Shortest Paths

In the past, we were able to use breadth-first search to find the shortest paths from a source vertex to all other vertices in some graph $G$. We weighed each edge equally so the shortest path between two vertices was the one that contained the fewest edges. Now, we introduce edge weights so the cost of traveling through edges can differ from edge to edge. The shortest path between two vertices is defined to be the path whose sum of edge weights is the least. BFS will not work on weighted graphs since the path with the fewest edges may not be the shortest if the edges it contains are expensive.

There are several variants on the shortest paths problem and the algorithms that we will go over that correspond to solving each problem are in parentheses:

- **Single-source shortest-paths problem:** Find a shortest path from a source vertex to each other vertex in the graph (Bellman-Ford, Dijkstra)

- **Single-destination shortest-paths problem:** Find a shortest path to a destination vertex from each other vertex in the graph (Bellman-Ford/Dijkstra on reversed graph)

- **Single-pair shortest-path problem:** Find a shortest path between a vertex $u$ and a vertex $v$ in a graph (Bellman-Ford, Dijkstra)

- **All-pairs shortest-paths problem:** Find a shortest path between every two vertices in a graph (Bellman-Ford/Dijkstra $V$ times, Floyd-Warshall)

Notation: Shortest Paths

Bellman-Ford and Dijkstra both solve the problem of finding a shortest path from a source vertex to each other vertex in the graph. We will designate the source vertex as $s$. Every vertex $v$ in the graph is augmented with the following parameters:

- **$v.d$** - The weight of the current shortest path from $s$ to $v$. This is initialized to be $\infty$ for all vertices besides the source vertex but decreases as paths are found and shorter paths are discovered. At the end of the algorithm, this will be the weight of the shortest path. $s.d$ is initialized to be 0.

- **$v.\pi$** - The parent vertex of $v$ in the current shortest path. This is initialized to be NIL but gets set to a vertex once a path is discovered from $s$ to $v$. As shorter paths to $v$ are discovered, the parent updates to reflect the change. At the end of the algorithms, this will be the parent of $v$ in the shortest path to $v$. $s.\pi$ will always be NIL.

Also,
w(u, v) is the weight of the edge from vertex u to vertex v

δ(u, v) is the weight of the shortest path from vertex u to vertex v

General structure of S.P. Algorithms (no negative cycles)

Here is a the general structure of shortest paths algorithms for weighted graphs with no negative cycles. Usually what changes between different algorithms is the order in which edges are relaxed.

Initialize: for v ∈ V: v.d ← ∞ v.π ← NIL

s.d ← 0

Main: repeat

select edge (u, v) [somehow]

“Relax” edge (u, v)

if v.d > u.d + w(u, v):

v.d ← u.d + w(u, v)

v.π ← u

until all edges have v.d ≤ u.d + w(u, v)

Running time special case

The running time of a shortest path algorithm heavily depends on the order in which edges are relaxed. In some cases the running time could become exponential.

In a generalized example based on Figure 1, we have n nodes, and the weights of edges in the first 3-tuple of nodes are $2^{2^3}-1$. The weights on the second set are $2^{2^2}-2$, and so on. A pathological selection (using only bottom edges) of edges will result in the initial value of $(v_{n-1}).d$ to be $2 \times (2^{2^2}-1+2^{2^2}-2+\cdots+4+2+1)$. In this ordering, we may then relax the edge of weight 1 that connects $v_{n-3}$ to $v_{n-1}$. This will reduce $(v_{n-1}).d$ by 1. After we relax the edge between $v_{n-5}$ and $v_{n-3}$ of weight 2, $(v_{n-2}).d$ reduces by 2. We then might relax the edges $(v_{n-3}, v_{n-2})$ and $(v_{n-2}, v_{n-1})$ to reduce $(v_{n-1}).d$ by 1. Then, we relax the edge from $v_{n-3}$ to $v_{n-1} again. In this manner, we might reduce $(v_{n-1}).d$ by 1 at each relaxation all the way down to $2^{2^2}-1+2^{2^2}-2+\cdots+4+2+1$. This will take $O(2^2)$ time.

Relaxation

Initializing the algorithms involves setting v.d to ∞ and v.π to NIL. Throughout the course of the algorithms, we will need to update these values to find shortest paths.
The idea is that if we found a path costing $u.d$ from $s$ to $u$ and there is an edge from $u$ to $v$, then the upper bound on the weight of a shortest path from $s$ to $v$ is $u.d + w(u, v)$. We can thus compare $u.d + w(u, v)$ to $v.d$ and update $v.d$ if $u.d + w(u, v)$ is smaller than the current $v.d$. In pseudocode, relaxing the edge $(u, v)$ is:

```
RELAX(u, v):
    if v.d > u.d + w(u, v) ## if we find a shorter path to v through u
        v.d = u.d + w(u, v) ## update current shortest path weight to v
        v.pi = u ## update parent of v in current shortest path to v
```

**Properties of Shortest Paths**

Using our definitions of shortest paths and relaxations, we can come up with several properties. These can all be found in CLRS in chapter 24.

**Triangle inequality:** For any edge $(u, v)$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$. In English, the weight of the shortest path from $s$ to $v$ is no greater than the weight of the shortest path from $s$ to $u$ plus the weight of the edge from $u$ to $v$. Note that the triangle inequality only applies to
the shortest paths between nodes. It does not apply to the edges between nodes in the graph. A counter example is a triangle graph with two edges with weight 2 and one edge with weight 5. The triangle inequality does not hold for these edge weights but does hold for the minimum paths between nodes.

**Optimal substructure:** Let \( \{v_1, v_2, v_3, ..., v_k\} \) be a shortest path that goes from \( v_1 \) to \( v_k \) through the vertices \( v_2 \) through \( v_{k-1} \). Any subpath \( \{v_i, v_{i+1}, ..., v_{j-1}, v_j\} \) must be a shortest path from \( v_i \) to \( v_j \). That is, a shortest path is constructed of shortest paths between any two vertices in the path. Proof by contradiction: if the subpath is not a shortest path between \( v_i \) and \( v_j \) then we can find a shorter path between \( v_1 \) and \( v_k \) by using the shortest path between \( v_i \) and \( v_j \).

**Upper-bound property:** We always have \( v.d \geq \delta(s, v) \) for all vertices \( v \). Once \( v.d = \delta(s, v) \), it never changes.

**No-path property:** If there exists no path from \( s \) to \( v \), \( v.d \) will always be \( \infty \).

**Convergence property:** If a shortest path from \( s \) to \( v \) contains the edge \( (u, v) \) and \( u.d = \delta(s, u) \) before relaxing edge \( (u, v) \), then \( v.d = \delta(s, v) \) at all times after relaxing edge \( (u, v) \).

**Path-relaxation property:** Let \( \{v_1, v_2, v_3, ..., v_k\} \) be a shortest path that goes from \( v_1 \) to \( v_k \). If the edges are relaxed in the order \( (v_1, v_2), (v_2, v_3), \) etc., then \( v_k.d = \delta(s, v_k) \) once the whole path is relaxed.

**Predecessor-subgraph property:** Once \( v.d = \delta(s, v) \) for all vertices \( v \), the predecessor subgraph is a shortest-paths tree rooted at \( s \). The predecessor subgraph is the subgraph of \( G \) that contains all the vertices with a finite distance from \( s \) (i.e. reachable from \( s \)) and only the edges that connect \( v \) to \( v.\pi \).

**Graph Transformation**

**Shortest path with even or odd length**

Given a weighted graph \( G = (V, E, w) \), suppose we only want to find a shortest path with odd number of edges from \( s \) to \( t \). To do this, we can make a new graph \( G' \). For every vertex \( u \) in \( G \), there are two vertices \( u_E \) and \( u_O \) in \( G' \): these represent reaching the vertex \( u \) through even and odd number of edges respectively. For every edge \( (u, v) \) in \( G \), there are two edges in \( G' \): \( (u_E, v_O) \) and \( (u_O, v_E) \). Both of these edges have the same weight as the original. Constructing this graph takes
linear time $O(V + E)$. Then we can run shortest path algorithms from $s_E$ to $t_O$. 