Assignment 5

CSCI 270

Paul Kim | pbkim@usc.edu | 1723717002

Question 1

(1) [10 points]
Imagine that you are going on a road trip. You start from position 1, and will end up at position n. At each integer position along the road (so positions $\{1, 2, \ldots, n\}$), there is a gas station. For each gas station i, you are given a price per gallon $p_i \geq 0$. One gallon of gas is exactly enough to make it one position on the line. Your car has a gas tank size of $s \leq n$ gallons, which starts out empty with you at the gas station at position 1.

You can stop at as many gas stations as you want. Every time you stop, you can decide how many gallons to put in your tank, though the total in your tank can never exceed s. The amount of gas you buy at any station will always be an integer. Your goal is to compute the minimum amount of money you can spend to get to location n. (It's ok if your tank runs empty right as you reach it.)

Give and analyze (prove correct and analyze running time) a polynomial-time algorithm for solving this problem.

Answer:

***Goal: Compute the minimum amount of money you have to spend to get to location n (in the cheapest fashion). If the problem asked for a minimum number of gas station stops, then we could use a greedy algorithm, but we want to find a set of gas stations from i to n, that yields the cheapest price.**

**Informally, we have a car with a tank capacity. Assume that at each linearly ordered vertex, gas is purchased at a certain price. Find the cheapest route to go from gas station i to gas station n.*

Dynamic Programming

We define OPT(i,g) = The minimum cost of arriving at gas station i, with g units of gas remaining in the tank.

Recurrence Relation: OPT(i,g) = min(OPT(i,g-1)+P[i], OPT(i-1, g+1)), with a few edge cases.

***Very similar to algorithm on Midterm where we have to find the optimal path (in a way).*

Observation: For station i >= 2, to get to the ith gas station with g gallons left in the tank, there are two things we could've done to get to that station with some j gallon in the tank. Either we could've bought gallons of gas at the ith station, or exhaust one remaining mile of gas from the previous station since every position is 1 mile away. To optimize, we want to take whichever is lesser.

Formal Algorithm:

P[1,...,n] = price per gallon at gas station i to n.

int A[n][s]; // N positions and S gas spaces.

A[1][0] = 0; // Base Case (we start with 0 gas at position 1, initially)

for (int i = 1; i <= s; i++)

    A[1][i] = P[1]*i; // We want to initialize the first column of the 2D array at the station at position 1.

for (int i = 2; i <= n; i++)

    for (int j = 0; j <= s; j++)

        if (j == S) // To prevent going out of bounds

            A[i][j] = A[i][j-1]+P[i];

        else if (j == 0) // To prevent going out of bounds

            A[i][j] = A[i-1][j+1];

        else // Recurrence that calculates the OPT solution

            A[i][j] = min(A[i][j-1]+P[i], A[i-1][j+1]);

return A[n][0]; // This will be the cheapest value since we want the solution with no leftover gas.

Runtime: O(N^2) since there are two nested-for loops. The first one iterates through the entire N stations. The inner loop and calculate each index position of the 2D array S times. However, in the prompt, the gas tank size is capped by the number of gas stations, so the worst-case runtime of O(N^2) is correct.

// Very Similar to Knapsack and Sequence Alignment Proof

Correctness Statement (Inductive Hypothesis): A[i][g] = OPT(i,g) for all i,g. Then the final answer at A[n][0] = OPT(n,0) where n is the last gas station is the correct solution for the problem. *I use g, j, and s interchangeably. They all mean the same thing (capacity of gas tank).

Proof of Correctness: Induction

We define OPT(i,g) = The minimum cost of arriving at gas station i, with g units of gas remaining in the tank.

Prove by induction on i.

Base Case: At gas station i = 1, we are given an empty tank, so we set OPT (1,0) = 0.

Inductive Step: Consider i+j > 0

    Case 1: If j == S, A[i][j] = A[i][j-1]+P[i] = OPT(i,j)

    Case 2: If j == 0, A[i][j] = A[i-1][j+1] = OPT (i,j)

    Case 3: Else,

A[i][j] = min(A[i][j-1]+P[i], A[i-1][j+1]); // (Algorithm Assignment)

A[i][j] = min(OPT[i][j-1]+P[i], OPT[i-1][j+1]); // (By the inductive hypothesis)

A[i][j] = OPT(i,j) // (Recurrence Relation we worked out)

Since by the inductive hypothesis, our algorithm constructs the correct 2D array, the solution at A[n][0] should be the cheapest way to get to station N, because it must be so that it is cheapest when we have no leftover gas.

Dynamic Proof Completed;

Question 2

(2) [10 points]
Imagine that you play a very simple board game in the style of "Chutes & Ladders", based on repeatedly rolling dice. In this game, there are n squares, numbered from 1 to n. Each square is either a normal square or a trap — this is given to you by an array of Boolean values $t[1, \ldots, n]$. Neither square 1 nor square n is a trap. You start at square 1. In each round, you roll a d-sided die (with $d \geq 1$), which returns a uniformly random number in $\{1, 2, \ldots, d\}$. Then, you advance that number of steps. If you land[1] on a square that is a trap, you immediately lose. Otherwise, you continue on the next turn with another die roll. If you reach square n, by landing on it or walking over it, you win.

Give and analyze (prove correct and analyze running time) a polynomial-time algorithm for computing the probability that you win. In your analysis, you can assume[2] that all arithmetic on fractions takes constant time.

Answer:

**Imagine a size N linear game board, with n squares, numbered from 1 to n, with a repeatedly rolling dice of side d, that returns a random number from (1,2,...d). Each square is either a normal square of a trap given as Boolean values t[1,...,n].*

**Assume: Square 1 and square N is a normal square. If you land on a square that is a trap, you lose. Otherwise, continue on the next turn with another die roll. If you reach square n, by landing on, or walking over it, you win.*

**Goal: Compute the probability that you will win.*

Dynamic Programming

Observations: Our OPT isn't really optimizing anything.

Let P(i) be the probability of reaching square "i" or past it, alive. We pretty much just loop through the array, and compute the possible ways you could've reached that square i.

**To get the probability of reaching i from either 1 or 2 squares before is their sum. So to get the probability of reaching i, you would add the probability of the terms terms for 1 before, 2 before, all the way up to d before., since you could've have rolled any number 1-d.*

**The die is D sided. which means you can roll D different numbers so there can only be D different*

ways to reach any square right (in one roll). So we just do (i-1), then (i-2), then (i-3) until we have D terms total, or until we've exhausted all the squares.

Recurrence: We can express the recurrence as P(i) = P(i-1)*1/d + P(i-2)*1/d + P(i-3)*1/d... for D terms or till there are no more squares.

Formal Algorithm:

*d = D sided dice.

T[i] = Array of Boolean that indicate trap or valid squares.

P[i] = Solution/Probability of landing on the ith square.

P[0] = 1; (100 percent success rate) (Base Case of the first square)

```
for (int i = 1; i < n; i++)
    if T[i] == true
        P[i] = 0; // Probability of Success is 0 thereafter.
    for (int j = 1; j <= d; j++)
        if (P[i] == 0)
            break;
        if (i-j >= 0)
            if P[i] has no value
                P[i] = P[i-j]*1/d;
            else
                P[i] += P[i-j]*1/d;
```

// We want to capture if we are at the last position(s), and we have the possibility to go beyond the last square because it still counts as a success. *N is the last square position.

```
    for (int t = 1; t < d; t++)
        P[N] += P[N-t]*1/d*(d-t);
```

Return P[n]; // Probability of Reaching the Last Target (Goal)

Runtime: O(N * Min(D, N)) since there are two nested-for loops. The first one iterates through N linear squares. The inner loop doesn't always run for d iterations if D > N (because of the break), so it iterates either D or N iterations, whichever is smaller.

Proof of Correctness: Induction

We define SOL(i) to be the probability of landing on, or passing over the last square successfully. For the sake of clearness, I refer to P(i) as SOL(i). P[i] is the array used in the algorithm. *Note the array and parenthesis difference.

Correctness Statement: prove that for all i, P[i] = SOL(i), then in the end, P[n] = SOL(N), where this yields the probability of success landing on, or moving over the last square.

Prove this by Induction on i.

Base Case (i = 0): P[0] = 1 = SOL(0). the first square is always valid with a 100 percent success rate.

**We will call it SOL rather than OPT, because we aren't really optimizing.*

Inductive Step (for some value i > 0):

We express the recurrence as P(i) = P(i-1)*1/d + P(i-2)*1/d + P(i-3)*1/d... for D(dice) terms or until there are no more squares left. This recurrence/inner for-loop uses the smaller sub problems (using the probability of landing on the previous D squares), and sums the probability of landing on those, to get the probability of reaching i. Our algorithm is a dynamic programming algorithm, where we calculate the solutions for the smaller instances, to formulate the solution for the bigger instance (the last square). The last square contains the optimal solutions for the smaller instances as sub-solutions.

**To get the probability of reaching i from either 1 or 2,...D squares before is their sum. So to get the probability of reaching i, you would add those terms for 1 square before, 2 squares before, all the way up to d squares before, since you could have rolled any number 1 through d previously, to get to that spot.*

P[i] = The sum of P[i-j]*1/d, (*Where j are the previous values/probabilities from 1 <= d where i-j >= 0, and d = the sides of the dice.) (*What our algorithm does)

P[i] = The sum of SOL(i-j)*1/d (*By the inductive hypothesis)

P[i] = SOL(i) (*What we derived on OPT)

**Explained what the code does, and why it is true. (Calculating the probabilities of all previous squares, summing those probabilities, to calculate new probabilities for further squares to the right).*

**The last part of the algorithm also holds true by what it is doing. Basically, every roll from the last (d-1) positions of the array may potentially roll pass the last square which counts as a successful roll. So what we need to/are doing is counting these final rolls that pass the last square as a successful roll and summing these possibilities as well to add to the final probability of success.*

Dynamic Proof Completed;