CSCI270 Homework 6

Jacob Ma

October 27, 2022

Problem 1

Climate change has been affecting California heavily — as you can see almost daily, there are more and more severe wildfires and other consequences from year to year. The severity of wildfires is closely related with the amount of rain (or lack thereof) that has occurred in the preceding year, so scientists are trying to calculate the probability that 2023 will be an even dryer year than 2022.

Specifically, they posit the following model. There are n days that they are interested in, and for each day i, their meteorological model gives them a probability $p_i \in [0, 1]$ that it rains on day i. For simplicity, they assume that rain on different days happens independently.[a] They are also given the number k of drought days in 2022. Now, they want to calculate the probability that there will be more than k drought days in 2023.

Give (and analyze, of course) an algorithm with running time $O(n^2)$ which correctly computes the probability that more than k days will be drought days in 2023. You may assume that arithmetic operations take time $O(1)$, even when they involve numbers that are the products of up to n probabilities.[b] You should give a full algorithm and proof here.

[a] Check your probability notes if you don't remember what that term means and implies.

[b] In reality, such numbers would likely need n decimal digits, and hence require time $\Theta(n)$ for arithmetic operations. You get to ignore this subtlety.

Solution

Algorithm as follows:

```
int computDroughtProbability(int n, int k, double[] p){
    // n := number of days we are interested in a year
    // k := threshold of number of drought days
    // p := array of size n+1 indicating the probability of rain during day 1 to day n, p[0] not used
    if (k > n){ return 0; } // Edge Case
    int[][] dp = new dp[n+1][n+1];
    define all dp[i][j]=0
    dp[0][0] = 1;

    for (int i = 1; i < n+1; i++){
        dp[i][0] = dp[i-1][0] * p[i];
        for (int j = 1; j <= min(i,n); j++){ // At most i drought days out of i days
            dp[i][j] = dp[i-1][j-1]*(1-p[i]) + dp[i-1][j]*p[i];
```

```
            }
        }

        int sum=0;
        for (int i = 1; i < n+1; i++){
            for (int j = k; j < n+1; j++){
                sum += dp[i][j];
            }
        }
        return sum;
    }
```

Correctness Proof. Let's define our probability

$$P(i,j) := \text{Exactly } j \text{ drought days in the first } i \text{ days}$$

And since raining on each day is independent, we know for a certain day i^{th}, it could either rain or drought. Thus, if in the first i days we have exactly j drought days, it could only be achieved by the following two conditions:

(1) The i^{th} day is drought, and we used to have exactly $j-1$ drought days before the i^{th} day.

(2) The i^{th} day is not drought, which means it is rainy, and we already have exactly j drought days before the i^{th} day.

Since the rainy event on each day is independent, and these two situations partitions the universal event, we would have the following recurrence relationship for $P(i,j)$:

$$P(i,j) = \underbrace{P(i-1,j-1)\cdot(1-p_i)}_{\text{Drough on day } i^{th}} + \underbrace{P(i-1,j)\,p_i}_{\text{Rain on day } i^{th}}$$

Based on this recurrence relationship, we claim that $dp[i][j] = P(i,j)$ in the algorithm above, and prove by induction on $i+j$.

Base Case: $i+j=0$, the only possibility is $i=j=0$. $dp[0][0]=1$, and there is indeed $P(0,0)$ to have 0 drought days during 0 days.

Induction Hypothesis: $dp[a][b] = P(a,b)$ for all a,b whenever $a+b<i+j$. [Strong Induction]

Induction Step:

(1) **Case 1:** $i=0$. We are considering having $j>0$ drought days within 0 days, which is impossible. $dp[i][j] = P(i,j) = 0$. Thus, holds.

(2) **Case 2:** $j=0$. The probability should be

$$P(i,0) = \prod_{j=1}^{i} p_i$$

This is computed by $dp[i][0] = dp[i-1][0]\cdot p_i$, which is multiplying all p_i. Thus $dp[i][0] = P(i,0)$, holds.

(3) **Case 3:** $j > i, \quad i, j > 0$. Since $dp\,[i]\,[j]$ will not be changed if $j > i$ since we only reach $j = \min\,(i, n)$ every time, it will stay 0 since initialized. And $P\,(i, j) = 0$ if $j > i$ since we cannot have number of drought days greater than the total number of days.

(4) **Case 4:** $j \leqslant i, \quad i, j > 0$. Based on the algorithm

$$\begin{aligned} dp[i][j] &= dp[i-1][j-1] * (1 - p[i]) + dp[i-1][j] * p[i] \\ &= P\,(i-1, j-1)\,(1 - p_i) + P\,(i-1, j) \cdot p_i \qquad \text{Induction Hypothesis} \\ &= P\,(i, j) \qquad \text{Recurrence Relationship} \end{aligned}$$

Thus, holds.

Thus, we know $dp\,[i]\,[j] = P\,(i, j)$.

Final Step: We also know that the probability of having more than k drought days is partitioned by all $P\,(i, j)$ where $j \geqslant k, \quad j \in \mathbb{N}$, thus the final answer could be get by adding all these up.

The conclusion follows the induction.

Runtime Analysis. Line 5 takes constant time. Line 6 to 8 initialized a 2D array with size $n+1 \times n+1$, thus takes $O\left((n+1)^2\right) = O\left(n^2\right)$ time.

From line 10 to 15 is a nested for loop takes at most n^2 iterations. In the first nested loop, there is $n+1$ iterations, so the operation on 11 will take $O\,(1) \cdot O\,(n+1) = O\,(n)$ in total. This is dominated by the $O\,(1)$ operation on line 13, which takes n^2 iterations, which has $O\,(1) \cdot O\left(n^2\right) = O\left(n^2\right)$ runtime.

The summation section from line 17 to line 23 has at most $(n+1)^2$ iterations, which includes a constant time operation. So this section have $O\left(n^2\right)$ runtime.

Thus, the whole algorithm will take $O\left(n^2\right)$ in total. □

Problem 2

Consider the problem of learning what is going on inside a system (black box) when you cannot observe the inside, and only see some kind of output that gives you partial information about the inside. We model this as follows.

There is a known directed graph $G = (V, E)$, with a known start node $s \in V$. Associated with each node $v \in V$ is a probability distribution q_v over letters of the alphabet[a] 'a'–'z', and a probability distribution p_v over edges $E_v \subseteq E$ leaving v. So $q_{v,x} \geq 0$ is the probability that the letter $x \in \{\text{`}a', \ldots, \text{`}z'\}$ is produced at node v. Because it is a probability distribution, we know that $\sum_{x \in \{\text{`}a', \ldots, \text{`}z'\}} q_{v,x} = 1$ for all v. Similarly, $p_{v,e} \geq 0$ is the probability of taking edge e out of v, for each $e \in E_v$. Here, we know that $\sum_{e \in E_v} p_{v,e} = 1$ for all nodes v.

The way such a system produces an output is now as follows: the system starts in $v_1 = s$. Then, for each time step $t = 1, 2, \ldots$, assume that the system is at node v_t. It randomly picks a letter x_t to output according to the distribution q_{v_t}. Then, it randomly picks an edge $e = (v_t, u)$ out of v_t to follow, according to the distribution p_{v_t}. The endpoint u of the edge e it picked becomes the new node $v_{t+1} = u$. This repeats for some number

$T \geq 0$ of steps. As a result, it produces some output string x as the sequence of letters x_t that are output. The computational question we are facing is now the following: we know G and s (and all the probabilities), but we can't see the sequence $\langle v_1, v_2, \ldots \rangle$ of vertices that the system is at. All we can observe is the output, i.e., the sequence of letters $x = x_1 x_2 \cdots x_T$ (which is also part of our input). There may be many different sequences $\langle v_1, v_2, \ldots, v_T \rangle$ of vertices which could have produced this same sequence x of letters. Among all of them, the goal is to output one with largest likelihood. The likelihood of a sequence $\langle v_1, v_2, \ldots, v_T \rangle$ is defined as

$$L(\langle v_1, v_2, \ldots, v_T \rangle) = \prod_{t=1}^{T} q_{v_t, x_t} \cdot \prod_{t=1}^{T-1} p_{v_t, (v_t, v_{t+1})}.$$

So it is the product of the probability of outputting the observed character x_t in each of the assumed states v_t, times the probability of taking the edge from v_t to v_{t+1} for each of the time steps $t = 1, \ldots, T-1$.
Give and analyze (running time and correctness) an algorithm for finding the maximum likelihood of any vertex sequence for the given string. You do not need to output the actual sequence. You should give pseudo-code for an actual implementation, but if you prove correctness of a recurrence, you do not need to do another correctness proof for the pseudo-code.

[a] We could make them numbers, or anything else.

Solution

First let's look at the algorithm. Note that based on the description of the problem, we *DO* have $p_{v,(v,v)}$ for a certain vertex v, thus $\sum_{e \in E_v} p_{v,e} = 1$.

```
stack<char> findGreatestLikelihood(G=(V,E), s, x, p, q){
    // All input parameters have the same meaning given in the problem, x has size T based on given
    double dp[][] = new int[V.size][T];

    // Initialization
    Initialize all values in dp[][] to 0;
    Initialize all values in B[][] to null;
    dp[s][1] = q_{s,x_1};

    // Main Recurrence
    for (each t : 2 ≤ t ≤ T){
        for (each v ∈ V){
            for (each u ∈ E_v){ // In other words, for all adjacent vertices u of v
                double newP = dp[u][t-1] · p_{u,(u,v)} · q_{v,x_t};
                if (dp[v][t] < newP) {
                    dp[v][t] = newP;
                }
            }
        }
    }

    // Backtracing
```

```
        int maxProb = max_{v∈V}[v][T];
        return maxProb;
    }
```

Then let's consider some key term in this algorithm:

Definition 0.1: $dp\,[v]\,[t]$ and Recurrence Relation

We define the $dp\,[v]\,[t]$ as the greatest probability of the sequence which has x_t at the t^{th} digit of the sequence while possessing the status of v.
And we have the recurrence relationship for some $v \in V$ and $2 \leqslant t \leqslant T$:

$$dp\,[v]\,[t] = \max_{\text{all adjecent vertice } u} dp\,[u]\,[t-1] \cdot p_{u,(u,v)} \cdot q_{v,x_t}$$

For $t = 1$, $dp[s][1] = q_{s,x_1}$ and for other $v \in V$, we have $dp[v][1] = 0$ as base case.

The algorithm is performing the recurrence relationship above, and based on the requirement of the problem, it suffices to show the correctness of the recurrence relationship.
Knowing this, let's prove the recurrence relationship is correct by induction on t. We are trying to prove that

$$\max_{v \in V} dp\,[v]\,[t] = OPT(\{x_1, \cdots, x_t\})$$

Proof by infuction. We conduct proof by induction on t.
Base case $t = 1$: We already know the starting point is s, so $dp[s][1] = q_{s,x_1}$, while other $dp[v][1] = 0$ for other $v \in V$.

$$\max_{v \in V} dp\,[v]\,[1] = \max_{v = v_1, \cdots, v_t} L(\langle v \rangle) = L(\langle s \rangle) = q_{s,x_1}.$$

Thus, the base case holds.
Induction Hypothesis: Assume $dp\,[v]\,[i]$ for any i which $2 \leqslant i \leqslant k-1$, satisfy

$$\max_{v \in V} dp\,[v]\,[i] = OPT\,(\{x_1, \cdots, x_i\})$$

Induction step: The answer we are trying to find right here is

$$\max_{v \in V} dp\,[v]\,[k]$$

And for each Note that based on the definition of the likelihood of a sequence $\langle v_1, v_2, \cdots, v_t \rangle$,

$$\begin{aligned} L(\langle v_1, v_2, \ldots, v_t \rangle) &= \prod_{i=1}^{t} q_{v_i,x_i} \cdot \prod_{i=1}^{t-1} p_{v_i,(v_i,v_{i+1})} \\ &= \prod_{i=1}^{t-1} q_{v_i,x_i} \cdot \prod_{i=1}^{t-2} p_{v_i,(v_i,v_{i+1})} \cdot q_{v_t,x_t} \cdot p_{v_{t-1},v_t} \\ &= L(\langle v_1, v_2, \ldots, v_{t-1} \rangle) \cdot q_{v_t,x_t} \cdot p_{v_{t-1},v_t} \end{aligned} \tag{1}$$

So we want to compute

$$\max_{v = v_1, v_2, \cdots, v_t} dp\,[v]\,[k]$$

It is computing the maximum out of the maximum of each

$$dp[v][t]$$

for each v_i, it could be reached from all $v_u \in V$, which partitioned all possibilities reaching v_i. Since the choice is indecent from x_{k-1} to x_k, thus we know that each of them satisfying the equation (1). Thus,

$$\begin{aligned}\max_{v \in V} dp\left[v\right]\left[k\right] &= \max_{v \in V; u \text{ adjacne to } v} dp\left[u\right]\left[k-1\right] \cdot p_{u,(u,v)} \cdot q_{v,x_k} \\ &= \max_{u \in V} OPT\left(\{x_1, \cdots, x_{k-1}\}\right) \cdot p_{u,(u,v)} \cdot q_{v,x_k} \\ &= OPT\left(\{x_1, \cdots, x_k\}\right)\end{aligned}$$

The algorithm does not contain while loop and will terminate, thus concludes the proof. □

Runtime Analysis. Line 3 to 8 is dominated by 2 initializations, which takes $O\left(T|V|\right)$ time.
Line 11 to 20 is a nested loops, which loops $T|V|$ times in the outer two loops. As for the innermost loop on line 14, it takes at most V iterations, each takes constant time operations. Thus this block takes $O\left(T|V|^2\right)$ time.
As for line 23 and 24, it takes constant time.
Thus, the algorithm is dominated by $O\left(T|V|^2\right)$ runtime complexity. □