Homework 3

CS 4104 (Spring)

Assigned on February 22, 2021.
Submit PDF solutions on Canvas
by 11:59pm on March 1, 2021.

My team-mate is: Joseph McAlister

Problem 1 (20 points) Solve exercise 5 in Chapter 4 (pages 190–191) of "Algorithm Design" by Kleinberg and Tardos. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible. Just in case the problem statement is not completely clear, you can assume that the road is the x-axis, that each house lies directly on the road, and that the position of each house can be specified by its x-coordinate.

Hint: Sort the base stations by x-coordinate. Develop your algorithm now. To design your proof of correctness, follow one of the proof strategies we discussed in class (and present in the textbook) for greedy scheduling problems.

Solution:

We begin by sorting the houses in order of increasing x coordinate.

Then, from the origin $x = 0$, or from the "West-most" end of the road, we move to the right, "east-bound," until the first house is reached at some position x_i.

Place a base station at x_{i+4} and delete all houses on the range $[x_i, x_{i+8}]$ which are in the range of the base station at the midpoint of that range.

Advance to the next available house and repeat until there are no more houses, then return the set S of positions of base stations $\{x_{i+4}, ..., x_k\}$.

Algorithm 1 PLACEBASESTATIONS(H)

```
S ← ∅
L ← SORT(H)
for every house index i and house position h ∈ L do
    Add h + 4 to S
    j = i
    {while the next house in L is in range of base station, remove it from L}
    while L[j + 1] − (h + 4) ≤ 4 do
        DELETE(L[j + 1])
        j + +
    end while
end for
return S
```

Proof of Correctness:

Let O be the optimal "schedule" of base station positions $\{o_1, ..., o_m\}$, we must show that the size of the result of our algorithm, $|S|$, is the largest possible in any set of mutually compatible schedules – that using the minimum amount of stations, we cover the maximum number of houses.

In other words, we will show that $|S| = |O|$.

Let $s_1, ..., s_k$ be the positions of base stations in S in increasing order of distance from the origin $x = 0$.

Let $o_1, ..., o_m$ be the positions of base stations in O in increasing order of distance from the origin $x = 0$, with $m \geq k$.

Then, $|S| = |O| \iff m = k$.

Claim: For all indices $i \leq k$, $pos(s_i) \geq pos(o_i)$.

Proof by induction:

Base case: $i = 1$, our greedy solution is optimal since we place the first base station as far east from the first house as possible while still covering the first house, maximizing the amount of houses this base station covers.

Inductive Hypothesis: Assume that the claim holds for $i \geq 1$ such that base stations at positions $\{s_1, ..., s_i\}$ cover all the same houses as the optimal solution $\{o_1, ..., o_i\}$

Inductive Step: if we add another base station with index $i + 1$ to S, we are guaranteed that $S = \{s_1, ..., s_i, s_{i+1}\}$ will still cover all the houses that would be covered by substituting s_{i+1} with o_{i+1}.

Therefore, $pos(s_{i+1}) \geq pos(o_{i+1})$.

Claim: $k = m$

Finally, if $k > m$, then S does not cover all houses, but we know that $pos(s_m) \geq pos(o_m)$ from the above inductive proof, so then O would also fail to cover all houses.

But this is a contradiction, since we assumed that O is the optimal solution.

Therefore $k = m$, and $|S| = |O|$, so our solution is optimal. ■

Proof of Runtime Complexity:

The most costly step of our algorithm is sorting the houses in the input set H on line 2 and storing it in a list L. We can assume that this step takes $O(n \log n)$ time.

In the worst case, where each house is at least 9 miles away from one another such that no base station covers more than a single house, our for loop on line 3 will iterate $|H|$ times. In any other (better) case, the inner while loop on line 6 is negligible since we can assume that houses do not overlap on the x-coordinate, so a finite amount of houses can exist on the interval $[x_i, x_{i+8}]$ which is asymptotically insignificant.

Therefore, the overall runtime complexity is $O(n + n \log n) = O(n \log n)$ since, after we sort the input, we may have to iterate over every element in the list of size n again to place the base stations. ■

Problem 2 (30 points) Solve exercise 13 in Chapter 4 (pages 194–195) of your textbook. A small business—say, a photocopying service with a single large machine—faces the following scheduling problem. Each morning, they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps the customers happiest. Customer i's job will take t_i time to complete. Given a schedule, i.e., an ordering of jobs, let C_i denote the finishing time of job i. For example, if job j is the first to be done, we would have $C_j = t_j$, and if job j is done right after job i, then $C_j = C_i + t_j$. Each customer i also has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So the company decides to order the jobs to minimize the weighted sum of completion times, $\sum_{i=1}^{n} w_i C_i$.

Design an efficient algorithm to solve this problem, i.e., you are given a set of n jobs with a processing time t_i and weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of completion times, $\sum_{i=1}^{n} w_i C_i$.

Hint: Try to use one of the techniques we have seen for proving the correctness of greedy algorithms. Working "backwards" from what you need to prove might help you to discover the algorithm. Note that the completion times are not part of the input. The algorithm has to compute them. So you cannot sort the jobs by completion time.

Solution:

In order to minimize the weighted sum of completion times $\sum_{i=1}^{n} w_i C_i$, we will sort the jobs by decreasing order of $\frac{w_i}{t_i}$ and process them in the resulting order.

Algorithm 2 SCHEDULEJOBS(J)

1: $S \leftarrow$ SORT(J) { Sort jobs by auxiliary criteria of w_i/t_i }
2: **return** S

Proof of Correctness:

Claim: The presented algorithm is at least as good as any other optimal schedule O with inversions.

If there exists an inversion in O such that for a pair of jobs i, j where i precedes j in O, and j before i in our greedy solution S, then there also exists a pair of adjacent jobs between i, j (which we will also call i, j for simplicity) such that $\frac{w_j}{t_j} \geq \frac{w_i}{t_i}$ according to our definition of greedy being the ordering of jobs by this proportion.

Claim: Swapping i, j does not increase the weighted sum of completion times $f = \sum_{i=1}^{n} w_i C_i$. Therefore after swapping all such adjacencies in O, we will have our greedy schedule S.

Proof: If we have two inverted, adjacent jobs i, j, then swapping them in a schedule does not affect the completion time C of any other jobs.

Before swapping, i, j account for the following partial sum in f:

$$\mathrm{w}_i(C + t_i) + w_j(C + t_i + t_j)$$

After swapping, i, j account for the following partial sum in f:

$$\mathrm{w}_j(C + t_j) + w_i(C + t_j + t_i)$$

The difference between these two is:

$$\begin{aligned} & (w_j C + w_j t_j + w_i C + w_i t_j + w_i t_i) && (1) \\ -& (w_i C + w_i t_i + w_j C + w_j t_i + w_j t_j) && (2) \\ = \quad & w_i t_j - w_j t_i && (3) \end{aligned}$$

Since we know that i, j are inverted in the schedule according to the greedy definition of order according to proportion of $\frac{w}{t}$, then we know that the difference between the product of a smaller weight and a bigger time and the product of a bigger weight and a smaller time (3) will be less than or equal to zero. That is $w_i t_j - w_j t_i \leq 0$.

Therefore, swapping two inverted jobs does not increase f, and so we can swap all inverted jobs in O until we arrive at S. Since our algorithm produces a schedule with no idle time and no inversions, it is optimal.

Proof of Runtime Complexity:

Provided that we do not have to actually execute the jobs in the greedy schedule S, the only significant step in our algorithm is the initial sort on line 1 which we can assume takes $O(n \log n)$ time.

Problem 3 (20 points) Consider the version of Dijkstra's algorithm shown below written by someone with access to a priority queue data structure that supports *only* the INSERT and EXTRACTMIN operations. Due to this constraint, the difference between this version and the one discussed in class is that instead of the CHANGEKEY$(Q, x, d'(x))$ operation in Step 8, this version simply inserts the pair $(x, d'(x))$ into Q. The danger with this algorithm is that a node x may occur several times in Q with different values of $d'(x)$. Answer the following questions.

1. (6 points) When the algorithm inserts a pair (x, d_1) into Q, suppose the pair (x, d_2) is already in Q. What is the relationship between d_1 and d_2?

   Solution: $d_1 < d_2$

2. (8 points) Using this relationship, how will you fix this algorithm? You just have to describe your correction in words, e.g., by saying "I will add the following command after Step X: ..." You do not have to prove the correctness of your algorithm.

   Solution:

   By adding an if statement after line 3 encompassing lines 4-10 to check whether or not v has been visited before the inner loop on lines 5-10 (i.e. If not $v \in S$), we can take advantage of the above relationship as well as fact that the PriorityQueue being used to maintain the tuples $(v, d'(v))$ can be implemented as a Min-Heap, meaning that if $d_1 < d_2$, ExtractMin will retrieve the tuple corresponding to the smaller of the two distance values first. If we check if the node has already been visited, i.e. v is already in S, then we can simply skip on to the next node.

3. (6 points) What is the running time of this modified algorithm? Just state the bound in terms of the number of nodes n and the number of edges m in G.

   Solution: Since checking membership of a set is a constant time operation, the runtime of the algorithm remains $O(m \log n)$.

Algorithm 3 Dijkstra's Algorithm(G, l, s)

```
1: Insert(Q, s, 0).
2: while S ≠ V do
3:    (v, d'(v)) = ExtractMin(Q)
4:    Add v to S and set d(v) = d'(v)
5:    for every node x ∈ V − S such that (v, x) is an edge in G do
6:       if d(v) + l(v, x) < d'(x) then
7:          d'(x) = d(v) + l(v, x)
8:          Insert(Q, x, d'(x))
9:       end if
10:   end for
11: end while
```

Problem 4 (30 points) Gasp! Your best friend Will Byers is trapped in *The Upside Down* yet again!! You have to rescue him from the Demogorgon!!! Fortunately, due to his repeated forays into *The Upside Down* in previous years, you have a very good map of this dimension. Will has figured out several hiding spots where the Demogorgon cannot see him; he is currently secreted in one of these spots. Moreover, due to Eleven's deep sensory perception, you have very good estimates of how safe it is to travel from one hiding spot to another without being devoured by the Demogorgon. Finally, you also know the locations of several interdimensional portals between our world and The Upside Down.

Using the algo-fu you have gained in CS 4104, you represent *The Upside Down* as an undirected graph $G = (V, E)$, where each node in V is either a hiding place or an interdimensional portal. Every edge (u, v) connects two nodes in V. The weight $w(u, v)$ of this edge is the probability that as you go from u to v (or v to u, since the edge is undirected), the Demogorgon will not devour you. Clearly, this weight is between 0 and 1 since it is a probability. The weight of a path in G is the product of the weights of its edges. This weight denotes the probability that the Demogorgon will not eat you as you traverse this path. With this set up, you formulate the `OperationSaveWillByers` problem:

> Given an undirected graph $G = (V, E)$, where every edge (u, v) is associated with a weight $w_{u,v}$ that lies between 0 and 1, a subset $S \subset V$ of nodes, a node t, and a parameter r that also lies between 0 and 1, is there any node in S such that the weight of the path from this node to t is at least r?

In this dimension (alas, we are back to the real world), your task is to find a problem X that we have discussed in class and use an algorithm A that we developed for X to solve `OperationSaveWillByers`. You are not allowed to change A. However, you can modify G to your heart's content and pass this modified version to A. Moreover, you can further manipulate the output to A to obtain a solution for `OperationSaveWillByers`. In other words, you use A purely as a subroutine without changing its internal steps. *Note:* The Demogorgon will devour all your points if you deviate from these instructions. *Hint:* I set this problem before describing any algorithm for the minimum spanning tree problem.

Solution:

In order to convert multiplication into addition and maximisation to minimisation at the same time to solve this problem, we convert all the weights of the edges E of G to be the negative log of their previous value.

Since all the weights are guaranteed to be between 0 and 1, the log value will be a negative value. The "safest" path, then would be the path whose sum of weights is closest to zero. By further negating the values such that the negative log probabilities are now all positive, the problem becomes one of minimization, where r now represents risk instead of safety.

Assuming the return value of Dijkstra's algorithm is a list of distances of all the reachable nodes in the input graph from the specified starting node t, we can iterate over the list of known interdimensional portal nodes in the graph, and check if the distance from t to each of these nodes is less than r. If so, we return true, otherwise the algorithm returns false after checking the cost of all the paths from t to and interdimensional node in S.

Algorithm 4 OperationSaveWillByers(G, S, t, r)

$E' \leftarrow \emptyset$ {Initialize an empty list of edges to store the $-\log$ edge weights}
for each $e \in E \in G$ **do**
  $e' \leftarrow -\log(e)$ {Compute $-\log e \quad \forall e \in E$}
  Add e' to E'
end for
$G' = (V, E')$ {Initialize a graph which contains the modified weights and all the nodes in G}
$P \leftarrow$ Dijkstra's(G', E', t) {Compute the shortest distances from all nodes $v \in G$ to t}
for each known interdimensional portal node $u \in S \subset V$ **do**
  $P_u \leftarrow P[u]$ {P_u is the distance from the interdimensional portal node to t, the sum of the weights of each edge comprising the path $u \rightarrow t$}
  if $P_u < r$ **then**
    return true {If the weight of any path from $u \rightarrow t$ is less than r, then it is safe}
  end if
end for
return false {If none of the paths satisfy the constraint imposed by r, return false, R.I.P. Will Byers}