CSE 101: Homework #6

Due on May 30, 2024 at 23:59pm

Professor Jones

Ray Tsai
A16848188

Problem 1

Let G be an directed acyclic graph with vertex set V and edge set E.

Design a (recursive) backtracking algorithm that counts the number of topological orderings of the vertices in V.

Algorithm Description:

First initialize set T which would store the resulting orderings. If $|V| = 0$, return an empty set. Construct a list in such that $in[v]$ is the indegree of vertex v in G, for all $v \in V$. For each $v \in V$ with $in[v] = 0$, recurse on the graph $G - \{v\}$ to obtain a set of topological orderings T_v. Then, for each ordering S in T_v, add $[v] + S$ to T. Return T after we have iterated through all possible v's.

Proof of Correctness:

Proof. Let $Top(G)$ denote the result of the algorithm after running on graph G, and let $\mathcal{T}(G)$ denote the set of all topological orderings of G. We proceed by induction on n to show that $Top(H) = \mathcal{T}(H)$, for all subgraph $H \subseteq G$ with n vertices.

The base case is trivial, as $Top(H)$ is the empty list when H has no vertices, which is obviously $\mathcal{T}(H)$.

Suppose $n \geq 1$. Assume that for all subgraph $H' \subseteq G$ with less than n vertices, $Top(H') = \mathcal{T}(H')$. Let H be a subgraph of G with n vertices.

For all $L \in Top(H)$, we have $L = v + L'$, where v is an vertex of H and $L' \in Top(H - \{v\})$. By induction, $L' \in \mathcal{T}(H - \{v\})$, and thus $L \in \mathcal{T}(H)$, as v has indegree 0 in H.

Now suppose $S \in \mathcal{T}(H)$. S starts with a vertex $u \in H$ with indegree 0 in H. By induction, $S - [u] \in \mathcal{T}(H - \{u\}) = Top(H - \{u\})$. Since $in[u] = 0$ in H, the outer loop in the algorithm runs through u, which then includes $S = [u] + (S - [u])$ in the outputing set during the inner loop. Hence, $S \in Top(G)$.

It now follows that $\mathcal{T}(H) = Top(H)$, and this completes the induction. Obviously, G is a subgraph of G, and hence the result. □

Problem 2

Recall the electric car problem but this time, each battery also has a price $p[i]$ to replace:

> Suppose you are driving along a road in an electric car. The battery of the electric car can bring you $x[0]$ miles. There are battery stations along the way at positive positions $D[1], \dots D[n]$ (in sorted order.) Each battery station can *replace* your battery and give you a new battery that can bring you a certain number of miles. The distances of the batteries are given in the array $x[0], x[1], \dots, x[n-1]$ and the price of each battery to replace is $p[1], \dots, p[n-1]$.
>
> You wish to start at position 0 with a full battery and end at position $D[n]$ by replacing batteries with the minimum total cost.

(a) Recall the following greedy algorithm that worked in homework 4:

> **Candidate Greedy Strategy III:**
>
> Travel to the battery station with the largest $D[i] + x[i]$ value (in other words, the battery that can take you the farthest down the road.) Replace the battery at that station and repeat the process starting from that station until you can reach position $D[n]$ and then go directly there.

Give a counterexample as to why this does not always give you the optimal solution.

Proof. Consider the input $D = [0, 1, 2, 3], x = [2, 2, 2, 2], P = [0, 0, 1, 0]$. Since station 2 has the highest $D[i] + x[i]$ value $(2+2 = 4)$ among all reachable stations from the start, the strategy picks 2, which costs 1 dollar. But then we may change battery at station 1 then go directly to the destination, which costs nothing. □

(b) Design a *reduction* algorithm that uses Dijkstra's algorithm. Compute the time analysis in terms of the number of battery stations n.

Algorithm Description:

Construct a weighted directed graph $G = (V, E, w)$, with $V = \{0, 1, \dots, n\}$ being the set of all stations. For distinct vertex $u, v \in V$, we add an edge (u, v) if v is reachable from u. For each edge $(u, v) \in E$, assign weight $w(u, v) = p[v]$ if $v \neq n$, and put $w(u, v) = 0$ if $v = n$. We now run Dijkstra's algorithm on G starting on vertex 0. Let P be the resulting path which ends on vertex n and return the vertex set of P.

Runtime Analysis:

G has $O(n)$ vertices and $O(n^2)$ edges, so constructing G takes $O(n^2)$ time. Running Dijkstra on G using array as a priority queue takes $O(n^2)$ time. Since Dijkstra also yields the cheapest path to a given vertex, we may obtain path P by tracing back the precding vertices starting from vertex n, which takes $O(n)$ time. In total, the algorithm runs in $O(n^2)$ time.

Problem 2 continued on next page...

(c) Design a DP tabulation algorithm by ordering the subproblems from n to 0: (step 1 and 4 have been done for you)

1: Define the subproblems:

Let $G[k]$ be defined to be the minimum price it takes to get to battery station n assuming you start at battery station k with the battery from battery station k.

2: Define and evaluate the base cases

$G[n] = 0$.

3: Establish the recurrence for the tabulation.

Suppose $0 \leq k < n$. Let $S = \{i \in [n] \mid D[k] < D[i] \leq D[k] + x[k]\}$, which is the set of all reachable stations from station k. Then,

$$G[k] = \min_{i \in S} G[i] + P[k].$$

4: Determine the order of subproblems:

Order the subproblems from n to 0.

5: Final form of output.

$G[0]$.

6: Put it all together as pseudocode

1. Initialize array G of size $n + 1$
2. $G[n] = 0$
3. **for** k from $n - 1$ to 0
4. **for** i from $k + 1$ to n
5. **if** $D[i] \leq D[k] + x[k]$
6. $G[k] = \min(G[k], G[i] + P[k])$
7. **return** $G[0]$

7: Runtime analysis

Each iteration of the inner loop takes constant time. Since the outer loop iterates through $0 \leq k \leq n - 1$ and the inner loop iterates through $k + 1 \leq i \leq n$, it takes

$$\begin{aligned} O\left(\sum_{k=0}^{n-1} \sum_{i=k+1}^{n} c\right) &= O\left(c \sum_{k=0}^{n-1} (n - k)\right) \\ &= O\left(n^2 - \frac{n(n-1)}{2}\right) \\ &= O(n^2) \end{aligned}$$

time.

Problem 3

In the kingdom of Dynamoprogamia, a law was passed that the currency used in the Kingdom would be in the denominations of perfect square numbers (i.e. in denominations of 1, 4, 9, 16, 25 and so on). You can assume the denominations can go as large as you want it to go as long as it's a perfect square number. Let's call this currency Square-Dollars. A money lender in this Kingdom wants to lend money to his customers in such a way that he gives them the least number of coins possible.

(a) The greedy strategy for this: "Pick the maximum denomination that's less than *or equal to* the amount you need to lend and subtract it from the amount you need to lend. Continue this process until you're left with nothing else to lend." Provide a counter example for this.

Proof. Consider the case 18. The algorithm picks 16 first, which leaves us $18 - 16 = 2$. But then 2 is not a square number, so in total this strategy uses more than 2 coins to lend 18 Square-Dollars. But then 18 can be lent with two 9-dollar coins. □

(b) Devise a Dynamic programming-based algorithm that returns the minimum number of coins needed to make change for n Square-Dollars. (your algorithm should run in $O(n^{1.5})$ time.)

Algorithm Description:

Initialize a list $[A[1], \ldots, A[n]]$ such that $A[i] = \begin{cases} 1 & \text{if } \sqrt{i} \in \mathbb{N} \\ i & \text{otherwise} \end{cases}$, for all $1 \leq i \leq n$. For each positive integer $i \leq n$, loop through each square number $s \leq i$, and make $A[i] = \min(A[i-s]+1, A[i])$. At the end of the algorithm, return $A[n]$.

Proof of Correctness:

Proof. Let $\delta(n)$ denote the minimum number of coins needed to lend n Square-Dollars. We proceed by induction on n to show that $A[n] = \delta(n)$.

Suppose $n = 1$. Since 1 is a square number, $A[1] = 1$, which is obviously equal to $\delta(1)$.

Now suppose $n \geq 2$. Assume that $A[i] = \delta(i)$ for all $i < n$. We may also assume that n is not a square number, otherwise $A[n] = 1 = \delta(n)$ and we are done. Hence, we have $\delta(n) > 1$. Let $c_1, \ldots, c_{\delta(n)}$ such that each c_j is a square number and $c_1 + \cdots + c_{\delta(n)} = n$. By induction, $A[n - c_{\delta(n)}] = \delta(n - c_{\delta(n)})$. But then note that $\delta(n) \geq \delta(n - c_{\delta(n)}) + 1$, otherwise $c_1, \cdots, c_{\delta(n)-1}$ sums up to $n - c_{\delta(n)}$ with less than $\delta(n - c_{\delta(n)})$ coins, contradiction. Since $A[n]$ is the minimum of $A[n-s]$ for all square number s, we have $A[n] \leq A[n - c_1] + 1$. Combining all inequalities, we get

$$A[n] \leq A[n - c_1] + 1 = \delta(n - c_1) + 1 \leq \delta(n) \leq A[n],$$

which yields $A[n] = \delta(n)$. □

Runtime Analysis:

Proof. Initializing A takes $O(n)$ time. Note that there are at most $\sqrt{i}$ square number $s \leq i$, for all $i \in \mathbb{Z}^+$. Hence, for each positive integer $i \leq n$, looping through all square number takes $O(\sqrt{n})$ time, and thus the outer loop which iterates through all $i \leq n$ takes $O(n^{1.5})$ time. In total, the algorithm takes $O(n^{1.5} + n) = O(n^{1.5})$ time. □

Problem 4

You are given an $n \times n$ matrix of 0's and 1's: $(A_{i,j})_{1 \le i \le n,\ 1 \le j \le n}$

Design a DP tabulation algorithm that finds the length of a side of the largest square entirely made of 1's: (step 1 has been done for you)

1: Define the subproblems:

Let $S[i, j]$ be defined to be the length of the side of the largest square entirely made of 1's with bottom right corner at entry $[i, j]$

2: Define and evaluate the base cases

When i or j is 1,

$$S[i, j] = A_{i,j}.$$

3: Establish the recurrence for the tabulation.

Suppose $i, j \ge 2$.

$$S[i, j] = A_{i,j}[\min(S[i-1, j-1], S[i-1, j], S[i, j-1]) + 1].$$

4: Determine the order of subproblems:

Order the subproblems in row-major order. That is, we are traversing through S one row at a time, from left to right, and top to bottom. Each row is completed before moving on to the next row.

5: Final form of output.

$\max_{[i,j]} S[i, j]$.

6: Put it all together as pseudocode

1. Initialize two dimensional array S of size $n \times n$ (with 1-based indexing)
2. $maxLen = 0$
3. **for** i from 1 to n
4. **for** j from 1 to n
5. **if** $i = 1$ or $j = 1$
6. $S[i][j] = A_{i,j}$
7. **else**
8. $S[i][j] = A_{i,j}[\min(S[i-1, j-1], S[i-1, j], S[i, j-1]) + 1]$
9. $maxLen = \max(maxLen, S[i][j])$.
10. **return** $maxLen$.

7: Runtime analysis

The algorithm is simply a nested loop iterating through each entry of the $n \times n$ array, with each iteration running in constant time. Hence, the total runtime is $O(n^2)$.