CS301 - Homework 4

Mustafa Yagiz Kilicarslan - 28126

December 11, 2022

1 Agricultural robotics problem

Recursive Formulation

Define Farm as a n x m sized matrix representing the 2D farm area.
For each cell (i, j), set Farm[i][j] = (i, j).
Define W as a n x m sized matrix representing weedy cells on the farm.
For each cell (i, j), set W[i][j] = 1 if cell (i, j) is weedy, otherwise set W[i][j] = 0.
Let $c[i][j]$ to be the maximum number of weedy cells on the path p from $s = (1, 1)$ to $t = (i, j)$.
Then $c[i][j]$ can be defined as the following recursive formulation:

$$c[i][j] = \begin{cases} 1 & i = 1, j = 1 \\ c[i][j-1] + W[i][j] & i = 1, 1 < j < m \\ c[i-1][j] + W[i][j] & j = 1, 1 < i < n \\ max(c[i-1][j], c[i][j-1]) + W[i][j] & otherwise \end{cases}$$

In other words, maximum amount of weed that the robot can collect starting from (1, 1) to any given cell (i, j) will be the maximum of the left and upper cells plus the worth of this cell.

Keep in mind that we need to handle left most column and top row separately as shown above so that our recursive algorithm do not exit the farm area.

Pseudocode of your algorithm

1: **procedure** RECURSIVE DYNAMIC PROGRAMMING SOLUTION(*farm*, n, m)
2: $c \leftarrow n$xm array all cells set to 0
3: $W \leftarrow n$xm array weedy cells set to 1, others set to 0
4: $FarmRobotIterativeDP(W, c, n, m)$
5: Return $FarmRobotPathFinder(c, n, m)$
6: **end procedure**

This pseudocode takes the farm and the size of the farm (n, m) as the input and returns a path from the starting point to the target. To do that, it first calculates the number of weeds that it can collect, starting from the starting cell to the other cells on the farm. It utilizes a table (tabulation), similar to LCS solution discussed in the class, to keep the previously calculated subproblems and uses those values in new calculations. After the table c is filled, then it uses backstracking, again similar to LCS, to create path from start to the target.

```
1: procedure FarmRobotIterativeDP(W, c, n, m)
2:     for i ← 1, n do
3:         for j ← 1, m do
4:             if i = 0 and j = 0 then
5:                 c[i][j] = W[i][j]
6:             else if i = 0 and m > j > 0 then
7:                 c[i][j] = W[i][j] + c[i][j − 1]
8:             else if n > i > 0 and j = 0 then
9:                 c[i][j] = W[i][j] + c[i − 1][j]
10:            else then
11:                c[i][j] = W[i][j] + max { c[i][j − 1]
                                           { c[i − 1][j]
12:            end if
13:        end for
14:    end for
15: end procedure
```

As it can be seen from the pseudocode, we will use a bottom-up approach to implement a dynammic programming algorithm. To go over each distinct subproblem we will use two loops (one nested in other). Within the loops we will check (i, j) to see if any of the following conditions are met. Only distinction from the recursive formulation is that indexes go from i = 0..n-1, j = 0..m-1 while in the recursive formulation they go from i = 1..n, j = 1..m.

If (i, j) is the starting cell (0, 0), then set $c[i][j] = W[i][j]$, meaning that if starting cell is weedy set the corresponding cell in c to 1, otherwise set it to 0.

If not starting cell and if a cell in the top row (i = 0, m ¿ j ¿ 0), then since this is a special condition where we need to be sure that it doesn't go out of range of the farm field, set the corresponding c cell to be the c value of the left cell plus the the current cells weedy value. If you were to go from the starting cell to the current cell, maximum amount of weed that you can collect is the left cells c value plus this cells weedyness.

Similarly if left most column (j = 0, n ¿ i ¿ 0), if you were to go from the starting cell to the current cell, maximum amount of weed that you can collect is the upper cells c value plus this cells weedyness.

Lastly, if not the starting cell nor a top row or most left column cell, then the current cells c value is equal to the maximum of upper cells and left cells c value plus the weedyness of the current cell.

Since we have calculated the left and upper cell c values when calculating a new distinct subproblem, we can simply look at their values and determine the new subproblems value accordingly. At the end we will reach the target cell and similarly calculate its value. At that state we have reached our solution. Since the c array is now filled, we can backstrack it (similar to LCS problem) to create a path from the starting cell to the target.

```
1: procedure FarmRobotPathFinder(c, n, m)
2:     i ← n − 1
3:     j ← m − 1
4:     path ← [(i, j)]
5:     while i > 0 and j > 0 do
6:         if i == 0 then
7:             j ← j − 1
8:         else if j == 0 then
9:             i ← i − 1
10:        else if c[i − 1][j] > c[i][j − 1] then
11:            i ← i − 1
12:        else
13:            j ← j − 1
14:        end if
15:        path.append((i, j))
16:    end while
17:    return reverse(path)
18: end procedure
```

This procedure utilizes backstracking algorithm similar to of LCS problem discussed in the lecture. In simple words, this algorithm takes the filled value table (c in this example), which has the maximum number of weeds that can be passed over from starting point. In this case, we will go from the target cell (n-1, m-1) to the starting cell (0, 0) backwardly. In that reverse path we will choose a new cell based on which is the maximum, left or upper, which can be seen in line 10. However, remember that if we are in the top row or the left most column then we need to update the path accordingly. In line 6, it shows that if we are in the top row, then we need to follow the path to left. In line 8, we update the path to upper cell if we are on the most left column. At the end, we return the reversed path, which has the cells of the path from start to the end as a list.

Time Complexity

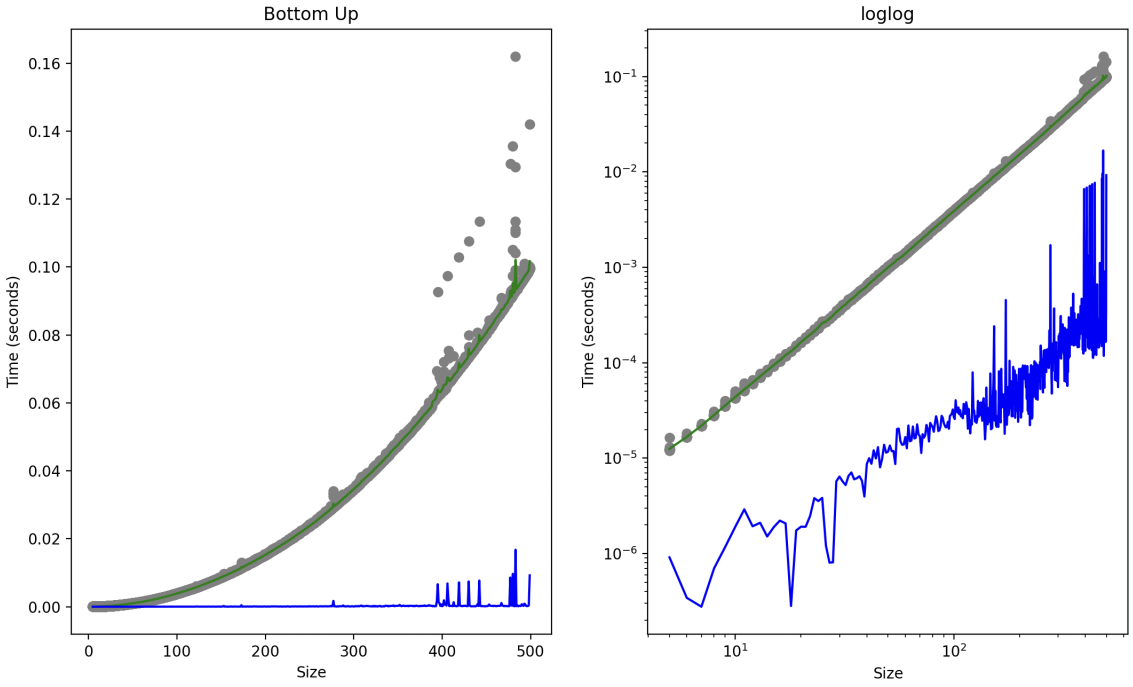
Time complexity of the dynamic programming algorithm is equal to at least the number of distinct subproblems in the given problem. For each tile in the farm we calculate the maximum number of weeds that can be collected for that cell. Since the size of the farm (nxm) is our input, the number of cells is equal to $n * m$. Therefore the dynamic programming algorithm takes $O(nm)$ time in the size of input n, m.

Additionally, the algorithm has a path finding procedure that takes the table calculated in the previous step as an input. Backstracking algorithm decreases the i and j parameters one by one until both reach to starting point (0, 0). Therefore, this operation takes $O(n + m)$ time similar to LCS backstracking.

Considering both of these procedures combined, overall algorithm takes $O(n * m) + O(n + m) = O(n * m)$ time complexity in the size of input.

Experimental Results

Following are the plots of performance testing results. Experimental findings confirm the theoretical results, where they are exponential, $O(n^2)$, in the size of input. Below, I explain why this result supports the theoretical results.



Legend: Mean = Green Line — Standard Deviation = Blue Line — Individual Samples = Grey Dots

In the performance tests, I set the size parameters as $n = m$ analyze the results easily. As can be seen from the left plot, the time complexity is exponential $O(n^2)$ in the size of input, which is **expected** because we set $n = m$, therefore the theoretical result $O(nm) = O(n * n) = O(n^2)$.

To be sure that the results are exponential, the right graph takes the logarithm of both the y-axis and x-axis, which is also known as *loglog plot*. The slope of this graph is the power of the exponential term, in our case 2. This proves that the experimental result support the theoretical results.

Details of these tests can be seen in the provided Python code, in which the White-box and Black-box testings are done extensively.