p \}) \)

Optimal substructure of a shortest path
- Relies on the property that shortest paths between vertices rely on
  shorter paths within it.
- Dijkstra relies on this as well as other famous algorithms.
- Lemma: Subpaths of shortest paths are shortest paths.

Given \( G(V, E) \) with weight \( w : E \rightarrow \mathbb{R} \), let \( p = \langle v_0, v_1, \ldots, v_k \rangle \) be the shortest path from \( v_0 \) to \( v_k \).

For any \( 0 \leq i \leq j \leq k \), let \( p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle \) be the subpath from \( v_i \) to \( v_j \). \( p_{ij} \) is the shortest path from \( v_i \) to \( v_j \).
Proof

Decompose path \( p \) into \( v_1 \rightarrow p_{ij} \rightarrow v_j \rightarrow p_{jk} \rightarrow v_k \).

Then we have \( w(p) = w(p_{ij}) + w(p_{jk}) + w(p_{ik}) \).

Assume that there is a path \( p' \) from \( v_i \) to \( v_j \) with weight \( w(p') < w(p) \). If \( w(p) \) contradicts the idea that \( p \) is a Sp,

Replace Shortest Paths!

Relaxation

For each vertex \( v \in V \), we keep track of the tentative \( v.d \). This is the upper bound on the weight of a shortest path from source \( s \) to \( v \). \( v.d \) is a shortest-path estimate. \( v.d \) is the actual Sp.

When do you initialize the graph? What do you set \( v.d \) to?

\[ v.d = \infty \]
\[ v.\pi = \text{null} \]

Relax \((u,v)\) basically tests whether or not the shortest path can improve by going through the \( u \) vertex (shortest path to \( v \)).

Relaxation step may decrease the value of the shortest path estimate \( v.d \).
Representing shortest paths

How do we represent a shortest path?

$G(V,E)$, we maintain $\pi$ for a vertex $v \in V$ a predecessor $v' \in \pi$ that is either another vertex or NIL.

No-path property

If there is no path from $s$ to $v$, then we always have $v.d = \infty(s,v) = \infty$

Convergence property

If $s \rightarrow u \rightarrow v$ is a shortest path in $G$ for some $u,v \in V$, and if $u.d = \delta(s,u) \leq \infty$ for any time prior to relaxing edge $(u,v)$, then $v.d = \delta(s,v)$ at all times afterward.

Upper-bound property

We always have $v.d \leq \delta(s,v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s,v)$, it never changes.
Relaxation cont.

\[
\text{Relax (u,v, w):
    
    if } v.d > u.d + w(u,v)
    
    v.d = u.d + w(u,v)
    
    v.\pi = u \quad \# \text{ set predecessor}
\]

Dijkstra's algo (HAND OUT the ALG-O)

Assume \( w(u,v) \geq 0 \) for each edge.

Set \( S \) of vertices whose final shortest path weights from source \( s \) have already been determined.

- Repeatedly pick the vertex \( u \in V - S \) with the
  - min shortest path estimate.
- Add \( u \) to \( S \) and relax all edges leaving \( u \).
- \( \pi_v \) for \( v \) \in \( S \), keyed by \( d \) values
- \( C = |V - S| \) always invariant

\# go step by step w/ the algo

3
EXAMPLE

![Graph Image]

<table>
<thead>
<tr>
<th>v.d</th>
<th>v.Δa</th>
<th>Relaxed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Proof of Correctness

Use a loop invariant

try to show that for each vertex u ∈ V, we have
w.d = δ(S, u) at the time when u is added to set S.
Once we show that w.d = δ(S, u), we can rely on
the upper bound property to prove that this
equality will keep on holding at all times.

Initialize the graph G = S = ∅. Invariant is: true by trivially

Maintenance =)

We want to show that in each iteration of the
algorithm w.d = δ(S, u) for each vertex that is added
to the set S. Focus on the beginning of the

attempt to prove by contradiction that

For the purpose of contradiction, let u be the first
vertex for which w.d ≠ δ(S, u) when it is added to set S.
Look at beginning of while loop to derive contradiction that
w.d = δ(S, u) @ that time by examining shortest path from
S to u. We have to have u ≠ S because S is
the first vertex added to S + S.d = δ(S, s) = 0
B/c u ≠ S we also knew that S ⊆ S before
u is added to the set. Then how to be some path
from s to u. All otherwise w.d = δ(S, u) = 0 by no path, which
violates our assumption that w.d ≠ δ(S, u).
Proof continued

B/c there is at least 1 path, there is a shortest path \( p \) from \( s \) to \( u \). Prior to adding \( u \) to \( S \), path \( p \) connects a vertex in \( S \), namely \( s \), to a vertex in \( V-S \), namely \( u \). Consider the first vertex \( y \) along \( p \) s.t. \( y \in V-S \) and let \( x \in S \) be \( y \)'s predecessor along \( p \). Thus, we can decompose the path into \( s \xrightarrow{p} x \rightarrow y \xrightarrow{p} u \).

We claim that \( y.d = \delta(s,y) \) when \( u \) is added to \( S \). Observe that \( x \notin S \). B/c we chose \( u \) as the first vertex for which \( u.d = \delta(s,u) \) when it is added to \( S \), we had \( x.d = \delta(s,x) \) when \( x \) was added to \( S \). Edge \( (x,y) \) was relaxed at that time, and the claim follows from the convergence property.

We can now get our contradiction to prove that \( u.d = \delta(s,u) \). B/c \( y \) appears before \( u \) on a shortest path from \( s \) to \( u \), and \( \delta \) is a metric and \( \delta \) is non-negative, we have \( \delta(s,y) \leq \delta(s,u) \), and thus we have

\[
\begin{align*}
y.d &= \delta(s,y) \\
&\leq \delta(s,u) \\
&= u.d \quad \text{(by upper bound)}
\end{align*}
\]
But if \( c \) but vertices \( u \) and \( g \) were in \( V-S \) when \( u \) was chosen in line 5, we have \( u.d \neq g.d \). Thus, the two inequalities are equalities.

\[ y.d = \delta (g, y) = \delta (g, u) = u.d \]

Consequently, \( u.d = \delta (g, u) \), which contradicts our choice of \( u \). Thus we conclude that \( u.d = \delta (g, u) \) when \( u \) is added to \( S \) and that this equality is always maintained.
Running Time Analysis

Depends on the implementation of the Q:
Normal linked-list/array $\Rightarrow O(1|E| + |V|^2) = O(|V|^2)$

Main operations are decrease-key $\Rightarrow |E| \text{ operations}$
Extract-min $\Rightarrow |V| \text{ operations}$

If keen to make max-priority queue, the runtime
thus improves to $O((|V|+|E|)\log \log |V|) \Rightarrow O(|E| \log |V|)$

Using more advanced data structures, we can achieve
$O(|E| + |V| \log \log |V|)$

Applications and Limitations

- Telephone network, connect call with the maximum
  bandwidth
- Network routing, lowest latency between links
- Mapping your route along campus
- GPS navigation systems

No cycles!
Similarly (for when Dijkstra is invalid)
Dijkstra's biggest problem is that it cannot
deal with negative weight edges.

This is the two people invented the
Bellman-Ford algorithm. (Richard Bellman/Leslie Ford Jr.)
Bellman-Ford runs slower than Dijkstra's
also, but is more versatile.

Negative cycle would make the "shortest path"
so, here Bellman-Ford will detect
negative cycle.

O(IViE|) more
O(iIVI) same
Why Dijkstra does NOT work for negative weights edges

Consider the graph

```
   5
  / 
 a   4
  
 b  -4
  
 c
```

Find SP from b to c:

Dijkstra will tell you that the SP from b to c = 4, which is larger than the actual shortest path of b-a-c.

This is because Dijkstra is supposed to find a SPT T where T is the union of the shortest paths from s to all other u ∈ V.

Ours is a cycle 3000
pwr SPT

PDIL