CSE421: Design and Analysis of Algorithms

Homework 4

Anup Rao Due:

An algorithm is said to run in polynomial time if it runs in time $O(n^d)$ for some constant d on inputs of size n. Each problem is worth 10 points:

1. Prove or disprove the following: Given any undirected graph G with weighted edges, and a minimum spanning tree T for that G, there exists some sorting of the edge weights $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$, such that running Kruskal's algorithm with that sorting produces the tree T.

   Solution: We shall show that there is a sorted order producing the MST T. Choose a sorted order $w(e_1) \leq \ldots \leq w(e_m)$ satisfying the property that if e_i, e_j are such that $w(e_i) = w(e_j)$ and e_i belongs to the tree but e_j does not, then e_i comes before e_j. In other words, within the edges of the same weight, put the edges of the tree first.

   We shall prove by induction on i that at the i'th step, Kruskals algorithm will include e_i if and only if e_i belongs to T.

   For the base case of e_1, observe that the algorithm must include e_1, and e_1 must belong to the tree. This is because if e_1 does not belong to T, then neither does any other edge of the same weight. So, adding e_1 to T gives a cycle, and all the other edges of the cycle must have larger weight than e_1. Then if we delete one of the other edges from the cycle we obtain a new tree of lighter total weight, contradicting the fact that T is an MST.

   Now, for $i > 1$, let K denote the edges picked by the algorithm so far. If e_i belongs to the tree then Kruskal's algorithm will include e_i, since this does not create any cycles in the edges K (which all belong to T by induction).

   If e_i does not belong to the tree, then we claim that e_i must create a cycle among the edges of K, and so Kruskal's algorithm will not include it. This is because e_i must create a cycle in T, so if it does not create a cycle in K, then some edge of the cycle in T does not belong to K, so this edge must be heavier than e_i. Adding e_i to T and deleting the heavier edge gives a cheaper tree, which is not possible since T is an MST.

2. You are given a graph G with n vertices and m edges, and a minimum spanning tree T of the graph. Suppose one of the edge weights $w(e)$ of the graph is updated.

   (a) Give an algorithm that runs in time $O(m)$ to test if T still remains the minimum spanning tree of the graph. You may assume that all edge weights are distinct both before and after the update. HINT: If $e \in T$, consider the cut obtained by deleting e from T. If $e \notin T$, consider the cycle formed by adding e to T.

   (b) Suppose T is no longer the minimum cost spanning tree. Give an $O(m)$ time algorithm to update the tree T to the new minimum cost spanning tree.

Solution. We start by proving two claims:

Claim 1. *Suppose the updated edge $e = \{u, v\}$ is in T. Let A and B denote the connected components of T obtained after deleting e. If the weight of edge e was decreased, then T remains an MST. If the weight of e was increased, let e' be the lightest edge crossing from A to B. Then the tree $T - e + e'$ is the unique MST of the new graph.*

Proof First observe that if the weight of e was decreased by c, then the cost of every spanning tree can decrease at most by c, and the cost of T decreases by c, so T must remain an MST.

If the weight of e was increased, let e' be the lightest crossing edge from A to B. For every other edge f of T, we proved in class that f must be the lightest crossing edge of some cut in the graph before the update, since f would be chosen in an execution of Kruskal's algorithm. So, f must retain this property after the update, since e is the only edge whose weight was changed, and the weight of e was increased. Thus, every such edge f remains in the MST by the cut property. Moreover, e' must also be in every MST by the cut property. So, $T - e + e'$ is the unique MST of the updated graph. ■

Claim 2. *Suppose the updated edge $e = \{u, v\}$ is not in T. Then if the weight of e is increased, T remains an MST. If e is added to T, then e creates a cycle C in T. If the weight of e was decreased, let e' denote the heaviest edge of C. Then $T + e - e'$ is the unique MST of the new graph.*

Proof If the weight of e is increased, then the cost of every MST can only increase, while the cost of our tree T remains the same, so T remains an MST.

If the weight of e is decreased and e' is the heaviest edge of the cycle, then e' certainly cannot be part of any MST by the cycle property. For all other edges $f \neq e$ that do not belong to T, we know that f must be the heaviest edge of a cycle in the graph before the update. This remains true after the update, since the update only decreased the weight of an edge. So, every such f cannot be part of any MST. Because any MST cannot contain f as above, and cannot contain e', the only possible MST in the graph is $T + e - e'$. ■

Given the claims above, the algorithm is as follows:

(a) If $e \in T$ and the weight of e was decreased, or $e \notin T$ and the weight of e was increased, then output that the T remains an MST.

(b) Otherwise, if $e \in T$, use BFS to compute A, and then run over all edges to find the lightest crossing edge from A to B. If $e = e'$ output that T remains an MST, otherwise output the tree $T - e + e'$.

(c) If $e \notin T$, use BFS to compute the cycle C, and let e' be the heaviest edge of C. If $e = e'$ output that T remains an MST, otherwise output $T + e - e'$.

The correctness of the algorithm follows from the claims above. The running time is $O(m)$, since BFS runs in time $O(m)$.

3. Suppose you are choosing between the following three algorithms:

   - Algorithm A solves the problem by dividing it into five subproblems of half the size, recursively solves each subproblem, and then combines the solution in linear time.
   - Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$, and then combines the solution in constant time.
   - Algorithm C solves the problem by dividing it into nine subproblems of one third the size, recursively solves each subproblem, and then combines the solutions in quadratic time.

   What are the running times of each of these algorithms?

   Solution: Algorithm A has the recurrence

   $$T(n) \leq 5T(n/2) + O(n).$$

   Since $5 > 2^1$ this is a case where the leaves dominate. The running time is $O(n^{\log_2 5})$.

   Algorithm B has the recurrence

   $$T(n) \leq 2T(n-1) + O(1).$$

   The i'th level of the tree has total work $O(2^i)$, and there are $n-1$ levels. So the running time is

   $$\sum_{i=0}^{n-1} O(2^i) = O(1) \cdot (2^n - 1)/(2-1) = O(2^n).$$

   Algorithm C has the recurrence

   $$T(n) \leq 9T(n/3) + O(n^2).$$

   Since $9 = 3^2$, this algorithm has running time $O(n^2 \log n)$ by the master theorem.

4. You are given two sorted lists of integers of length m and n. Give an $O(\log m + \log n)$ time algorithm for computing the k'th smallest integer in the union of the lists.

   Solution: We shall carry out a variant of binary search. Let $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ be the given lists.

   (a) If $m = 1$
      i. If $y_{k-1} \leq x_1 \leq y_k$, output x_1,
      ii. Else If $x_1 \leq y_k$, output y_{k-1}
      iii. Else output y_k.

   (b) If $n = 1$
      i. If $x_{k-1} \leq y_1 \leq x_k$, output y_1,
      ii. Else If $y_1 \leq x_k$, output x_{k-1}
      iii. Else output x_k.

(c) If $\lfloor m/2 \rfloor + \lfloor n/2 \rfloor \geq k$ then

   i. If $x_{\lfloor m/2 \rfloor} \leq y_{\lfloor n/2 \rfloor}$, then recursively find the k'th smallest number in $x_1, \ldots, x_m$ and $y_1, \ldots, y_{\lfloor n/2 \rfloor}$.

   ii. If $x_{\lfloor m/2 \rfloor} > y_{\lfloor n/2 \rfloor}$, then recursively find the k'th smallest number in $x_1, \ldots, x_{\lfloor m/2 \rfloor}$ and $y_1, \ldots, y_n$.

(d) If $\lfloor m/2 \rfloor + \lfloor n/2 \rfloor < k$ then

   i. If $x_{\lfloor m/2 \rfloor} \leq y_{\lfloor n/2 \rfloor}$, then recursively find the k'th smallest number in $x_{\lfloor m/2 \rfloor + 1}, \ldots, x_m$ and $y_1, \ldots, y_n$.

   ii. If $x_{\lfloor m/2 \rfloor} > y_{\lfloor n/2 \rfloor}$, then recursively find the k'th smallest number in $x_1, \ldots, x_m$ and $y_{\lfloor n/2 \rfloor + 1}, \ldots, y_n$.

To efficiently implement the recursive calls, we only pass in the end-points of the new input intervals used, rather than copying the whole input into a new array.

To prove correctness, when $n = 1$ or $m = 1$, the algorithm uses the sorted lists to find the k'th smallest element in constant time. In the other cases, the algorithm always eliminates half of one list. In case (b), i, we must have that the k'th smallest number is at most $y_{\lfloor n/2 \rfloor}$, since there are $\lfloor m/2 \rfloor + \lfloor n/2 \rfloor \geq k$ numbers that are at most $y_{\lfloor n/2 \rfloor}$. Thus, the recursive step will correctly find the k'th smallest number in the overall list. All the other cases hold for the same reason: in each case we eliminate one half of one of the lists.

There can be at most $O(\log n) + O(\log m)$ recursive calls, because in each call one of the lists is halved.