Brigham Michaelis
November 30th 2017
A00364835

Homework 7

Problem 1.

In class we studied algorithms for computing shortest paths. Sometimes people are interesting in finding not only an arbitrary shortest path but also a shortest path with certain properties. Let's consider again the flight example given in Question 4 of the last assignment. Assume that saving money is very important to you (you need to pay tuition, you have kids to take care of, etc.), and so you want to find a shortest (i.e., cheapest) path regardless of how many edges it has. Even in this case, there is still something you can do to "optimize" your life. Indeed, it may be possible that there are multiple shortest paths from s to t (i.e., theses paths have different edges, but they have the same length). **In this case, instead of returning an arbitrary shortest path, it would certainly be better to return a shortest path with as few edges as possible.** This is the problem we are considering in this exercise. But now we are considering a general graph not necessarily a DAG. The problem is formally defined as follows.

Given a directed graph G = (V, E), let n = |V | and m = |E|. Every edge (u, v) of G has a nonnegative weight w(u, v) ≥ 0. Let s and t be two vertices in G. The length of a path in G is the sum of the weights of all edges in the path. A shortest path from s to t is a path with the minimum length among all paths in G from s to t. In class, we studied Dijkstra's algorithm for computing such a shortest path.

It is possible that there are multiple different shortest paths from s to t in G (i.e., these paths may have different edges but their lengths are the same). **In some applications, we may want to find a shortest path from s to t that has edges as few as possible.**

A path π in G from s to t is said to be optimal if:

(1) π is a shortest path from s to t, and

(2) among all shortest paths from s to t, π has the minimum number of edges.

Refer to Figure 1 for an example.

Modify Dijkstra's algorithm to compute an optimal path from s to t. Your algorithm should be able to output the actual optimal path, but for simplicity, you only need to maintain the correct predecessor information v.pre for each vertex v of the graph. Your algorithm should have the same time complexity as Dijkstra's algorithm.(20 points)

Description

This can be done by modifying Dijkstra's algorithm to have each node keep track of its edge counts(The number of edges needed to reach that node), and then to preform some condition checks to see if some value should be overwritten. We show the code below in the Psuedo Code section.

Psuedo Code

```
Dijkstra (G,s)
{
for each vertex v // Every node needs to keep track of: the weight to get to that node, how many
                  // edges need to be traversed to get to that node, and the predecessor node.
        v.d = ∞;
        v.edge_count = 0;
        v.pre = NULL;
s.d = 0; // We set s.d to 0 and its edge count to 0
s.edge_count = 0;
Q; // This is a min heap that is now loaded with all the vertices with v.d used as the keys.
while(Q !empty)
{
        u = extract_min(Q);
        for each v ∈ adj(u)
                // This is key change to Dijkstra's algorithm. Not only are we keeping track of the
                // edge counts (see below), but we are also checking if v.d == u.d + w(w,v). If this
                // condition is true and if v's current edge count is greater than u's edge count + 1
                // then we change v.pre and v.edge_count to reflect the change. Note there is no need
                // to perform a decrease key operation as the key remains unchanged. We are only
                // selecting the "optimal" path.
                if(v.d == u.d + w(u,v))
                        if(v.edge_count > u.edge_count+ 1)
                                v.pre = u;
                                v.edge_count = u.edge_count + 1;
                if(v.d > u.d + w(u,v))
                        v.d = u.d + w(u,v);
                        decrease_key(Q,v,v.d);
                        v.pre = u;
                        v.edge_count = u.edge_count + 1;

}
}
```

Correctness

This algorithm works because of two changes to Dijkstra's algorithm. They are:
1. Each node is going to keep track of it's edge count. This is done by just counting the edges that are traversed to get to a node and incrementing appropriately.
2. We need to perform a check when [v.d == u.d + w(u,v)]. If this condition is true than we need to compare the edge counts and update with respect to the smaller of the two edgecounts.

Runtime

This algorithm runs in Dijkstra time O((m+n)(logn)). The change that was made adds constant work to the problem. So the overall runtime remains unchanged.

Problem 2.
Let G = (V, E) be an undirected connected graph, and each edge (u, v) has a positive weight w(u, v) > 0. Let s and t be two vertices of G. Let π(s, t) denote a shortest path from s to t in G. Let T be a minimum spanning tree of G.
Please answer the following questions and explain why you obtain your answers.

This should be done with variables as the increment

(a) Suppose we increase the weight of every edge of G by a positive value δ > 0. Then, is π(s, t) still a shortest path from s to t? (10 points)
Description
Not necessarily. Consider the graph given below. It is obvious that the shortest path from s to t is: s → p → t. With a path weight of 2. But if we increment all the edges by some positive value, say 5; Now the shortest path is s → t , with a path weight of 10. versus a path weight of 12. **Thus we can see that the answer is "not necessarily".**

(b) Suppose we increase the weight of every edge of G by a positive value δ > 0. Then, is T still a minimum spanning tree of G? (10 points)
Description
The answer is yes. Consider how Prim's algorithm works. Prim's algorithm will always give us the minimum spanning tree of G. Now if we increase every edge by some positive value, then Prim's algrorithm will continue to work properly, and will produce the same tree as before. Because Prim's algorithm is only dependent on comparing multiple possibilities and picking the minimum. Thus if we increment all the edges by some value, their relative comparative results will not change.

Note: For each of the two questions, your answer should be either "Yes" or "Not necessary". Again, please explain why you obtain your answers.

Problem 3.
Let G = (V, E) be an undirected connected graph of n vertices and m edges. Suppose each edge of G has a color of either blue or red. Design an algorithm to find a spanning tree T of G such that T has as few red edges as possible. Your algorithm should run in O((n + m) log n) time. (20 points)

Description
For this problem we are going to modify Prim's algorithm. We will show the modifications that are necessary in the Psuedo Code.
Assumptions:
1. The edges no longer have weights but colors
2. These edge colors are predetermined and are accessed by something like w(u,v).

Psuedo Code
PrimsMod(G)

```
{
// We have a definition that ∞ > red > blue. This definition is key. It is what makes the modified
// algorithm work.
for each vertex v
        v.key = ∞; // Remember reds are greater than blues.
        v.pre = null;
        v.flag = false; // This indicates whether the node is included in the spanning tree T
// pick an arbitrary vertex s
s.key = blue; // we set s to the minimum i.e. blue.
// We are going to use a priority queue to maintain all vertices v with v.key as the key of Q
// And the queue is a min heap queue where blue values are treated as smaller than red values and
// red values are smaller than ∞.
Q; // this is our min heap that sorts by colors…. ∞ > Red > Blue
while(q !empty)
{
        u=extract_min(Q); // This will pull blue edges if possible, otherwise it will pull reds.
        u.flag = true;
        for each vertex v ϵ adj(u)
                if (!v.flag && (v.key > w(u,v)))
                        v.key = w(u,v);
                        v.pre = u;
                        decrease_key(Q,v,v.key);
}
}
```

Correctness
The only changes that were necessary were introducing red and blue values.
That follow the rule: ∞ > red > blue, and setting s.key to blue. And with that we are done. This will use Prim's algorithm to find the minimum spanning tree where the fewest red edges are used. This happens because of the predefined relationship of ∞ > red > blue.

Runtime
This algorithm runs in O((m + n)(log(n))). We are doing a modification to Prim's algorithm and we are adding constant time functions so the overall time remains unchanged.