Illinois Institute of Technology
Department of Computer Science

Solutions to Third Examination

CS 430 Introduction to Algorithms
Spring, 2009

10:30am–12:30pm, Monday, May 11, 2009
104 Stuart Building

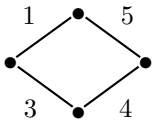
1. BFS and DFS **[20 points]**

   If F has a back edge (that is, an edge going to an ancestor), then in T that edge would be a tree edge (by the nature of BFS). Thus if $F = T$, there can be no back edges in F; because G is connected, this means that G is tree and hence it must be equal to F and T.

2. Shortest Paths **[20 points]**

   (a) **[10 points]** If the edge weights are doubled, the weight of any path in the graph doubles, so the relative lengths of the paths are unchanged and P is necessarily still a minimum-weight s–t path.

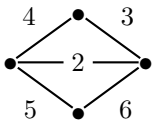
   (b) **[10 points]** If the edge weights are squared, the relative lengths of the paths can change and P is *not* necessarily still a minimum-weight s–t path. Here is a simple example:



   The upper path from the leftmost vertex to the rightmost vertex is shortest before we square the weights, but the lower path is shortest after we square the weights.

3. Spanning Trees **[20 points]**

   (a) **[2 points]** The minimum spanning tree must be unique because as we execute Kruskal's algorithm each step has just one edge to choose (there are no ties because all weights are distinct). **[3 points]** The second-best MST is not necessarily unique, as the following example shows:



In this graph the minimum spanning tree is the one with edge weights 2, 3, 5; there are two second-best spanning trees, one with edge weights 2, 4, 5 and the other with edge weights 2, 3, 6.

(b) **[8 points]** Let T' be a second best minimum spanning tree. Let $(u, v) \in T - T'$. Then, $T' + (u, v)$ contains a cycle and one of the edges in the cycle is not in T. Let this edge be (x, y). Then we must have $w(x, y) > w(u, v)$, for otherwise, we could replace (u, v) in T by (x, y) to get a spanning tree better than T. But notice that $T'' = T' - (x, y) + (u, v)$ is also a spanning tree because (u, v) and (x, y) are in the same cycle. Moreover, the weight of T'' is less than the weight of T'; that is, T'' must be a minimum spanning tree. By the uniqueness of the minimum spanning tree in part (a), we have $T'' = T$ and T and T' dier by only one edge.

(c) **[Extra 5 points]** For each $u \in V$, perform a breadth first search to fi the maximum weight path between u and every other $v \in V$. The time complexity of breadth first search is $O(|V| + |E|)$. Since T is a spanning tree, it has $|V| - 1$ edges. So the time complexity of this algorithm is $O(|V|^2)$.

(d) **[7 points]** The algorithm works as follows: use Prim's or Kruskal's algorithm to find a minimum spanning tree, T. Then find an edge $(x, y) \in E - T$ that minimizes $w(x, y) - w(max[x, y])$, where $max[x, y]$ is found as in part (c). Give as output the tree $T' = T - max[x, y] + (x, y)$.

The time required is then $O(|E| + |V| \lg |V|)$ to find the minimum spanning tree with Prim's algorithm or $O(|E| \lg |V|)$ with Kruskal's algorithm. Finding an edge $(x, y) \in E - T$ that minimizes $w(x, y) - w(max[x, y])$ takes time $O(|V|^2)$ from part (c). Thus if we use Prim's algorithm the overall time is $O(|V|^2)$; if we use Kruskal's algorithm the overall time is$O(|E| \lg |V| + |V|^2)$.

4. NP-Hardness **[20 points]**

(a) **[10 points]** All we need for membership in NP is a polynomial verification algorithm—that ism given two ascending sequences of indices, $r_1 < r_2 < \cdots < r_s$ and $c_1 < c_2 < \cdots < c_t$, we simply check that $F[r_i, c_j] = T[i, j]$, for all $1 \leq i \leq s$, $1 \leq j \leq t$. This takes time (st), clearly polynomial in the input size.

(b) **[10 points]** Following the hint, if we are given a BCBS problem to solve, we transform it to a pattern-in-picture problem by considering the adjacency matrix of a the graph as a picture and take the pattern to be a $k \times k$ array of 1s. Now, if the BCBS input graph $G = (V, E)$ has two disjoint subsets V_1, V_2 of V such that $|V_1| = |V_2| = k$ and such that for every $v \in V_1$ and every $u \in V_2$, there is an edge $(u, v) \in E$, the vertices of V_1 and V_2 give us the location of the pattern. Conversely, if picture has the pattern within it, the subsequences $r_1 < r_2 < \cdots < r_s$ and $c_1 < c_2 < \cdots < c_t$ identify the sets of vertices V_1 and V_2.

5. Approximation Algorithms **[20 points]**

(a) **[5 points]** Maintain a priority queue of the shortest distance to each vertex not yet on the tour from a city already on the tour, keeping track not only of the distance, but also

the city on the tour. Initially, we have a starting city, so we need to do $n-1$ INSERT operations. Then, as long as the priority queue is not empty, we do an EXTRACT-MIN and insert the city extracted into the tour next to the city to which it was closest. Then we do a DECREASE-KEY for each city not on the tour for which the distance from the newly added city is smaller than the value in the queue, changing the closest city for that city. Thus we do $n-1$ EXTRACT-MIN operations and $(n-2)+(n-3)+\cdots+0 = \Theta(n^2)$ DECREASE-KEY operations.

Using the table on page 456 as a guide, we see that if we use a Fibonacci heap for the priority queue, the cost of each of the $n-1$ INSERT operations is $O(n)$, the cost of the $n-1$ EXTRACT-MIN operations is $O(n \log n)$, and the cost of the $\Theta(n^2)$ DECREASE-KEY operations is $\Theta(n^2)$ for a total time of $\Theta(n^2)$.

(b) **[5 points]** In the simplest case, when $k > j > i$, the cost of going from city i to city j to city k is the sum of $|j-i| + |k-j| = |k-j|$ which by definition is no smaller than the cost of going from city i to city k. Other relative orderings of the city numbers have similar analyses.

(c) **[5 points]** The optimal tour is 1—2—3— $\cdots$ —n—1 with cost n. Clearly no tour of all n can have cost less than n, so this is optimal.

(d) **[5 points]** The heuristic can produce the tour $\cdots$ —5—3—1—2—4—6— $\cdots$ which has cost $2n-2$.