Homework Set 3

Miles Van de Wetering, Charles Hill, Cierra Shawe

February 14, 2017

Problem 1 - a

How could we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph G' = (V', E') with weighted edges, where V' = V \{v}, and the shortest-path distance between any two nodes in G' is equal to the shortest-path distance between the same two nodes in G, in O(V^2) time.

To solve this, we simply choose any vertex to remove. We prevent any shortest paths that make use of this vertex from changing by replacing paths that are broken by the removal of v with new edges. If v has no outgoing edges, it is at the end of every path it's a part of (i.e., v is a *sink*). Therefore, it can be deleted without affecting any other pair of vertices. v has no incoming edges, it is the beginning of every path it's a part of (i.e., v is a *source*). Therefore, it can be deleted without affecting any other pair of vertices. If, however, v has *at least* one incoming *and* outgoing edge, then we must replace the paths that are impacted by the deletion of v. The algorithm for this is as follows:

First, for each vertex v, examine the set of outgoing edges from v. While there exists some edge pairing $e_1 = (v, u), e_2 = (v, u)$ mark the edge with the larger weight as 'hidden.' We will use this later. Alternatively, if the graph is sparse then we may skip this step. Once this is complete, we may assert (and rely upon the fact) that *at most one unhidden edge* connects any vertex v to any other vertex u. Up to one edge may *additonally* connect u back to v. (this may be done in O(E), and needn't be done if the graph is sparse)

Next, for each pair of edges $e_1 = (u_1, v), e_2 = (v, u_2), v, u_1, u_2 \in V, e_1, e_2 \in E$ (note that no such pairing of edges exists if v is a *sink* or *source*) examine the unhidden edge connecting u_1 and u_2 (if one exists) $e_3 = (u_1, u_2)$. If $w(e_1) + w(e_2) < w(e_3)$ then mark e_3 as hidden and add a new, unhidden edge $e_4 = (u_1, u_2)$ with weight $w(e_4) = w(e_1) + w(e_2)$. If there is no edge $e_3 = (u_1, u_2)$ then add a new edge in the same manner. Then, delete v.

In this way, we preserve all shortest paths that made use of v.

The first portion of this algorithm runs in O(E), while the second portion runs in O(V^2) because there may be up to |V| incoming edges to v, and there may be up to v outgoing edges, and we need to account for every combination of in->out paths, which is V*(V-1) (since we exclude the possibility of self loops).

Problem 1 - b

Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances from v to every other vertex, and from every other vertex to v, in the original graph G, in O(V^2) time.

We break this problem into two parts. First, we will calculate the shortest path from v to all us in V'. Then, we will compute the shortest path from all us in V' to v.

To compute the shortest path from v to all us in V', we first observe that the shortest path between any $u_i \to u_j$ hasn't changed, and *will not change* with the addition of v back into the graph.

```
Now, for each outgoing edge $e = (v, u_i)$:
    First, initialize $dist(v \to u_i) \leftarrow \infty$
    For each vertex $u_j \in V' \backslash u_i$
        Check the shortest path length $p = dist(u_i \to u_j)$. If $p + w(e) < dist(v \to u_j)$ then $dist(v \to u_j) \leftarrow p + w(e)$
```

```
Secondly, for each vertex $u_i \in V'$
    First, initialize $dist(u_i \rightarrow v) \leftarrow \infty$
    For each incoming edge $e = (u_j, v)$
        Check the shortest path $p = dist(u_i \rightarrow u_j)$. If $p + w(e) < dist(u_i \rightarrow v)$ then $dist(u_i \rightarrow v) \leftarrow p + w(e)$.
```

This runs in O(V^2) time, since the first step checks each outgoing edge (of which there may be up to V), and for each of these edges checks the distance between a node and every other node, which of course is V calculations. Hence, V^2. The second portion is V^2 for very similar reasons - we check each incoming edge and for each node we check the distance between that node and each other node.

Problem 1 - c

To calculate all pairs' shortest paths from the above two algorithms, only simple modifications are required.

Here we define a recursive version of our algorithm (*easily* adaptable to be iterative). A(V) represents running part a on the set of vertices V, and B(V, v) represents adding vertex v back into the set V:

```
Recursive:
    C(V) =>
            A(V, v) (choose any vertex from V to use as v)
            C(V\v)
            B(V, v)
```

The run time of C(V) is O(V^3) since A runs in O(V^2) and B also runs in O(V^2), and C(V) runs both A and B V times. So, V^3.