JONATHAN PETERSEN
A01236750
CS 5050 - ADVANCED ALGORITHMS

Homework 7

Graph Algorithms II

1. Create a k-link shortest path algorithm for DAG's that runs in O(k(m +n)) time. Only return the length of the path. (Hint: Use Dynamic Programming or modify the shortest path algorithm discussed in class.)

Given a DAG G with m edges and n vertices implemented as an adjacency list A, and a set of weights w where w(n1, n2) is the weight of the edge from n1 to n2, create an additional parameter hops where hops(n1, n2) is the shortest number of edges to travel from the starting node s to n2 through n1.

Then, simply visit each vertex v in topological order, and for each vertex u in the adjacency list of the current vertex, check if the distance to u through v is smaller than the previous smallest found distance and if u can be reached in less than k hops from s through v. If so, change the current smallest distance to u to be equal to the distance to v plus the weight of the smallest edge from v to u. Pseudocode for this solution is as follows:

```
int k_shortest_path(Graph G, Node s, int k) {

      // Assume G has an adjacency list A
      // Assume G has weights weight[m][n]
      hops[m][n] = 0; // Stored in a similar structure to A.

      for each vertex v,
            v.dist = INF;

      s.dist = 0;

      topological_sort(G);

      for each vertex v,
            for each vertex u in A[v],
                  if (u.dist > v.dist + weight[u][v] && hops[u][v] < k)
                        u.dist = v.dist + weight[u][v];
                        hops[u][v] += 1;

      return t.dist;
}
```

This solution takes O(m) time to construct an empty structure with the same layout as A, n time to set all the distances to infinity, O(m + n) time to sort the graph topologically, and finally, since both looking up the hops value and incrementing it are O(1), O(m + n) time to compute the smallest k-length path for all of the nodes. Therefore, this algorithm runs in O(m + n) time.

2. Modify Dijkstra's algorithm to compute an optimal path on a general weighted graph. Your algorithm should be able to output the actual optimal path, and should run with the same time complexity as Dijkstra's algorithm.

This time we will construct an array of size n called hops such that hops[n] is the smallest number of hops found along a shortest path to that node. We then implement Dijkstra's algorithm, only modifying the section where we compare distance values and update predecessors. In our modified algorithm, if we find a path to v from u that is shorter than the shortest known path, we set the new shortest distance and predecessors as before, but also set the shortest-path number of hops to v as the number of hops to u plus one and move on to the next edge. Additionally, if the distance to v is equal to the shortest known distance, but the number of hops is smaller, we replace the predecessor value with the current node and update the hops value as normal. Once we have completed the algorithm, we can simply backtrack from the ending node t though the predecessor values until we reach s, adding the nodes we visit to the front of the list that forms our shortest path. Pseudocode is as follows:

```
dijkstra_optimal(G, s) {
        hops[n] = INF;                                                  //O(n)
        for each vertex v {
                v.dist = INF;
                v.pre = NULL;
        }
        s.dist = 0;
        PriorityQueue Q(V);     // The key of Q is v.dist. Implemented with Fibonacci heap.

        while (!Q.empty()) {
                u = extract_min(Q);                                     //O(n log(n))
                for each vertex v in Adj[u]{
                        if (v.dist > u.dist + w(u, v)) {
                                v.dist = u.dist + w(u, v);
                                decrease_key(Q, v, v.dist);             // O(m)
                                hops[v] = hops[u] + 1;                  // O(1)
                                v.pre = u;
                        }
                        else if (v.dist = u.dist + w(u, v) && hops[u] + 1 < hops[v]) { //O(1)
                                v.pre = u;
                                hops[v] = hops[u] + 1;                  //O(1)
                        }
                }
        }

        // Get shortest path
        vertex r = t;
        vertexStack R;
        while (r.pre != NULL) {
                R.push(r);
                r = r.pre;
        }
        return R;
}
```

Our additional operations only take O(n) time, so the whole algorithm is still O(m + n log(n)).

3. Given an undirected weighted graph G and a minimum spanning tree T of G, answer the following questions:

   3.1. Suppose the weight of every edge of G is increased by a specific positive value. Will a shortest path on G remain a shortest path?

   Yes. A shortest path is only concerned with finding the smallest relative distance between two nodes, and the specific numeric value of the edge is only useful in helping us determine an ordering on the edges to find the "shortest." So long as every node is increased by the same amount, there is no way for a longer path to suddenly become a shorter one. Therefore, the shortest path will remain a valid shortest path on G.

   3.2. Suppose the weight of every edge of G is increased by a specific positive value. Is T still a minimum spanning tree of G?

   Yes. Similar to the logic above, a MST is only concerned with relative smallness of the edges. Since increasing each edge by the same amount cannot make a larger edge smaller than another edge originally smaller, the MST of the graph still must traverse the same edges, and is therefore still valid.

4. Given an undirected connected graph with two types of edges (blue and red), design an algorithm to find a spanning tree T using as few red edges as possible. Your algorithm should run in $O((n + m) * \log n)$ time.

We will use a modified version of Prim's algorithm to solve this problem. The basic idea is that we will construct a spanning tree following only blue edges, and then we will visit any unvisited nodes along the remaining red edges. Prim's algorithm runs in $O(m + n \log(n))$ time when using a Fibonacci heap, and our modifications do not alter this value.

Pseudocode is on the following page:

```
modified_prim(G, s) {
      for each vertex v {
            v.key = INF;
            v.pre = NULL;
            v.flag = 0;
      }

      // Traverse all of the Blue edges, forming a spanning tree
      s.key = -INF;
      PriorityQueue Q1(V); // Implemented with a Fibonacci Heap
      Vertex[] hasRed = [];
      while (!Q1.empty()) {
            u = extract-min(Q1);
            u.flag = 1;

            int visited = 0;
            for each vertex v in Adj[u] {
                  if (color(u, v) == BLUE) {
                        visited++;
                        if (v.flag = 0 && v.key > w(u, v)) {
                              v.key = w(u, v);
                              decrease-key(Q1, v, v.key);
                              v.pre = u;
                        }
                  }
            }
            if (visited != Adj[u].size()) {
                  hasRed.push(u);
            }
      }

      // Traverse red edges to any unvisited nodes.
      PriorityQueue Q2(hasRed);
      while (!Q2.empty()) {
            u = extract-min(Q2);
            u.flag = 1;

            for each vertex v in Adj[u] {
                  if (color(u, v) == RED) {
                        if (v.flag = 0 && v.key > w(u, v)) {
                              v.key = w(u, v);
                              decrease-key(Q2, v, v.key);
                              v.pre = u;
                        }
                  }
            }
      }
}
```