Henry Daniels-Koch
December 9 2016

Algorithms: Lab 10

1). To compute the most reliable path between two vertices, we use a similar algorithm to Dijkstra's algorithm for finding the shortest path. However, instead of adding the weight of of path to the current path length, we multiply the probability that the channel will not fail by the total current channel's probability that it will not fail, d[v]. We first set the source node's probability to 1 and the probabilities of all other nodes to 0. As with Dijkstra's algorithm, we iterate until a queue is empty. However, we delete the maximum value node in the queue instead of the minimum. For each edge touching the current edge, we relax that edge. This relaxation is done by comparing the probability of computed path so far at the previous node by the probability of the path. Differing from Dijkstra, we select the path if it has a HIGHER probability instead of a shorted path length. We then pick the edge with the maximum probability. This algorithm runs in $0(|E|log|V|)$ since it runs similarly to Dijkstra's algorithm The pseudocode is as follows:

```
maxProbPath(source vertex s, end vertex endV)
    //initialize
    d[s]=1; FOReachvV,v=s: d[v]=0
    FOR each v  V : Insert(Q, v, d[v])
    //do one round of edge relaxation in greedy order of d[] S= not empty
    WHILE Q not empty DO
        u = Deletemax(Q)
        S = S  u
        FOR each e = (u, v) in E with v in V  S DO
            IF d[v] < d[u] * r(u, v) THEN
                d[v] = d[u] * r(u, v)
                Change(Q, v, d[v])
                pred[v] = u
    return d[endV]
```

2). To calculate the maximum bandwidth path, we need to calculate the minimum bandwidth along each path. To do this, we perform a modified Dijkstra algorithm on a node a. We initialize the first node's bandwidth as ∞ and every other node's bandwidth to 0. Until the queue is empty, we delete the maximum bandwidth node in the queue since we want path's with high bandwidths. For each connected edge, we then relax it by comparing the minimum bandwidth of the total path computed so far to the bandwidth of the current path from u to v. If this value is greater than the bandwidth currently at node v, then we store this bandwidth in v and change the queue accordingly. At the end, we return the bandwidth of node b, which is automatically that maximum bandwidth path from a to b. This algorithm is so similar to Dijkstra's algorithm and does not have any additional calls. Therefore it runs in $O(|E|log|V|)$. The pseudocode is as follows: (took the base code from Professor Toma's website and modified it***).

```
maxBandwidthPath(source vertex a, end vertex b)
    //initialize
    d[a]=1; FOReachvV,v=a: d[v]=0
    FOR each v  V : Insert(Q, v, d[v])
    //do one round of edge relaxation in greedy order of d[] S= not empty
    WHILE Q not empty DO
        u = Deletemax(Q)
        S = S  u
        FOR each e = (u, v) in E with v in V  S DO
            IF d[v] < min(d[u] ,bandwidth(u, v)) THEN
                d[v] = min(d[u],bandwidth(u, v))
                Change(Q, v, d[v])
```

```
            pred[v] = u
    return d[b]
```

3a). For a large enough graph in a rectangular grid that has a large internal area compared to edge nodes (more square than thin rectangle), there are roughly 3 edges for each node. Therefore, $E \sim 3V$. Since Dijkstra runs in $O(|E|log|V|)$, with $E = 3V$, Dijkstra will run in $O(|3V|log|V|) = O(|V|log|V|)$.

b). In order to find the length of the shortest paths from the root to all other vertices, we can topologically sort the graph. We begin by initializing the first node to 0 and all other nodes to ∞. Then, for each node in topological order, we relax all outgoing edges. This works because a directed, acyclic graph always contains a node of indegree 0. This is easily observed from the graph in the figure. If we relax all outgoing edges, we will always have one node free. Since topological sort runs in $O(|E| + |V|)$ and $E \sim 3V$, then this algorithm runs in $O(|3V| + |V|)$=$O(V)$. The pseudocode is as follows:

```
shortestPath(source c)      for all v: d[v] = ∞
d[s] = 0
for all u in topological order
    for all outgoing edges u to v
        if d[v] > d[u] + w(u,v)
            d[v] = d[u] + w(u,v)
            pred[v]=u
```

c). To solve the all pair shortest paths problem, we run the algorithm above on every node. While this takes $O(|V|^2)$, there is no faster algorithm in the worst case scenario. We note that all nodes are reset by the above algorithm to essentially get a clean start for each source node. The pseudocode is as follows:

```
APSP(graph G)
    for each node v in G: shortestPath(v)
```