SHORTEST PATH

Today we will discuss this problem: in a directed graph G with V vertices and E non-negative weighted directed edges, what is the length of the shortest path from a given vertex A to another vertex B, where a path length is defined as the sum of all edge weights along that path? (*Note that there is no need to return the path itself, merely the distance)*

Before we go into the algorithms, we will lay down some preliminary points first.

The Shortest Paths Tree

Suppose we draw all the shortest paths from A to every other vertex in G. It is always possible to draw the shortest paths such that the set of edges used forms a directed acyclic graph. The directed acyclic graph is a graph which contains no cycles, looks quite similar to a tree, and indeed behaves the same for our purposes - there is only one path from A to any other vertex in the shortest paths tree.

The exception to being able to draw the shortest paths tree is if there is a negative weight cycle in the graph.

The two graph theory algorithms we will be using today use this concept to construct the shortest path from A *to every other vertex.*

Negative Edge Weights

This doesn't pop up too often, but occasionally a shortest paths problem will have some edges with negative weights. This is not an issue for one of our algorithms unless a **negative weight cycle** appears, because then many (if not all) of the shortest paths become negative infinite because a shortest path can run around the negative-weight cycle to constantly decrease the path length. Thus, the problem becomes meaningless.

The shortest **simple** path, which takes care of this case, is an NP-hard problem and will not be discussed here. Thus, we will assume there are no negative-weight cycles.

Other Points

- Relaxation: If a better distance is found, then the algorithm will replace the current shortest distance with the new shortest distance.
- The shortest path from A to A is always **0**.

Bellman-Ford

Bellman-Ford finds the shortest path from a given vertex to every other vertex in a directed graph in O(VE).

Pseudocode

1. Input your graph. Construct a `dist` array, where `dist[U]` contains the length of the shortest path from A to U.
2. Set `dist[A]` to be **0**. Set every other entry to infinity (you can't do this in C++, so set it to some arbitrarily large number that is also higher than the maximum possible path length).
3. For every edge (I,J) (edge that goes from vertex I to vertex J), `dist[J] = min(dist[J], dist[I] + wt(I,J))` where `wt(I,J)` represents the weight of the edge that goes from I to J.
4. Do step 3 a total of V-1 times.

Why Does This Work?

Suppose at the start of every execution of step 3 we take the set of all vertices that have their shortest distance determined and the set of all vertices that do not. Because the shortest paths tree always exists, at least one element of the set of vertices that do not have their shortest path determined (unless the set is empty) is a direct descendant of one of the vertices in the set of vertices that do have their shortest distance determined. The highest number of step 3 executions that need to be done is V-1, since the longest path in the shortest paths tree can be at most V-1 edges long.

This looks exactly like BFS, except done in a very roundabout way, since we do not know what the shortest paths tree looks like when starting with the graph.

Bellman-Ford does indeed work with negative edges, unlike the next algorithm we will be looking at, and it can also detect the presence of negative weight cycles: simply run step 3 once again and then check if any of the values change. If they do, then there is a negative weight cycle present.

Dijkstra

Dijkstra's shortest path algorithm runs in O((V + E)logV) time complexity. Unlike Bellman-Ford, it will completely break if any of the edge weights are negative.

The concept behind Dijkstra is that it is a greedy algorithm that will build the shortest paths tree one edge at a time, and record down distances as it finds new vertices.

Before diving into the implementation of Dijkstra, we will introduce a new data structure known as the **priority queue**. The priority queue is like the queue, except when an insertion occurs, the value is inserted such that all the values in the priority queue remain in sorted order. The way to declare C++ priority queues is as thus:

```
priority_queue<pair<int, int>,
               vector<pair<int, int>>,
               greater<pair<int, int>>> Q;
```

This declares a priority queue of <int,int> pairs. The underlying structure used to store the priority queue is a vector of pairs, and the function used to sort the priority queue is greater. If no third parameter is specified (no function for sorting), then by default the priority queue will sort in descending order. Priority queues will also be used in MST problems, which will be elaborated on later.

Here is a bit of pseudocode:

1. Set **every** entry in `dist` to infinite.
2. Take the pair (0,A) and push it into your priority queue.
3. Take the smallest pair (d,v) (sorted by first priority) in your priority queue, pop it out, and check if `dist[v]` is infinite. If it is not, repeat step 3.
4. If it is, then set `dist[v] = d`. Loop through all edges (v,u) emanating from v, and push each `{d + wt(v,u), u}` into the priority queue.
5. Go back to step 4 until the priority queue is empty.

After executing Dijkstra's shortest path algorithm, you will end up with the same endpoint as Bellman-Ford: a static array containing the weights of the shortest paths from A to every other vertex. If the shortest path is infinite, then there is no path from A to that particular vertex.

Why Does This Work?

Dijkstra keeps track of all the current paths: their distance, and where they end up. It greedily takes the smallest one and attempts to use it, and if it works then it has just discovered another edge of the shortest paths tree. It keeps doing this until all potentialities have been exhausted, and what comes out at the end is the lengths of the shortest paths to all other vertices.