CS 4820, Spring 2014	**Homework 2, Problem 1**
Name: (your name)	
NetID: (your netid)	
Collaborators: (your collaborators)	

(1) *(10 points)* Consider the following candidate algorithm for computing a minimum weight spanning tree of a graph with n vertices and m edges.

The algorithm builds a subgraph proceeding in rounds and starting from the empty subgraph. In each round, for each connected component of the current subgraph, the algorithm selects an edge of minimum weight that leaves the component. At the end of the round it adds these selected edges to the subgraph. The algorithm stops when the subgraph is connected.

(1.a) Show that the subgraph computed by the algorithm is a minimum weight spanning tree of the input graph. You may assume that the input graph is connected and that all edge weights are distinct.

(1.b) Show how to implement the algorithm in time $O(m \log n)$.

Warning: *This minimum spanning tree algorithm is moderately well known, and it is possible to find analyses of it on the Web. If you use Web sources you must cite them, and citing a Web source that contains an analysis of this algorithm will result in loss of points. For this reason, you are advised to solve the problem using your textbook and lecture notes (and office hours or Piazza, if desired), but not consulting the Web.*

Solution

Part a. Note that the algorithm always terminates. The reason is that in every round at least one new edge is added to the subgraph.

The correctness of the algorithm follows directly from the cut property of MSTs.

Claim 1. The edges added by the algorithm to the subgraphs are contained in every MST of the graph.

Proof. Consider an edge e added to the subgraph in one of the rounds. Then the subgraph contains a component S such that e is the minimum weight edge leaving that component. By the cut property (for the cut S), the edge e is contained in every MST of the graph.

Claim 2. The final subgraph is a MST.

Proof. Consider any MST T^*. By Claim 1, the final subgraph is a subgraph T^*. At the same time, the final subgraph is connected (the algorithm only stops when the subgraph is connected). Therefore, the final subgraph cannot be a proper subgraph of T^*. It follows that the final subgraph is equal to T^*.

Part b. The crux of the running time analysis is the following property of the algorithm. The following claim shows that the number of rounds is bounded by $O(\log n)$.

Claim 3. The number of components decreases at least by a factor of 2 in each round.

Proof. In each round, the number of components decreases by the number of edges added in the current round (each edge joins two distinct components). (Note that the correctness of the algorithm implies that no cycles are created in the subgraph.) Therefore, we are to show that the number of new edges added in a round is at least half of the number of components. The reason for that is that each component selects an edge and every edge can be selected by at most two of the components (an edge leaves at most two components).

Claim 4. The algorithm can be implemented in time $O(m \log n)$ plus the time for $O(m \log n)$ `find` operations and $n - 1$ `union` operations.

Proof. To implement the algorithm we pursue the following strategy:

We start by creating a union-find data structure to keep track of the components of the subgraph during the computation. To implement a round, we make a pass over all edges in the graph (in any order). For each edge we use our union-find data structure to identify the components of the endpoints of the edge (two `find` operations per edge). For each component, we keep track of the minimum weight edge leaving the component among the edges we have seen so far in the current round. (We can use an array indexed by the names of the components to keep track of these edges.) After we made a pass over all the edges, we make an additional pass over the minimum weight edges for the components and for each such edge, we join the components of their endpoints (one `union` operation per selected edge).

Ignoring the union operations, each round can be implemented in time $O(m)$ plus the time for $2m$ `find` operations. By Claim 3, the number of rounds is $O(\log n)$. Thus, without the union operations, the total time is $O(m \log n)$ plus the time for $O(m \log n)$ `find` operations. Since each `union` operation corresponds to an edge in the final subgraph, the total number of `union` operations is $n - 1$.

By (4.23) in Section 4.6 of the textbook, an array-based implementation of the union-find data structure supports $O(m \log n)$ find operations and $n - 1$ union operations in time $O(m \log n)$.

CS 4820, Spring 2014	**Homework 2, Problem 1**
Name: (your name)	
NetID: (your netid)	
Collaborators: (your collaborators)	

(2) *(10 points)* Solve Exercise 4.16 in Chapter 4 of the Kleinberg-Tardos textbook.

Hint: The simplest proof of correctness that we know of uses an exchange argument.

Karma exercise: Show how to implement your algorithm in time $O(n \log n)$.

Solution

The algorithm initializes a set of *events* denoted by their occurrence times x_i $(i = 1, \ldots, n)$, and a set S of *intervals* denoted by their left endpoints $t_i - e_i$ and right endpoints $t_i + e_i$ $(i = 1, \ldots, n)$. As usual, we say that an interval with endpoints a, b *contains* an event x if it is the case that $a \leq x \leq b$.

The events are sorted in increasing order of occurrence time, and they are processed one by one in that order. When each event x_i is processed, the algorithm identifies all of the intervals in S that contain x_i. If no interval in S contains x_i, then the algorithm terminates and declares that no association exists. Otherwise, among all the intervals in S that contain x_i, it selects the one whose right endpoint $t_j + e_j$ is earliest. It associates x_i with t_j, removes the interval $(t_j - e_j, t_j + e_j)$ from the set S, and continues processing the next event in the list. If it reaches the end of the list of events and finds that it has associated each one with a timestamp t_j, then it terminates and declares that an association exists.

Analysis of running time. Sorting the events by occurrence time takes $O(n \log n)$ time. For each event x_i, we need to compare it with at most n intervals to determine which intervals contain x_i; each comparison requires $O(1)$ operations, so the total time spent identifying the intervals that contain x_i is $O(n)$. Among the intervals that contain x_i (at most n of them) we can find the one with the rightmost endpoint using an additional $O(n)$ steps. Thus, the algorithm performs $O(n)$ work per event, for a total running time of $O(n^2)$. (The $O(n \log n)$ running time of the initial sorting operation has lower order of growth than $O(n^2)$, hence it can be omitted from the final running time bound.)

Correctness. Clearly, if the algorithm declares that an association exists, then this declaration is correct because the algorithm, during its execution, has actually computed one valid association. The harder part of proving correctness is to show that if the algorithm declares than an association does not exist, then this answer is correct. We will prove this by contradiction, using an exchange argument. Suppose that the algorithm terminates after associating the first $k < n$ events with timestamps; let A denote this partial association. (We interpret A as a function from the set of events to the set of timestamps.) Suppose, for contradiction, that there exists a complete association of events with timestamps, B, that satisfies the constraint that every event is associated to a different timestamp, and that if x_i is associated to t_j by B, then $|x_i - t_j| \leq e_j$. Assume without loss of generality that the events $x_1, x_2, \ldots, x_n$ are sorted in increasing order of their occurrence time. Define the *agreement parameter* $\alpha(A, B)$ to be the largest index i such that each events $x \in \{x_1, \ldots, x_i\}$ satisfies $A(x) = B(x)$. In other words, $\alpha(A, B)$ measures the

length of the longest prefix of the event sequence on which A and B agree. Among all the valid complete associations B, let us choose the one that maximizes $\alpha(A, B)$. We claim that in that case, $\alpha(A, B) = n$, contradicting the fact that, by assumption, A makes only $k < n$ assignments in total.

To prove that $\alpha(A, B) = n$, we again argue by contradiction. If $\alpha(A, B) = j < n$, then let $t_p = B(x_{j+1})$. We know that the greedy algorithm had the option of assigning x_{j+1} to t_p — because it only deleted the timestamps to which $x_1, \ldots, x_j$ were assigned, and B made the same assignments for these events — but we also know that $A(x_{j+1}) \neq B(x_{j+1})$ because $\alpha(A, B) = j$. Hence $A(x_{j+1}) = t_q$ for some $q \neq p$. Furthermore, since the greedy algorithm always selects the intervals whose right endpoint is earliest, we know that $t_q + e_q \leq t_p + e_p$. Now, since B is a complete association of events to timestamps, there must be some other event $x_m \leq x_{j+1}$ such that $B(x_m) = t_q$. (See Figure 1.) Consider modifying B to a different association B' by swapping x_{j+1} and x_m. In other words, B' is the same as B except that $B'(x_{j+1}) = t_q$ and $B'(x_m) = t_p$. We claim that B' is a valid association of events to timestamps. It is clear that B' defines a one-to-one correspondence because it is obtained from B by simply swapping two values. Furthermore, we know that $|x_{j+1} - t_q| \leq e_q$ because $A(x_{j+1}) = t_q$. Finally, the fact that $|x_m - t_p| \leq e_p$ is proven by the string of inequalities

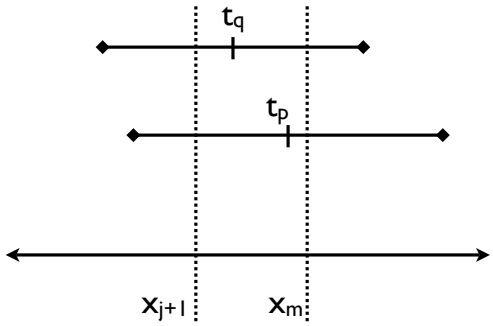

Figure 1: Exchange argument for P2.

$$t_p - e_p \leq x_{j+1} \leq x_m \leq t_q + e_q \leq t_p + e_p$$

where the first and third inequalities follow from the fact that B is a valid assignment, whereas the second and fourth inequalities have been proven earlier in this paragraph. We have thus shown that B' is a valid complete association of events to timestamps. Also note that $\alpha(A, B') \geq j + 1$, contradicting our assumption about the maximality of $\alpha(A, B)$. This completes the proof by contradiction, and establishes the correctness of the greedy algorithm.

CS 4820, Spring 2014	**Homework 2, Problem 1**
Name: (your name)	
NetID: (your netid)	
Collaborators: (your collaborators)	

(3) *(10 points)* In class, we have seen several examples of optimization problems. These problems come with a notion of what constitutes a solution, and they come with an objective function that measures the quality of a solution. The goal is to find a solution that minimizes (or maximizes) the objective function. For example, in the minimum spanning tree problem, solutions are spanning trees of the given graph and the objective function is the total edge weight of a spanning tree. In the scheduling problem to minimize lateness, solutions are valid schedules for the given requests and the objective function is the maximum lateness of a request.

This way of framing the task of algorithm design — that problems have predefined, mathematically precise objectives, and that the the algorithm designer's job is to solve those predefined problems — is convenient from the standpoint of teaching lectures on algorithms or assigning exercises, but it isn't always an accurate reflection of how algorithm design takes place in real life. In many cases, there is more than one objective function that may be deemed appropriate for the problem at hand. For example for spanning trees, an alternative objective function could be the maximum weight of an edge of the tree (instead of the sum of the weights), whereas for valid schedules, we could consider the sum of the latenesses of requests (instead of the maximum lateness).

What is the right objective function? The choice of the objective function is often guided (and sometimes misguided) by applications and heuristic reasoning. However, it is important to take algorithmic considerations into account. For some objective functions the problem may be hard to solve, whereas for other objective functions, there may be efficient algorithms to solve the problem. In such cases, it often happens that the precise formulation of the objective is dictated by the choice of algorithm and not the other way around.

The following problems study the interaction between algorithms and objective functions.

(3.a) Consider the scheduling problem in Section 4.2 of the textbook. Suppose the goal is to minimize the sum of the latenesses of requests. Show that for this objective function, the earliest-deadline-first algorithm does not always find an optimal schedule.

(3.b) Again consider the scheduling problem in Section 4.2 of the textbook. Suppose every request has a positive weight w_i and the goal is to minimize the weighted sum of latenesses $\sum_i w_i \ell_i$.

Give an efficient algorithm for the special case that all deadlines are equal to the time the resource becomes available (i.e., $d_i = s$ for all $i \in \{1, \ldots, n\}$).

Karma exercise: One of the remarkable features of the minimum spanning tree problem is its robustness: in a very strong sense, the minimum spanning tree is optimal for any reasonable choice of the objective function, not just for the sum of the edge weights. In this sense, the minimum spanning tree problem behaves very differently from the scheduling problem addressed in problems 3a and 3b above.

More precisely, suppose f is a function that assigns a value to every $(n-1)$-tuple of numbers, and suppose

- f **is symmetric:** its value is invariant under permuting the elements of the $(n-1)$-tuple;
- f **is monotone:** its value never decreases when we increase one element of the $(n-1)$-tuple while holding the other elements fixed.

Examples of symmetric monotone functions are the sum, the maximum, and the median.

For any such f, we can consider the "f-minimum spanning tree problem" in which the goal is to find a spanning tree that minimizes the value of f, when it is applied to the $(n-1)$-tuple of edge weights in the tree. Prove that any minimum spanning tree of an edge-weighted graph is also an f-minimum spanning tree for *every* symmetric, monotone objective function f.

Solution

Part a. One idea to construct a counterexample: If the goal is to minimize the maximum lateness, it doesn't matter how many jobs are late. On the other hand, if the goal is to minimize the sum of latenesses, then schedules with a small number of late jobs are preferable.

Here is a counterexample that follows this idea: The starting time for the resource is $s = 0$. We have one request with length k and deadline k. In addition, we have k request of length 1 and deadline $k + 1$.

In the earliest-deadline-first schedule, the request of length k is scheduled first and the other requests are scheduled after it. For k large enough, roughly half of the short requests have lateness at least $k/2$. Therefore, the sum of latenesses is $\Omega(k^2)$.

At the same time, if we schedule the short requests first and the long request after them, the total lateness is k (the short requests finish before their deadline and the long request has lateness k).

It follows that for large enough k, the schedule computed by the earliest-deadline-first algorithm is not optimal (and the gap is quite large).

Part b. The following solution refers to requests as websites and refers to the weights w_i as costs c_i.

The algorithm restores the websites in decreasing order of c_i/t_i, where c_i is the rate of lost dollars per hour for site i, and t_i is the number of hours to finish the job.

Analysis of running time. Computing the ratio for each job requires $O(n)$ time, and sorting them requires $O(n \log n)$ time.

Correctness. The proof of correctness uses an exchange argument, similar to the proof of correctness of the scheduling algorithm in Section 4.2 of Kleinberg & Tardos. If you have a schedule with two consecutive jobs i, j (in that order) and you swap the order of i and j, the net change can be broken down as follows.

- Site i is restored t_j hours later, which increases the lost revenue by $c_i t_j$.
- Site j is restored t_i hours earlier, which decreases the lost revenue by $c_j t_i$.
- All other sites are restored at the same time.

Thus the net change in lost revenue is

$$c_i t_j - c_j t_i = \left(\frac{c_i}{t_i} - \frac{c_j}{t_j} \right) t_i t_j. \tag{1}$$

This implies the following facts.

1. In any optimal schedule, there can't be two consecutive jobs i, j such that $(c_i/t_i) - (c_j/t_j)$ is negative. If there were, one could decrease the lost revenue by swapping the order of i and j.
2. Therefore, in every optimal schedule, the jobs are sorted in order of decreasing c_i/t_i.
3. Swapping the order of two consecutive jobs i, j such that $c_i/t_i = c_j/t_j$ has no effect on the amount of lost revenue.
4. Therefore, for every ordering of the jobs such that the ratios c_i/t_i form a non-increasing sequence, the cost is equal to that of an optimal schedule.