Foundations of Algorithms Homework 3 SU24

Strategies for solving ***Collaborative Problem*** problems are to be formed by your assigned group, with each member *participating and contributing fully*. All answers must be your own work, in your own words, even on collaborative problems—collaborate on strategies only. Individual participation will be evaluated through collaborative assessments, submitted when the homework is due.

1. [15 points total] Consider the following algorithm for doing a postorder traversal of an AVL binary tree with root vertex $root$.

Algorithm 1. Postorder Traversal

```
function POSTORDER($root$)
    if $root \neq$ null then
        POSTORDER($root.left$)
        POSTORDER($root.right$)
        visit $root$
```

   Prove that this algorithm run in time $\Theta(n)$ when the input is an n-vertex binary tree.

2. [15 points total] We define an AVL binary search tree to be a tree with the binary search tree property where, for each node in the tree, the height of its children differs by no more than 1. For this problem, assume we have a team of biologists that keep information about DNA sequences in an AVL binary search tree using the specific weight (an integer) of the structure as the key. The biologists routinely ask questions of the type, "Are there any structures in the tree with specific weight between a and b, inclusive?" and they hope to get an answer as soon as possible. Design an efficient algorithm that, given integers a and b, returns true if there exists a key x in the tree such that $a \leq x \leq b$, and false if no such key exists in the tree. Describe your algorithm in pseudocode. What is the time complexity of your algorithm? Explain.

3. [20 points total] Suppose you are acting as a consultant for the Port Authority of a small Pacific Rim nation. They are currently doing a multi-billion-dollar business per year, and their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port. Here is a basic sort of problem they face.

   A ship arrives with n containers of weight $w_1, w_2, ..., w_n$. Standing on the deck is a set of trucks, each of which can hold K units of weight. (You may assume that K and w_i are integers.) You can stack multiple containers in each truck, subject to the weight restrictions of K. The goal is to minimize the number of trucks that are needed to carry all the containers. This problem is NP-complete.

   A greedy algorithm you might use for this is the following. Start with an empty truck and begin piling containers 1,2,3,... onto it until you get to a container that would overflow the weight limit. (These containers might not be sorted by weight.) Now declare this truck "loaded" and send it off. Then continue the process with a fresh truck. By considering trucks one at a time, this algorithm may not achieve the most efficient way to pack the full set of containers into an available collection of trucks.

   (a) [10 points] Give an example of a set of weights and a value for K where this greedy algorithm does not use the minimum number of trucks.

   (b) [10 points] Show that the number of trucks used by this greedy algorithm is within a factor of two of the minimum possible number for any set of weights and any value of K.

4. [20 points total] Suppose you are consulting for a company that manufactures computer equipment and ships it to distributors all over the country. For each of the next n weeks, they have a projected supply s_i of equipment (measured in pounds) that has to be shipped by an air freight carrier. Each week's supply can be carried by one of two air freight companies, A or B.

- Company A charges at a fixed rate r per pound (so it costs rs_i to ship a week's supply s_i).
- Company B makes contracts for a fixed amount c per week, independent of weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

A *schedule* for the computer company is a choice of air freight company (A or B) for each of the n weeks with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks in time. The *cost* of the schedule is the total amount paid to companies A and B, according to the description above.

We seek a way in polynomial-time to take in a sequence of supply values $s_1, ..., s_n$ and return a schedule of minimum cost. For example, suppose $r = 1$, $c = 10$, and the sequence of values is

$$11, 9, 9, 12, 11, 12, 12, 9, 9, 11.$$

Then the optimal schedule would be to chose company A for the first three weeks, company B for the next block of four contiguous weeks, and then company A for the final three weeks. One way to do this is set up the Bellman equation, then let a solver get the answer in polynomial time.

For this problem, derive the Bellman equation, and talk through how you derived it. You must write down your Bellman equation, and you must sufficiently explain your rationale to get credit for this problem.

Programming Assignment Algorithm

The following question asks you to write an algorithm for an upcoming programming assignment. We want you to think ahead about how to solve the problem. You will implement this algorithm (with any corrections) in that upcoming programming assignment.

5. [30 points total] ***Collaborative Problem:*** Programming Assignment 2 (PA2) explores the *Traveling Salesperson Problem (TSP)* in the form of the The Maryland Lighthouse Challenge[1], a biennial event where people race around the state of Maryland attempting to visit all 10 historic lighthouses and one lightship spread out among the most beautiful parts of the state. The winner is the first team that visits all of the lighthouses during the challenge weekend. One of your professors and his wife[2] have competed very favorably[3] in this ;-)

   The lighthouse challenge can be modeled as a variation of the *traveling salesperson problem (TSP)* that recurs frequently in networking, semiconductor layouts,and efficient routing. This problem asks you for a strategy of finding the shortest *Hamiltonian path*, the path and starting point that leads contestants to visit all lighthouses finishing at the earliest possible time. (Unlike traditional TSP, you do not return to the starting lighthouse—this is a straight up race from a start point to a separate finish point.)

   Special note—despite the algorithm taking a starting lighthouse, you are expected to choose the best lighthouse to start at. Assume some caller function, a *kickoff function*, will try all combinations of starting light. See the mention of the phantom below, which is an easy way to find the best starting light automatically.

   If you need a template for writing algorithms, check this out at Overleaf.com, an online LaTeX editor.[4] (Even if you're using Word, you can format at Overleaf and then paste a screenshot.)

   (a) [15 points] Construct a recursive, brute-force algorithm for finding the optimal path among a small set of lighthouses, starting at a given light, and analyze its running time.

   Write a recursive brute-force algorithm in good pseudocode that follows this signature. Note, it must use recursion to operate (that is part of this assignment):

[1]Full URL: https://cheslights.org/maryland-lighthouse-challenge/
[2]Full URL: https://www.facebook.com/cheslights/photos/a.2861267277221280/2995180270496646/
[3]Full URL: https://cheslights.org/1st-finishers-results-2019-md-lighthouse-challenge/
[4]Full URL: https://www.overleaf.com/read/gsjhctdjbbsw

Algorithm 2. Fastest Path

Input: list of lighthouse names L;
starting lighthouse s guaranteed not to be in L;
dictionary of travel times TRAVEL_TIME$[L_i, L_j]$ for every pair of lighthouses in L
Output: ordered list of lighthouses corresponding to the fastest (shortest) possible path starting at s, and travel time

```
1: function FASTEST-PATH($s$, $L$)
2:     my algorithm
3:     return fastest_path, fastest_time
```

(b) [5 points] Contestants may choose to start at any light, and some lights are better to start at than others, because they lead to faster paths. Your programming assignment submission will need to include a kickoff function that tries all possible starting lights. The kickoff function may end up looking a lot like your recursive function, minus the recursive call, leading to redundant code.

It turns out that there's a special trick you can use to simplify the initiation of the algorithm. I call it a *phantom lighthouse*. This is a special lighthouse that is mystically adjacent to every other lighthouse in the path.

Using the phantom lighthouse concept, specify the initial call to Fastest-Path using this phantom. Describe the modifications to the the input data structures needed to support the phantom, and generally explain how the phantom works.

(c) [10 points] Read the paper, An Empirical Study of the Multi-Fragment Tour Construction Algorithm for the Travelling Salesman Problem[5] by El Krari, Ahiod, and El Benani.

Multi-Fragment (MF) is one way to create approximate solutions to TSP. It trades off speed for accuracy: it may find tours that differ by about 5% from the optimal solution, but runs in n^2 time instead of the brute-force time. It works by sorting the edges—travel times, in our case—from least to greatest, then adding edges to the graph one by one until it gets a valid tour.

The algorithm in the paper is a sketch (I call it a construction), because it glosses over one very important condition needed to ensure a valid tour—that all nodes are visited exactly once—and therefore we must avoid creating cycles when adding edges.

Also, note that MF solves the problem for a *tour*, which includes going back to start, but that we are interested in a *path* that does not return to start. In some cases, the shortest/fastest path is **not** a proper subset of the shortest/fastest tour.

You may find something useful in the CLRS book, and if you do, you may call out to those algorithms in your pseudocode rather than having to rewrite them. As an example, say I wanted to use INSERTION-SORT from Chapter 2; I could say "A = [1, 2, 3] ... call INSERTION-SORT(A)" and that is acceptable.

Write the completed Multi-Fragment algorithm using good pseudocode. Assume the presence of a sort function - do not write pseudocode for a sort function. Likewise in the programming assignment, do not code up your own sort routine, use the Python built-in one instead.

[5] Full URL: https://scholar.google.com/scholar?q=An+Empirical+Study+of+the+Multi-Fragment+Tour