Dijkstra's Algorithm

Dijkstra's algorithm is a natural, greedy approach towards solving the shortest path problem. The shortest path tree is built up, edge by edge. Given a partially constructed tree (initially just the *source*), let u be the node not currently in the tree such that u can be connected to the tree via an edge (u, v) *and* the path from *source* to u (via v) is cheaper than any other path that could be formed by extending the tree by one edge. Then, u is added to the tree.

The inputs to the algorithm are the initial vertex, a, the final vertex, z, the weight of edge (i, j), $w(i, j) > 0$, and the label of vertex x, $L(x)$. At termination, $L(z)$ is the length of the shortest path from a to z.

```
1  DIJKSTRA($w, a, z, L$)
2
3      $L(a) = 0$
4      for all vertices $x \neq a$ do
5      $L(x) = \infty$
6      $T$ = set of all vertices whose shortest distance
       from $a$ has not been found
7      while $z \in T$ do
8          choose $v \in T$ with minimum $L(v)$
8          $T = T - \{v\}$
9          for each $x \in T$ adjacent to $v$ do
11                 $L(x) = \min\{L(x), L(v) + w(v, x)\}$
```

Just like the *Prim's* algorithm, the running time of this algorithm is $O(m \log n)$, where m is the number of edges, and n is the number of nodes.

⋆ Shortest Path Tree

⋆ MSP

⋆ Shortest Path Tree for directed graph

Correctness of Dijkstra's Algorithm

We now show that the algorithm IS correct when all edge weights are nonnegative. Let $\delta(s, v)$ be the true cost of a shortest path from the source s to node v. Just as in the proof of correctness of Prim's algorithm, correctness follows from the following claim:

Claim: If the partially constructed tree T, constructed after i edges have been added, is a subtree of some shortest path tree, then so is the tree T', constructed after $i + 1$ edges are added.

Proof of claim: Let T_{final} be some shortest path tree that contains T as a subtree. Let (u, v) be the $(i + 1)^{st}$ edge added by the algorithm. If T_{final} contains (u, v), we are done. Otherwise, let P be the path of T_{final} from the source s to v.

⋆

Let T'_{final} be the tree obtained from T_{final} by removing the edge into v of tree T_{final} (in our example this edge is (b, v) and adding the edge (u, v). The subtree of T_{final} rooted at v hangs beneath u in T'_{final}.

We claim that the path from s to v in T'_{final} is no more costly than the path from s to v in T_{final} and so T'_{final} is indeed a shortest path tree of the graph. To see this, let y be the first node on path P that is not in tree T (note that y may

be v, although this is not so in the example above). Let x be the predecessor of y on path P.

Since we are assuming that all the weights are non-negative, and since a shortest path from s to v goes through y, it must be the case that

$$\delta(s, y) \leq \delta(s, v).$$

When x was added to the tree, the algorithm ensured that

$$cost(y) \leq cost(x) + w(x, y) = \delta(s, y).$$

Also, at the point v is added to the tree, it must be that

$$cost(v) \leq cost(y)$$

since the algorithm always adds a node with minimum cost and v is selected for addition before y. Putting all of this together, we have

$$cost(v) \leq cost(y) \leq \delta(s, y) \leq \delta(s, v).$$

Since $parent(v) = u$, there is a shortest path from the source to v that passes through u and so T'_{final} is indeed a shortest path tree of the graph. This completes the proof. ■

Bellman-Ford Algorithm

We next describe the Bellman-Ford algorithm for the single-source shortest paths problem, which works even when the input graph has negative weights. The Bellman-Ford algorithm has running time $O(mn)$.

This algorithm for the single-source shortest paths problem works correctly even in the presence of negative weights, and is a generalization of Dijkstra's algorithm. The algorithm maintains, for each node v, a variable $cost(v)$ which represents our current best guess as to what the shortest path is from the source to v. Initially, $cost(s) = 0$ and $cost(v) = \infty$ for all $v \neq s$ (where s is the source). Just like Dijkstra's algorithm, the new Bellman-Ford algorithm builds a tree of paths by "relaxing" all the edges of the graph in turn. "Relaxing" edge (u, v) means the following: if $cost(u) + w(u, v)$ is less than $cost(v)$, we know that we can improve our path to v by going through u. We do this by setting $cost(v)$ to

be $cost(u) + w(u, v)$ and setting $parent(v)$ to u.

It is necessary that edges be relaxed more than once in order that the end result be a shortest path tree. The Bellman-Ford algorithm simply relaxes all edges (in arbitrary order), then repeats this step, and again, until all edges have been relaxed $n - 1$ times:

```
for i <-- 1 to n-1 do
   for each edge (u,v) do
      if cost(v) > cost(u) + w(u,v) then
         cost(v) <-- cost(u) + w(u,v)
         parent(v) <-- u
```

⋆ Shortest Path Tree

To show that the Bellman-Ford algorithm is correct, we need to argue that after $n-1$ iterations of the inner for loop, the resulting tree is indeed a shortest paths tree. The argument goes as follows: after the first iteration of the outer for loop (when $i = 1$), the shortest path to any node which happens to be one edge away from the source s in the shortest paths tree has been found. The shortest path to those nodes that are two edges away from the source in the shortest paths tree is found after the second iteration of the outer for loop ($i = 2$). And so on. Since all nodes are of distance at most $n-1$ from the source in the shortest paths tree, the shortest path to all nodes is found after $n-1$ iterations of the outer for loop.

Since the outer for loop is executed n times and the inner for loop is executed m times, the total running time is $O(mn)$.

Floyd-Warshall Algorithm

Finally, we describe the Floyd-Warshall algorithm for the all source (all destinations) shortest path problem. This algorithm has running time $O(n^3)$. Note that this is better than simply running the Bellman-Ford algorithm n times, once per source, when the number of edges m grows faster than n.

The idea of this algorithm is to compute, for all pairs (u, v), the shortest path from u to v which passes through only the first i nodes as intermediate nodes, $0 \leq i \leq n$. Let $cost_i(u, v), 0 \leq i \leq n$, denote the cost of this shortest path. Upon completion of the algorithm the cost of the shortest path between every pair of nodes is known. From this information, the actual paths can easily be computed.

Calculating $cost_0(u, v)$ is easy:

⋆

$$cost_0(u,v) = \begin{cases} & \text{if } u = v \\ & \text{if there is an edge from } u \text{ to } v,\ u \neq v \\ & \text{otherwise} \end{cases}$$

We next show that once the costs $cost_{i-1}(u, v)$ have been computed, it is easy to compute $cost_i(u, v)$. There are two possibilities:

⋆

1. The shortest path from u to v visiting only nodes in the set $\{1, \ldots, i\}$ actually only visits the first $i - 1$ nodes. Clearly, in this case

$$cost_i(u, v) =$$

⋆

2. The shortest path from u to v visiting only nodes in the set $\{1, \ldots, i\}$ *does* visit node i. In this case, assuming that the graph has no negative cycles, the shortest path from u to v visits i exactly once. Therefore,

$$cost_i(u, v) =$$

From these two cases, we can see that

$$cost_i(u, v) = \min\{cost_{i-1}(u, v), cost_{i-1}(u, i) + cost_{i-1}(i, v)\}.$$

Using this fact, it becomes clear how to write the Floyd-Warshall algorithm:

```
for i=1 to n
   for each vertex u do
      for each vertex v do
         cost_i(u,v)= min{cost_{i-1}(u,v), cost_{i-1}(u,i) + cost_{i-1}(i,v)}
```

Note that the main part of the algorithm consists of three nested for loops, each iterated n times, with a constant-time piece of code within the three loops. Therefore, the running time of this algorithm is $O(n^3)$.
Upon completion of the algorithm, if the graph has no negative cycles, we have found the costs of the shortest paths between all pairs (u, v). If the graph *does* have a negative cycle, we can detect this as follows:

```
for each vertex u do
    if cost(u,u) < 0 then output ‘‘negative cycle!’’
```

Space use can be large with this algorithm. The way it is written above, we need n (the outside loop) $n \times n$ arrays (for storing the $cost_i(u, v)$'s), so the space required is $\Theta(n^3)$. We can reduce this requirement by realizing that at iteration i of the outer for loop, we only need array $cost_{i-1}$,so we can reduce the space to $\Theta(n^2)$. There is actually a way to get the space down to $O(n)$.

⋆ All-Pairs Shortest Paths