CSE 101: Practice Problem Solutions

1 Problems

1. 8.1
   To solve TSP-OPT we will make a polynomial number of calls to TSP, varying our input b using binary search. Let T be the sum of all the edge weights in G. We know that the minimum tour is at most T, but at least 0. We can use these two bounds to run a binary search routine. We first query TSP with $b = T/2$. If it answers NO, then we know that the min tour must be between T and $T/2$. If it answers YES, then we know that the min tour must be between 0 and $T/2$. We can continue the binary search in this recursive fashion until we've reached the resolution of the min-edge length in the graph, and we would have only made $\log T$ calls to the decision version. We know that $\log T$ must be polynomial in the input size because of the following argument. Let K be the largest edge weight in G. If we replace all the edge weights with K then we will need at most $|E| \log K$ bits to represent T, which is still polynomial in the input size. So, we will only make a polynomial (in the input size) number of calls to TSP in our binary search routine that solves TSP-OPT. This shows that TSP-OPT is what is called "NP-hard."

2. 8.2
   Let G be the instance on which we want to return the path, and let A_D be the algorithm that decides whether G has a Rudrata path. We've already run A_D on G and it returned YES, indicating that a Rudrata path exists, but now we'd like to actually find out the edges involved in the path. Imagine removing an edge e from G and calling A_D. If it says YES, then we know that a Rudrata path still exists in this smaller graph, and therefore must exist in the original. If it says NO, then we know that e was a part of the Rudrata path. We can put this logic in a loop, repeatedly removing edges until only the Rudrata path itself is left, and then we halt.

3. 8.12

   (a) We must show that given a solution S to the k-Spanning Tree problem, that we can check in polynomial time whether it is in fact a

k-Spanning Tree. This amounts to verifying that every node in the original G is used in S, that S has no cycles (i.e. it is a tree), and that every node in the tree has degree at most k. All of these can be checked efficiently, and therefore k-Spanning Tree is a search problem.

(b) We've already shown that k-Spanning Tree is in NP in part (a). We now must show that we can solve some already known NP-complete problem given an efficient solution to k-Spanning Tree. We will show that we can solve the Rudrata Path problem given an efficient algorithm, A_{ST}, for k-Spanning Tree. Let G be an unweighted undirected graph for which we'd like to decide whether or not a Rudrata Path exists in. Imagine running A_{ST} with k=2 on G where we add weights equal to 1 on every edge. Notice that at tree that has each vertex with degree at most 2 is a path. If A_{ST} says YES, then that means there exists a path that spans all the vertices in G (i.e. a Rudrata Path). If it says NO TREE EXISTS, then there is no path without loops that touches all the vertices (i.e. no Rudrata path). We've therefore shown that Rudrata Path can be solved using k-Spanning Tree. In other words, we've reduced Rudrata Path to k-Spanning Tree, which along with the fact that k-Spanning Tree is in NP, shows that it is NP-complete.

4. 8.14

First let's show that this problem is in NP. Given a solution S that claims to be a clique of size k in G and an independent set of size k, we can easily check both these claims in polynomial time by examining G. This shows that it is in NP.

Now let's show that we could solve Independent Set if we could solve this problem. In Independent Set, we are given a graph G and a number k and we are asked to decide whether an independent set of size k exists in G. Let A be the algorithm the solves the clique PLUS independent set of size k problem. We are given an input G to the independent set problem. Augment G with k new vertices that are connected to each other in a clique and are connected to no other vertices in G. This new graph trivially has a clique of size k now. Now run A on this augmented graph. If it answers YES, then that must mean that G had an independent set of size k. If it answers NO, then there cannot be an independent set of size k. This shows that this problem is NP-complete.

5. 9.7

(a) When $k = 2$, this problem becomes exactly the (s,t)-minimum cut problem, which we know can be solved efficiently using a variety of max-flow algorithms.

(b) Imagine trying to find the minimum cut that separates s_1, from the rest of the terminals. We could create a dummy vertex t and connect

s_2, s_3 to t with edge weights that are infinite. Letting $s = s_1$, we could then find the minimum (s,t)-cut in this graph, which would be the minimum cut to separate s_1 from the other terminals. Let E_1 be the set of edges in this cut, and let $\delta(E_1)$ be the value of the cut. We could repeat this procedure to find the minimum cut to separate s_2 from the other terminals, and s_3 from the others, giving us E_2 and E_3 (and $\delta(E_2), \delta(E_3)$). Let OPT be the value of the best 3-Way Cut. We know that $OPT \geq \delta(E_i),\ \forall i = 1, 2, 3$, since we must at least separate one of the terminals from the rest, and $\delta(E_i)$ is the minimum way to do such. Let C be the union of the two E_i that have the smallest cut values $\delta(E_i)$ of the three. Without loss of generality let the two smallest be E_1, E_2. We know $\delta(C) \leq \delta(E_1) + \delta(E_2) \leq 2OPT$, which shows that this is a 2-approximation.

(c) The local search algorithm is to calculate for each i the E_i that is the minimum cut that separates s_i from all the other terminals. Let C be the union of the $(k-1)$ E_i with the smallest cut values. Then this algorithm is a $(k-1)$-approximation to the optimal solution.