CS 570 - HW 2 solutions

Due Sunday, February 16 (by 23:30)

1

Sort edges by weights. Greedily remove the minimum edge until s and t is unconnected. Then add the last removed edge back and run graph traversal algorithm to find a path from s to t. Time complexity is $O(E(V+E))$.

Proof:
Denote our solution as P and the optimal path as P^*. If P^* is not in our final graph, some edges in P^* are removed because of smaller edge weight. So the value of P^* is not larger than P. Because P^* is optimal, the value of P^* is not smaller than P. So P and P^* have the same value, that is, our derived path is optimal.

Partial Credit:

- Algorithm is greedy - 4 points.
- Runtime is $O(E(V+E))$ - 2 points.
- Proof - 4 points.

2

Proof by contradiction:
If T_1 and T_2 are distinct minimum spanning trees, then consider the edge of minimum weight among all the edges that are contained in exactly one of T_1 or T_2. Without loss of generality, this edge appears only in T_1, and we can call it e_1.
Then $T_2 \cup \{e_1\}$ must contain a cycle, and one of the edges of this cycle, call it e_2, is not in T_1.
Since e_2 is an edge different from e_1 and is contained in *exactly one* of T_1 or T_2, it must be that $w(e_1) < w(e_2)$. Note that $T = T_2 \cup \{e_1\} \setminus \{e_2\}$ is a spanning tree. The total weight of T is smaller than the total weight of T_2, but this is a contradiction, since we have supposed that T_2 is a minimum spanning tree.

Partial Credit: No.

3

To do this, we apply the Cycle Property nine times. That is, we perform BFS until we find a cycle in the graph G, and then we delete the heaviest edge on this cycle. We have now reduced the number of edges in G by one while keeping G connected and (by the Cycle Property) not changing the identity of the minimum spanning tree. If we do this a total of nine times, we will have a connected graph H with $n-1$ edges and the same minimum spanning tree as G. But H is a tree, and so in fact it is the minimum spanning tree. The running time of each iteration is $O(m+n)$ for the BFS and subsequent check of the cycle to find the heaviest edge; here $m \leq n+8$, so this is $O(n)$.

Partial Credit:

- Finding the algorithm - 4 points
- Proof - 3 points
- Runtime - 3 points

4

Let the sequence S consist of $s_1, ..., s_n$ and the sequence S' consist of $s'_1, ..., s'_m$. We give a greedy algorithm that finds the first event in S that is the same as $s'1$, matches these two events, then finds the first event after this that is the same as s'_2, and so on. We will use $k_1, k_2, ...$ to denote the match we have found so far, i to denote the current position in S, and j the current position in S'.
Initially i=j=1
While $i \leq n$ and $j \leq m$
If s_i is the same as s'_j, then
let $k_j = i$
let $i = i+1$ and $j = j+1$
otherwise let $i = i+1$
EndWhile
If $j = m+1$ return the sub-sequence found: $k_1, ..., k_m$
Else return that "s' is not a sub-sequence of S"

The running time is $O(n)$: one iteration through the while loop takes $O(1)$ time, and each iteration increments i, so there can be at most n iterations. It is also clear that the algorithm finds a correct match if it finds anything. It is harder to show that if the algorithm fails to find a match, then no match exists. Assume that S' is the same as sub-sequence $s_{l_1}, ..., s_{l_m}$ of S. We prove by induction that the algorithm will succeed in finding a match and will have $k_j \leq l_j$ for all $j = 1, ..., m$.

For each $j = 1, ..., m$ the algorithm finds a match k_j and has $k_j \leq l_j$.

Proof: The proof is by induction on j. First consider $j = 1$. The algorithm lets k_1 be the first event that is the same as s'_1, so we must have that $k_j \leq l_j$. Now consider a case when $j > 1$. Assume that $j - 1 < m$ and assume by the induction hypothesis that the algorithm found the match k_{j-1} that is the same as s'_j, if such a character exists. We know that l_j is such a character and $l_j > l_{j-1} \geq k_{j-1}$. So $s_{l_j} = s'_j$, and $l_j > k_{j-1}$. The algorithm finds the first such index, so we get that $k_j \leq l_j$.

Partial Credit:

- Finding the algorithm - 4 points
- Proof - 3 points
- Runtime - 3 points

5

This problem can be solved by extending Dijkstra's algorithm:
Let $\mathcal{S}$ be the set of explored nodes.
For each $u \in S$, we store the earliest time $d(u)$ when we can arrive at u and the last site $r(u)$ before u on the fastest path to u.
Initially $S = \{s\}$ and $d(s) = 0$.
While $S \neq V$
Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{(u,v)} f_e(d(u))$ is as small as possible.
Add v to S and define $d(v) = d'(v)$ and $r(v) = u$.
By using priority queue, the time complexity is $O(E \log(V))$.

Rubrics:

- Time complexity is $O(E \log(V))$, $O((V + E) \log(V))$, or $O(E + V \log(V))$ - 10 points.
- Use modified Dijkstra and time complexity is $\Omega(E \log(V))$ - 8 points.
- Use other shortest distance algorithm and the time complexity is $\Omega(E \log(V))$ and polynomial - 5 points.
- Algorithm is wrong or has exponential running time - 2 points.

6

Run modified BFS from s in order to find t. The modification is that you keep track of the weight needed to reach each node and only enqueue adjacent nodes whose weight is less than or equal to the known short distance.
Once you have found t, you're done.

Proof:
We already know that there is at least one path from s to t, so BFS will find a path.
Since all weights are non-negative, additional edges can only increase the overall weight of a path. For this reason, we don't need to enqueue nodes with a weight greater than the shortest distance, since they cannot be part of the shortest path.

Runtime: The added decision criterion whether to enqueue adjacent nodes adds 1 per node expansion. Since BFS expands each node at most once, it adds 1 per node, so overall we get O(2V+E) = O(n).

Partial Credit:

- Algorithm - 4 points.
- Proof - 3 points.
- Runtime - 3 points.

7

The optimal strategy is the obvious greedy one. Starting with a full tank of gas, go to the farthest gas station you can get to within n miles. Fill up there. Then go to the farthest gas station you can get to within n miles of where you filled up, and fill up there, and so on.
Looked at another way, at each gas station, you should check whether he can make it to the next gas station without stopping at this one. If you can, skip this one. If you cannot, then fill up. You don't need to know how much gas you have or how far the next station is to implement this approach, since at each fill-up, you can determine which is the next station at which you'll need to stop.
This problem has optimal substructure. Suppose there are m possible gas stations. Consider an optimal solution with s stations and whose first stop is at the kth gas station. Then the rest of the optimal solution must be an optimal solution to the sub-problem of the remaining $m-k$ stations. Otherwise, if there were a better solution to the sub-problem, i.e., one with fewer than $s-1$ stops, we could use it to come up with a solution with fewer than s stops for the full problem, contradicting our supposition of optimality.
This problem also has the greedy-choice property. Suppose there are k gas stations beyond the start that are within n miles of the start. The greedy solution chooses the kth station as its first stop. No station beyond the kth works as a first stop, since you would run out of gas first. If a solution chooses a station $j < k$ as its first stop, then you could choose the kth station instead, having at least as much gas when you leave the kth station as if you'd have chosen the jth station. Therefore, you would get at least as far without filling up again if you had chosen the kth station. If there are m gas stations on the

map, you needs to inspect each one just once. Therefore, the running time is $O(m)$.

Partial Credit:

- Finding the algorithm - 3 points
- Proof - 4 points
- Runtime - 3 points

8

If the minimum spanning tree remains intact, do nothing.

Assuming that removing the edge disconnects T, let A and B be the two trees remaining after the edge is removed from T.
Algorithm: Select the minimum weight edge $e = (x, y)$ from G_1 that reconnects A and B. Then $T' = A \cup B \cup e$ is a MST of G_1.
Proof: Suppose T' were not minimum. Then there would be a MST M of smaller weight. And either M contains e or it doesn't.

If M contains e, then it must have the form $A' \cup B' \cup e$ where A' and B' are minimum weight trees that span the same vertices as A and B. These can't have weight smaller than A and B, otherwise T would not have been an MST. So M is not smaller than T' after all, a contradiction; M can't exist.

If M does not contain e, then there must be some other path P from x to y in M. One or more edges of P pass from a vertex in A to another in B. Call such an edge c. Now, c has weight at least that of e, else we would have picked it instead of e to form T'. Note that $P \cup e$ is a cycle. Consequently, $M \setminus c \cup e$ is also a spanning tree. If c were of greater weight than e, then this new tree would have smaller weight than M. This contradicts the assumption that M is a MST. Again, M can't exist.

Since in either case, M can't exist, T' must be a MST.
Runtime: The search for this e takes $O(E)$ time (check all edges whether they connect A and B and out of those, select the one with the lowest weight). $O(N)$ if the separated portion is a leaf.

Partial Credit:

- Finding the algorithm - 3 points (subtract 1 point if they miss that the MST may not be disconnected)
- Proof - 4 points
- Runtime - 3 points

9

(1) $T(n) = \Theta(n^2)$. (2) $T(n) = \Theta(n \log n)$. (3) $T(n) = \Theta(n^3)$. (4) $T(n) = \Theta(\log n)$. (5) $T(n) = \Theta(n^{\log_2 \sqrt{7}})$.

Partial Credit: 2 points for each one.

10

By master theorem, the running time of ALG is $\Theta(n^{\log_2 7})$. When $\mathbf{a > 16}$ [1], the running time of ALG' is $O(n^{\log_4 a})$. Then we require:

$$\log_4 a < \log_2 7 \Rightarrow a < 49$$

So the largest value of a is $49 - \epsilon$. ???

Rubrics:

- Wrong answer - deduct 8 points.
- Missing [1] - deduct 3 points.