

# CSE 101 Class Notes

## Dynamic Programming: Shortest Paths

June 2, 2004

There are several useful algorithms related to finding shortest paths in a graph. Today, we will look at the problem of finding the shortest path between two nodes.

### Bellman-Ford Algorithm

Our first attempt at a backtracking algorithm is:

```
1   SP(x, y, G)
2     if x = y
3       return 0
4     s <- Inf
5     for z in neighbors(x)
6       s <- min(s, d(x,z) + SP(z,y,G))
7     return s
```

But this quickly runs into trouble, since we don't keep track of where we've already visited. It turns out that our problem is equivalent to a more general one, that of finding the shortest path between one node and all others. Since a shortest path can't contain a cycle, we must have found the shortest after following  $n = |V|$  edges, so to refine our approach, we keep track of how many edges we have crossed so far:

```
1   SP (x, y, t, G)
2     if x = y
3       return 0
4     if t = 0
5       return Inf
6     s <- Inf
7     for z in neighbors(x)
8       s <- min(s, d(x,z) + SP(z, y, t-1, G))
9     return s
```

where  $t$  starts at  $n$ .

### Dynamic programming version

Subproblems can be identified by a pair of starting vertex ( $x$  or  $z$ ) and path length  $t$ . We have two base cases (lines 2-3 and 4-5 above): First, the distance from  $y$  to itself is 0. Second, the distance from anything to  $y$  in more than  $t$  steps is  $\text{Inf}$ . This leads to the following dynamic programming algorithm:

```
1  SP(x, y, G)
2  N <- size(V)
3  D[v in V, 0 .. N] : Array of distances
4  for t = 0 .. N
5  D[y,t] <- 0
6  for z in V - {y}
7  D[z,0] <- Inf
8  for t = 1 .. N
9  for v in V - {y}
10 D[t,w] <- Inf
11 for w in neighbors(v)
12 D[t,w] <- min(D[t,w], d(w,y) + D[t-1, w])
13 return D[N,x]
```

### Example

Here is how this algorithm works on our example above, when starting from New York:

Time	SF	LA	SD	LV	D	NO	Ch	Atl	Ph	NY
1	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	0
2	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	13	8	$\infty$	0
3	$\infty$	$\infty$	$\infty$	$\infty$	19	13	13	8	$\infty$	0
4	27	$\infty$	$\infty$	24	19	13	13	8	17	0
5	27	28	19	20	19	13	13	8	17	0
6	...	...	...	...	...	...	...	...	...	...

### Running time

There are three loops (lines 8, 9, 11), each of which can run no more than  $|V|$  times, so the algorithm is at least  $O(n^3)$ . However, we also know that the loop on line 11 only executes twice for each edge, or  $O(m)$  times, so we can provide a tighter bound of  $O(n^2 + nm) = O(nm)$ .

### Floyd-Warshall

Another approach is to branch based on whether or not a node occurs along the shortest path. In other words:

```

1   FW(x, y, V[1..n])
2     if n == 0
3       return d(x, y) // Inf, if not connected
4     else
5       return min(
6         FW(V[n], y, V[1..n-1]) + FW(x, V[n], V[1..n-1]),
7         FW(x, y, V[1..n-1]))

```

The dynamic programming version of this is:

```

1   FW(x, y, G = (V,E))
2     n <- size(V)
3     SP[u in V, v in V, 1..n] : Array of distances
4     SP[u,v,0] <- d(u,v)
5     for i = 1 .. n
6       for u in V
7         for v in V
8           SP[u, v, i] <- min(
9             SP[u, V[i], i-1] + SP[V[i], v, i-1],
10            SP[u, v, i-1])
11    return SP[x,y,n]

```

Note that here, although the algorithm is always  $O(n^3)$  because of the 3 loops on lines 5-7, we are actually computing the shortest paths between *all* pairs of nodes.