Matthew Feng October 5, 2018

6.046 Problem Set 4

Collaborators: *Eric Qian, Tyler Barr, Alex Guo*

Problem 1

(a)

Since we must first prioritize Jane's commute, we first compute the shortest path from Jane's office o to her house h. We can do this $O(E + V \log V)$ time using Dijkstra's algorithm with Fibonacci heaps. After we have found this path, we create a supernode z, and connect z to every node in that path with an edge of weight 0, and then delete the edges in the original graph. We can then run Prim's algorithm starting with node z to find the "minimum spanning tree" that also includes the shortest route from o to h (in reality, we are finding the minimum spanning trees for each of nodes in the shortest path and then connecting them via the edges in the shortest path). Finally, we remove the 0 weight edges and add back the original shortest path edges and return that set of edges. The algorithm will find the minimum cost edges to form the minimum spanning tree because after we remove the shortest path edges from the graph and then run the MST algorithm, we will find a true MST for the graph. By re-adding the shortest path edges, we ensure that Jane has her shortest path, and the rest of the MST has minimum cost paths given the requirement that Jane's shortest path must be in the graph. Prim's algorithm also runs in $O(E + V \log V)$ time with a Fibonacci heap, so the overall algorithm will run in $O(E + V \log V)$ time.

(b)

We can build off of our start from the previous part and only consider the edges in our pseudo-MST as well as the k edges with reduced cost. We want to reuse as much of our previous MST as possible, which consists of pieces that we are sure will be part of our new MST T''. We first break up T' along any of k edges that have reduced values. We know that each of the edges in the disjoint sets that we have now separated T' into (including the disjoint set containing shortest path nodes, because we include a supernode that connects all of those with edges of weight 0, so that they will definitely be part of the MST) will be part of a MST, because of the **optimal substructure property**. After we sort the k edges in $O(k \log k)$ time, we can use Kruskal's algorithm to reconnect the sub-MSTs considering only the k reduced cost edges, because of the **cut property** — the light edges that cross the

various cuts consisting of $(S, V - S)$ where S is a disjoint set created from when we broke up our original MST must be among the k edges, else they would not have been broken (or would have been part of the original MST). Since we only look at $k = O(V)$ edges when running Kruskal's algorithm, Kruskal's algorithm runs in $O(V \log V)$ time, and thus the total running time of our algorithm is $O(V \log V)$.

Problem 2

(a)

We construct the digraph $G = (V, E)$ with capacity function $c(u, v) : E \to \mathbb{R}$ denoting the maximum flow that can pass through edge (u, v) at any moment in time.

This is a variation on the bipartite matching problem. As such, we have m vertices $r_j \in V, \forall j \in [1, m]$ that will represent riders waiting to be matched, and n vertices $k_i \in V, \forall i \in [1, i]$ that represent cars that can hold riders. We have a source s and sink t as well, which will allow us to think of the flow in this network intuitively as riders getting (matched) into cars.

We then will construct m edges (s, r_j), $\forall j \in [1, m]$, where $c(s, r_j) = 1$, as each rider only needs one car.

We will also construct n edges (k_i, t), $\forall i \in [1, n]$, where $c(k_i, t) = s_i$, the seat capacity of each car. By setting the capacities in this way, we can sure that each car is matched with at most its seat capacity.

Finally, for each rider j, we construct edge (r_j, k), $\forall k \in C_j$, each with infinite capacity.

(b)

We make $H + 1$ copies of the graph G, indexed as $G^i = (V^i, E^i)$, where the superscript represents the number of hours that have elapsed (thus $i \in [0, H]$). For every airport $v_n^i \in V^i$, we construct edges $(v_n^i, v_m^{i+h_j})$ with capacity c_j for all flights e_j starting from airport v_n and landing in airport v_m. We also construct edges (v_n^i, v_n^{i+1}) with infinite capacity to represent passengers who wait at the airport for their flight. Then, we connect all Boston airport destination nodes t^i to a true sink node t^*, where each of those connecting edges has infinite capacity as well. Finally, we connect a source node s to v_n^0 for every $v_n \in V$ with edge (s, v_n^0) with capacity c_n, representing the number of people starting off from that airport. If we run max flow on this graph, we will be able to find the maximum of passengers that Ryde can

bring to Boston, because we have encoded every possible travel path in our flow network.

(c)

Since our graph will have $HV + 2$ nodes (H copies of V nodes, as well as s and $t*$), and $HE + V + H$ edges (H copies of E edges, with V edges connecting s to all vertices in V^0, and H edges connecting t^i to t^*), for a total runtime of $O((HV + 2)(HE + V + H)^2)$, or $O((HV)(HE + V + H)^2)$.

(d)

We can run DFS on the residual graph G_f twice. We first run DFS once from the source s, keeping the directions in G_f intact, so that we can find all the nodes that are part of the connected component that s belongs to. We then flip the directions of the edges in the residual graph, and run DFS from t, so that we can find all the nodes in the connected component containing t. We find the connected components for s and t so that we can find the augmenting paths from s to t that are missing a single edge. We know that s and t must be in separate connected components, else the flow that we had found would not have been a max flow (because there would have been an augmenting path). For every node u that is in s's connected component, we want to see if we can add a flight to some node v in t's connected component — if so, then that is a valid flight to add. DFS runs in $O(H(V + E))$ time. Since each connected component can have at most $O(HV)$ nodes, and we want to compare all possible pairs, finding all possible edges that might have a valid flight takes $O(H^2V^2)$ time, so the total runtime of the algorithm is $O((HV)^2)$.