1.5em 0pt

Solutions for Problem Set 5

Name: Ajeya Kempegowda

1. (6 points) Making change

You are given unlimited quantities of coins of each of the denominations $d_1, d_2, \dots, d_m$ (all positive integers) and a positive integer n. You are asked to find the smallest number of coins that add up to n, or indicate that the problem has no solution.

Design a dynamic programming algorithm to solve this problem. Justify its correctness and analyze its asymptotic worst-case running time.

Answer:

Algorithm 1 Smallest number of coins

```
procedure MIN COINS($coins[d_1, d_2, \dots, d_m], n$)
    if $n == 0$ then                                              ▷ base case
        return 0
    else
        Define global array C, to store optimal solutions for the smaller problems.
        Initialize all elements of C to $\infty$.
        for $i \leftarrow 1 : n + 1$ do                            ▷ Iterate for all values of n
            for $j \leftarrow length(coins)$ do
                if $coins[j] \leq i$ then                          ▷ select coins smaller than i
                    $C[n] \leftarrow min(C[n - coins[j]] + 1, C[n])$    ▷ add 1 to solution of (amount-coin value)
                end if
            end for
        end for
        if $C[n] == \infty$ then
            return no solution
        else
            return C[n]
        end if
    end if
end procedure
```

Running time:

Since we have two nested for loops which iterates through the length of the coins and the inner loop iterates to the required value, the time complexity of the algorithm in $O(mn)$, where m is the length of the coins we have.

The above algorithm exhibits an optimal substructure. Consider any optimal solution to making change for n value using coins of denominations $d_1, d_2, \cdots d_k$. Now consider breaking that solution into two different pieces along any coin boundary. Suppose that the left-half of the solution amounts to b value and the right-half of the solution amounts to *(n - b)* value

By contradiction, suppose that there was a better solution to making the left-half of the optimal solution . Then the left-half of the optimal solution could be replaced with this better solution, yielding a valid solution to making change for n cents with fewer coins . But this contradicts the supposed optimality of the given solution. An identical argument applies to the right-half of the solution. Thus, the optimal solution to the coin changing problem is composed of optimal solutions to smaller sub problems.

Let C[p] be the minimum number of coins of denominations $d_1, d_2, \cdots d_k$ needed to make change for p cents. In the optimal solution to making change for p cents, there must exist some first coin d_i, where $d_i \leq p$. Furthermore, the remaining coins in the optimal solution must themselves be the optimal solution to making change for pd_i cents, since coin changing exhibits optimal substructure as proven above. Thus, if d_i is the first coin in the optimal solution to making change for p cents, then $C[p] = 1 + C[p - d_i]$.

This has been clearly incorporated in the algorithm (line 10) and the base cases have been handled(line 2). After we're iterated through length of the coin denominations we have if we haven't met the target yet, we'd be returning a no solution else the correct solution would be returned from the nth position from the global array C.

Problem 2. (7 points) Overnight Stops on a Road Trip

You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You *must* stop at the final hotel (at distance a_n), which is your destination.
You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty – that is, the sum, over all travel days, of the daily penalties.

Give an efficient algorithm that determines the optimal sequence of hotels at which to stop. Prove the correctness of your algorithm and analyze its asymptotic worst-case running time.

Answer:

Let OPT(i) be the minimum total penalty to get to hotel i.

To get OPT(i), we need to consider all hotels j we can stay at the night before reaching hotel i. For each of these possibilities, the minimum penalty to reach i is the sum of the minimum penalty OPT(i) to reach i + and the cost of a one-day trip from j to i.

$$OPT(i) = min_{0 \leq j < i}\{OPT(j) + (200 - (a_j - a_i)^2)\} \tag{1}$$

Algorithm 2 Overnight Stops

```
1: procedure STOPS($a[a_1, a_2, \ldots, a_n], n$)
2:     if $n == 0$ then                                          ▷ base case
3:         return 0
4:     else
5:         for $i \leftarrow 1 \cdots n$ do                            ▷ n subproblems
6:             for $j \leftarrow 0 \cdots i - 1$ do
7:                 $OPT[i] = min\{OPT[j] + (200 - (a_j - a_i))^2\}$          ▷ O(i)
8:             end for
9:         end for
10:        return OPT[n]
11:    end if
12: end procedure
```

The overall complexity is :

$$\sum_{n=1}^{i} O(i) \tag{2}$$

we know,

$$\sum_{n=1}^{i} f(n) = 1 + 2 + 3 + \cdots n = \frac{n(n+1)}{2} \tag{3}$$

Since, we are adding first n-1 hotels, but the first (N-1) numbers. So N becomes (N-1), and (N+1) becomes N, which gives us

$$\sum_{n=1}^{i} O(i) = 1 + 2 + 3 + \cdots (n-1) = \frac{n(n-1)}{2} \tag{4}$$

Therefore, the time complexity of the algorithm would be $O(n^2)$ as per equation (2)

Further, as per equation 1, the formula to stop at optimal hotels while reducing the penalties throughout the travel days has been properly incorporated in the algorithm (line 7) while the base cases have been handled in line 2 3. As aforementioned, if our sub-problems are optimal then the entire solution constructed by these sub-optimal problems are also optimal. Here, from lines 5 to 7 we're iteratively finding the these optimal solutions. Hence, the correctness follows.

Problem 3. (7 points) Shortest paths using at most k edges

Given a directed graph $G = (V; E)$ with positive integer weights on the edges, a source $s \in V$, a destination $t \in V$, and an integer k, design an efficient algorithm to determine the shortest-weight path from s to t containing at most k edges; if there is no path from s to t using at most k edges, your algorithm must indicate so.

Prove the correctness of your algorithm and analyze its worst-case time complexity.

Answer:

From any given vertex j, from j = 1 to k, we can find the distance to any vertex u using exactly j edges as for all edges (x, u) where x is a predecessor of u with an edge from x to u.

$$D[j, u] = min\{D[j-1, x] + l_e\} \tag{5}$$

If there are no incoming edges then $D[m, u] = \infty$

This will result in k cross V matrix which will keep track of all shortest path weights using exactly j edges from s at iteration j

Algorithm 3 Shortest paths using at most k edges

```
 1: procedure MIN-PATHS-K-EDGES($G, s, t, l_e, k$)
 2:     let $u\epsilon V$ and $j\epsilon k$                                ▷ Initialization
 3:     total paths = [empty list]
 4:     ∀ vertices u except s
 5:     $D[u, k] = \infty$
 6:     for $j \leftarrow \cdots k$ do                                    ▷ O(k)
 7:         for  every edge($x, u$) do
 8:             if $D[j-1, x] + l_{xu}) < D[j, u]$ then
 9:                 $D[j, u] = D[j-1, x] + l_{xu}$
10:             end if
11:         end for
12:     end for
13:     paths[j, u] = x
14:     if $D[k, t] = \infty$ then           ▷ failure case if value is not updated from infinity
15:         return no paths found
16:     else
17:         for $j \leftarrow k \cdots 0$ do
18:             total paths.add(paths[j,t])                   ▷ store the traversed paths
19:             $t \leftarrow paths[j, t]$
20:         end for
21:     end if
22:     return total paths
23: end procedure
```

Running time :

Since we visit $|E|$ edges for each number of the required edges (k) in $|V|$, the time complexity is $O(k|E|+|V|\ )$.

Proof of correctness:

For the base case of induction, consider the case before iterations, where $D[u, k] = \infty$. This is correct as the source distance (u) is 0 i.e there is no path from u with 0 edges.

For the inductive case, Consider a moment when a vertex's distance is updated by distance(j) := distance(u) + weight(xu). By inductive assumption, distance(j) is the length of some path from source to j. Then distance(j) + weight(xu) is the length of the path from source to u that follows the path from source to j and then goes to u.

Now, consider a shortest path P from source to u with at most k edges. Let x be a vertex before u on this path. Then, the part of the path from source to x is a shortest path from source to v with at most k-1 edges, else then there must be some shorter path from source to x with at most k-1 edges. Now, we could then append the edge xu to this path to obtain a path with at most k edges . By inductive assumption, distance(x) after k-1 iterations is at most the length of this path from source to x. Therefore, weight(xu) + distance(x) is at most the length of P. In the kth iteration, distance(u) gets compared with weight(xu) + distance(x), and is set equal to it if weight(xu) + distance(x) is smaller. Therefore, after k iterations, distance(u) is at most the length of P, i.e., the length of the shortest path from source to u that uses at most k edges.