CSE 202: Design and Analysis of Algorithms **(Due: 10/26/19)**

Homework #2

Instructor: Ramamohan Paturi *Name:* Shihan Ran, *Netid:* A53313589

Problem 1: Maximum Length Chain of Subwords

Problem Description

Input: We are given a set of n distinct strings of length at most k over a finite alphabet σ.

Output: A sequence of strings that form a chain under the (consecutive) subword relation; i.e., if the output is $w_1, w_2, \dots, w_t$ then we can write $w_{i+1} = uw_iv$ for some strings u, v.

Find a chain of maximum length.

Solution

(High-level description)

We use dynamic programming to solve this problem.

Subproblem: We refer $T(n)$ as the maximum length of chain containing w_n.

Base Case: For a set of only one string, the solution is simply the string itself.

Iteration: We can have the state transition function as follows:

$$T(n) = \max(T(i) + 1),\ \text{where } w_n = uw_iv, 1 \le i < n$$

Since our algorithm should output a sequence of string chain. We need to store some additional values:

1. *maxLength*: the length of the maximum chain
2. *maxIndex*: the tail string's index of the maximum length chain
3. $P(n)$: the index i that maximize $T(n)$

We update *maxLength* and *maxIndex* each time after we computed $T(n)$. With *maxLength*, *maxIndex* and $P(n)$, we can retrace our maximum length chain of subwords following the order

$$[w_{maxindex}, w_{P(maxindex)}, w_{P(P(maxindex))}, \cdots]$$

To find out whether $w_n = uw_iv$, we need to define a new algorithm called *isSubsequence*. We can choose a pattern matching algorithm to do the string comparison. Here, we choose KMP algorithm and the time complexity is proportional to the length of the string, which is O(k).

(Correctness)

Base Case: A set of only one string trivially has only one option for its maximum length chain of subwords.

Induction Hypothesis: Suppose all subproblems are correctly computed up to an arbitrary length of k.

Induction Step: We would like to prove that if the induction hypothesis holds, then our algorithm's "iteration" component correctly computes the subproblem of the length of $k + 1$. The following chain of logic should be sufficient:

1. Per the hypothesis, all the lengths smaller than k are correct.
2. For string w_{k+1}, we have $T(k+1) = \max(T(i) + 1),\ \text{where } w_{k+1} = uw_iv, 1 \le i < k + 1$. We exhaustively check relevant subproblems.
3. We choose the $T(i)$ that yields a max $T(k + 1)$. Therefore, we choose the right option from among our previous calculated values and record the index i as $P(k + 1) = i$.
4. Thus, given that our hypothesis holds, our update correctly computes the length of $k + 1$'s solutions.

Because our base case, hypothesis, and step are correct, our algorithm is proven correct with a proof by induction.

(Time complexity)

The $T(n)$ loop will traverse n strings, which takes $O(n)$ time. At each $T(n)$, we will iterate through $i, 1 \le i < n$ elements, takes $O(n)$ time. When at each w_i, we do a KMP algorithm to find out whether w_i is a subword of w_n, which will take O(k) time, and finding $\max(T(i) + 1)$ will only take $O(1)$.

Overall, our algorithm takes $O(kn^2)$.

Problem 2: Business plan

Problem Description

Consider the following problem. You are designing the business plan for a start-up company. You have identified n possible projects for your company, and for, $1 \leq i \leq n$, let $c_i > 0$ be the minimum capital required to start the project i and $p_i > 0$ be the profit after the project is completed. You also know your initial capital $C_0 > 0$. You want to perform at most k, $1 \leq i \leq n$, projects before the IPO and want to maximize your total capital at the IPO. Your company cannot perform the same project twice.

In other words, you want to pick a list of up to k distinct projects, $i_1, \ldots, i_{k'}$ with $k' \leq k$. Your accumulated capital after completing the project i_j will be $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$. The sequence must satisfy the constraint that you have sufficient capital to start the project i_{j+1} after completing the first j projects, i.e.,$C_j \geq c_{i_{j+1}}$ for each $j = 0, \ldots, k' - 1$.You want to maximize the final amount of capital, $C_{k'}$.

Solution

(High-level description)

We use the greedy algorithm to solve this problem.

According to the problem description, we have the following observations:

1. We can only perform at most k projects before the IPO.

2. We want to maximize our total capital. Each project has a p_i as the profit after the project is completed. Our accumulated capital after completing the project i_j should be $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$.

3. We should have sufficient capital to start the project i_{j+1} after completing the first j projects.

Intuitively, if we are to write a greedy algorithm for a problem like this, we might first consider choosing the projects based on its profit, i.e., we choose the project that has the highest profit each time. However, we should still consider another thing: we should have sufficient capital to start this profitable project. We now formally describe our algorithm:

Sort projects based on its minimum capital required to start the project. When choosing the j-th project, we should choose the most profitable project among all the projects i_{j+1} where $c_{i_{j+1}} \leq C_j$. After chosen k projects, terminate and return the final capital as well as what we've chosen.

(Correctness)

We use the exchange argument to prove the correctness of this algorithm.

Imagine an optimal solution OPT and our greedy solution is G. For ease of comparison, we sort projects in both solutions by increasing profits. For the purpose of comparison, we will consider projects to be equal if they have the same profit and start capital (so duplicate projects are equal to each other). Now for project o_i and g_i, we account for the following two cases:

1. $o_i = g_i$: The solutions agree, so we can move on.

2. $o_i \neq g_i$: We argue that this is not possible. Suppose there exists an optimal solution OPT in which $o_i \neq g_i$. Then, because the greedy algorithm prioritizes projects with higher profit when the start capital is sufficient. And after the most profitable project has been done, the current capital C_i should also be the highest, which will leave more choosing space for the next project. Hence, it is impossible for OPT (or any other solution) to have a higher capital than G. Therefore, this case can only lead to OPT to be worse than G. However, this would contradict our knowledge that OPT is optimal. By proof by contradiction, there is no optimal solution in which $o_i \neq g_i$.

Therefore, we have proven that any optimal solution is necessarily equal to the greedy solution.

(Time complexity)

The initial sort is $O(n \log n)$. Each iteration afterward can be done by $O(\log n)$ if we maintain a max-heap data structure to help us choose the most profitable project under the condition $c_{i_{j+1}} \leq C_j$. And we do that iteration for k times, i.e., this takes $O(k)$ time complexity.

Overall, our algorithm can be completed in $O(n \log n + k \log n) = O(n \log n)$.

Problem 3: Minimum Cost Sum

Problem Description

You are given a sequence $a_1, a_2, \ldots, a_n$ of nonnegative integers, where $n \geq 1$. You are allowed to take any two numbers and add them to produce their sum. However, each such addition has a cost which is equal to the sum. The goal is to the find the sum of all the numbers in the sequence with minimum total cost. Describe an algorithm for finding the sum of the numbers in the sequence with minimum total cost. Argue the correctness of your algorithm.

Solution

(High-level description)

We use the greedy algorithm to solve this problem.

According to the problem description, we observe the fact that the larger number sum, the larger cost to do an addition. Hence, intuitively, if we are to write a greedy algorithm for a problem like this, we might first consider choosing the two numbers based on its sum. We now formally describe our algorithm:

Sort the sequence $A = a_1, a_2, \ldots, a_n$. Then we need to form a Huffman tree with all the numbers are the leaves with the same order as the sorted sequence A'. The key idea is each time, we choose to do the addition among the smallest two numbers. For n numbers, we need to perform addition for $n-1$ times. Hence, at time step i, we will extract two smallest numbers from the Huffman tree, do an addition, then we add their sum to the Huffman tree. After iterating $n-1$ times, the final minimum value of the Huffman tree is the sum of the numbers in the sequence with the minimum total cost.

(Pseudo Code)

Algorithm 1: Minimum Cost Sum

```
Input: the sequence A
Output: sum of the numbers
sort A;
Construct a priority queue Q based on the sorted A;
for i = 1 to length(A) − 1 do
    z = new node;
    z.left = Extract − Min(Q);
    z.right = Extract − Min(Q);
    z.value = z.left.value + z.right.value;
    Insert(Q, z);
return Extract − Min(Q);
```

(Correctness)

With each sorted a_i ordered in the leaves of the Huffman tree, we can simply cite the correctness proof of the Huffman Coding from here.

(Time complexity)

The sorting takes $O(n \log n)$ time. The binary Huffman tree is constructed using a priority queue and each priority queue operation takes time $O(\log n)$. We need to perform addition for $n-1$ times.

Overall, our algorithm takes $O(n \log n)$ time complexity.

Problem 4: Speech recognition

Problem Description

We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. The label of a directed path is defined to be the concatenation of the labels of the edges on that path.

Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = (\sigma_1, \dots, \sigma_k)$ of characters from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of the algorithm. Clearly write any dynamic programming formulation you may use to solve this problem.

Solution

(High-level description)

We use dynamic programming to solve this problem.

We first define $T(i, v_j) = 1$ if there exist a path from v_0 to v_j labeled with the prefix of length i of s, i.e., labeled with $\sigma_1, \dots, \sigma_i$, and $T(i, v_j) = 0$ otherwise. We have $0 \le i \le k$ and $1 \le j \le |V|$. As the description of this problem, our algorithm should return a path in G that begins at v_0 and has s as its label if $T(k, v_j) = 1$, otherwise return NO-SUCH-PATH.

We initialize $T(0, v_0) = 1$ and for other $j \neq 0$, we have $T(0, v_j) = 0$. Our transition function can be the followings:

$$T(i, v_j) = \max \{T(i-1, v_u) | (v_u, v_j) \in E \text{ and } \sigma(v_u, v_j) = \sigma_i\}$$

This transition function means to set the value of $T(i, v_j)$, we should check whether there is an edge (v_u, v_j) labeled with $\sigma(v_u, v_j)$ leading to $T(i, v_j)$ from a vertex v_u that was reached in the previous step. If there is such a vertex v_u and $T(i-1, v_u) = 1$, then we can arrive v_u by a path labeled with the prefix of length $i-1$ of s, i.e., $\sigma_1, \dots, \sigma_{i-1}$. We use the max to ensure that if there exist one path, then $T(i, v_j) = 1$.

Since our output should be the path begins at v_0 and has s as its label. We need to store some additional values:

1. $maxIndex$: the index of the tail vertex v_j in this generated path
2. $P(j)$: the index u that maximize $T(i, v_j)$

We update $maxIndex$ and $P(j)$ each time after we computed $T(i, v_j)$. With $maxIndex$ and $P(j)$, we can retrace our generated path following the order

$$[v_{maxindex}, v_{P(maxindex)}, v_{P(P(maxindex))}, \cdots]$$

(Correctness)

Base Case: $S = (\sigma_1)$, then our algorithm just find if there is an edge (v_0, v_j) labeled with $\sigma(v_0, v_j) = \sigma_1$ leading to $T(1, v_j)$ from a vertex v_0.

Induction Hypothesis: Suppose all subproblems are correctly computed up to an arbitrary length of n.

Induction Step: We would like to prove that if the induction hypothesis holds, then our algorithm's "iteration" component correctly computes the subproblem of the length of $n+1$. The following chain of logic should be sufficient:

1. Per the hypothesis, all the lengths smaller than n are correct.
2. For σ_{n+1}, we have $T(n+1, v_j) = \max(T(n, v_u)) | (v_u, v_j) \in E$ and $\sigma(v_u, v_j) = \sigma_{n+1}$. We exhaustively check relevant subproblems.
3. We choose the $T(n, v_u)$ that yields a max $T(n+1, v_j)$. Therefore, we choose the right option from among our previous calculated values and record the vertex index j as $P(n+1) = j$.
4. Thus, given that our hypothesis holds, our update correctly computes the length of $n+1$'s solutions.

Because our base case, hypothesis, and step are correct, our algorithm is proven correct with a proof by induction.

(Time complexity)

The $T(i, v_j)$ loop will traverse from $\sigma_1, \ldots, \sigma_k$, which takes $O(k)$ time. At each $T(i, v_j)$, we will iterate through $|V|$ vertexes, takes $O(|V|)$ time. When at each v_j, we will check if $(v_u, v_j) \in E$ and $\sigma(v_u, v_j) = \sigma_{n+1}$ for each v_u, which takes $O(|V|)$ time.

Overall, our algorithm takes $O(k|V|^2)$ time.