CS 374 **Spring 2015**

Homework 5

Nikhil Unni (nunni2)

1. (a)

```
Q ← makePQ();
insert(Q, (s,0));
prev(s) ← ∅;
foreach node u ≠ s do
    insert(Q, (u, L‖V‖));
    prev(u) ← ∅;
end
S ← ∅;
for i ← 1 to ‖V‖ do
    (v, dist(s,v)) ← extractMin(Q) S = S ∪ {v};
    foreach u in Adj(v) do
        if dist(s,v) + l(v,u) < dist(s,u) then
            decreaseKey(Q, u, dist(s,u), dist(s,v) + l(v,u));
            prev(u) ← v;
        end
    end
end
```

Algorithm 1: Bounded-Edge Dijkstra's Algorithm

```
A ← zero-index-based array of size L‖V‖ + 1;
GLOBALMIN ← 0
```

Algorithm 2: makePQ()

```
A[val] ← A[val] ∪ (v,val);
```

Algorithm 3: insert(Q, (v,val))

```
A[oldVal] ← A[oldVal] - v;
insert(Q, (v, newVal));
```

Algorithm 4: decreaseKey(Q, v, oldVal, newVal)

```
for i ← GLOBALMIN to L‖V‖ do
    if A[i] ≠ ∅ then
        GLOBALMIN ← i;
        return any element in A[i] and remove from A[i];
    end
end
return ∅;
```

Algorithm 5: extractMin(Q)

The algorithm literally copies the original Dikstra's algorithm from the lectures (while changing a few of the arguments so that writing out functions is nicer), with my own priority queue implementation. The entire queue is just a bucket sort like implementation, taking advantage of the fact that the edge lengths in Dikstra's algorithms are always going to be non-strictly increasing, so we can keep track of a global minimum when searching the buckets, instead of starting from 0 every time. This makes the amortized time of extractMin $O(L\|V\|)$ overall throughout the entire function. Besides this, makePQ, insert, and decreaseKey are all $O(1)$ operations since we know of the indices we're dealing with for the array.

I can prove the correctness of the extractMin function as proof of correctness, as most of the algorithm is the same. Trivially, iterating until the first non-null entry in the array to find the

minimum can be proven by contradiction. Assume that the first instance found is not the smallest. This means for some index $p < i$, there is an entry. But the first non-null entry was at i, so there is a contradiction, and thus the first non-null bucket contains a minimum.

Because we are always picking the correct minimum value at any point, the values of the points we pick have to be greater than or equal to the last value, meaning that the GLOBALMIN is not missing any values.

This makes the algorithm overall $O(nL + m)$.

(b) Using the exact same Dikstra's algorithm as above, only now changing the queue once again, if we use a min-heap of size L, where each element in the heap represents a vertex to be pulled. Since we know that the range of values are bounded by the last distance to the origin as well as the last distance + L, by the pidgeonhole principle there must be some repeats. However, it is all in the range of L unique values, meaning we can use a heap of maximum size L for our priority queue.

This makes all queue operations $O(logL)$, as discussed in lecture. Finally, this will make the overall algorithm $O((n + m)logL)$ time.