CS301 ASSIGNMENT 4

Şevval Karadeniz 28181

December 11, 2022

1 Recursive formulation of agricultural robotics problem.

1.1 Identification of sub-problems

To find a way to find the path for collecting the maximum number of weeds we need to consider a better way for each cell. We can come to each cell from the left or top of the cell. Therefore, when we are trying to find the path for collecting a maximum number of weeds we need to check whether the left of the cell has come with a higher number of weeds in total or the top of the cell has come with a higher number of weeds in total. According to this comparison, we can decide which way we should continue since this is a recursive algorithm we are doing the search backward. So, in each iteration, we are going back from the end of the matrix to the (0,0) point of the matrix. Then, we are choosing the option that has the higher number of weeds. As a result, our sub-problems are for each cell, finding whether the cell that is above our current cell has a path with a higher number of weeds than the cell that is on the left of our current cell or vice-versa.

1.2 Optimal Sub-structure Property

To find the path with the maximum number of weeds, we should consider the best way for each cell in the way. In other words, when we are finding a way with a maximum number of weeds, we are also finding a way with a maximum number of weeds for each cell in the path. So, it is not just the best option, in the end, it is also the best option for each sub-problem

1.3 Overlapping Subproblems

Since we are checking both the above and the left side of each cell we may have to check the same cell more than once. In this case, we need to calculate the same problem again.

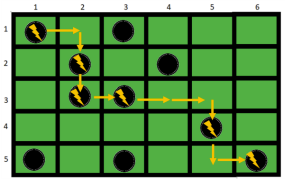

Figure 1: Path with maximum number of weeds

For example, example in the above, since our algorithm is recursive we are going backward. When we are coming back from cell (5,6), we need the check cells (5,5) and (4,6). For cell (5,5) we need to check (4,5) and (5,4) for cell (4,6) we need to check (4,5) and (3,6) since we check (4.5) more than once we can say that we have overlapping sub-problems

1.4 Recursive Formulation for finding the path

For finding the maximum number of weeds we should call the recursive function F(m,n): F(m,n)= max { F(m-1,n), F(m,n-1)+ C(m,n) } where C(m,n) is 1 if current cell has a weed, 0 otherwise

For finding the path, we should call the recursive function F'(m,n) for appending the path to the array dp: dp.append(F(m-1,n) ≥ F(m,n-1) ? F(m-1,n): F(m,n-1))

If F(m-1,n) ≥ F(m,n-1) call F'(m-1,n) else call F'(m,n-1)

2 Pseudocode of the algorithm

2.1 Naive Recursive Algorithm

```
def findingMaxNumberOfWeed(table[][],m,n,c){
        if ( (m<0) || (n<0) ) {return;}
        if ( (m==0) ∩ (n==0) ) {
            table[m][n]=c[m][n] ;
                return table[m][n];
        }
        if ( (m==0) ∩ (n! =0) ) {
            table[m][n]=max{findingMaxNumberOfWeed(table,m,n-1,c),0} + c[m][n]
                return table[m][n];
        }
        if ( (m! =0) ∩ (n==0) ) {
            table[m][n]=max{findingMaxNumberOfWeed(table,m-1,n,c),0} + c[m][n]
                return table[m][n];
        }
        else{
            table[m][n]=
            max{findingMaxNumberOfWeed(table,m-1,n,c),findingMaxNumberOfWeed(table,m,n-1,c)}+C[m][n];
            return table[m][n];
            }
}
        def findingPath(table[][],m,n,pathArray[]){
            // table is the array that we create with maxNumberOfWeed function
                if ( (m<0) || (n<0) ) {return;}
                if ( table[m-1][n] > table[m][n-1] ) {
                    pathArray.append("m-1,n");
                        findingPath(table,m-1,n,pathArray);
                }
                else{
                    pathArray.append("m,n-1");
                        findingPath(table,m,n-1,pathArray);
                }
        return pathArray;
            }
```

2.2 Dynamic Programming Recursive Algorithm with Memoization

In order to prevent the calculation of the same sub-problems again and again we need a memoization method

```
        def findingMaxNumberOfWeed(table[][],m,n,c){
            // where c is the array that symbolizes our farm, and the table is the array that we
            are creating for finding the maximum number of weeds that are collectible until reaching a
            certain cell
                if ( (m<0) || (n<0) ) {return;}
                if ( table[m][n] != undefined ) {return table[m][n];} // I assumed if the value is
                not calculated yet it is undefined
                if ( (m==0) ∩ (n==0) ) {
```

```
                table[m][n]=c[m][n] ;
                    return table[m][n];
            }
            if ( (m==0) ∩ (n! =0) ) {
                table[m][n]=max{findingMaxNumberOfWeed(table,m,n-1,c),0} + c[m][n]
                    return table[m][n];
            }
            if ( (m! =0) ∩ (n==0) ) {
                table[m][n]=max{findingMaxNumberOfWeed(table,m-1,n,c),0} + c[m][n]
                    return table[m][n];
            }
            else{
                table[m][n]=
                max{findingMaxNumberOfWeed(table,m-1,n,c),findingMaxNumberOfWeed(table,m,n-
            1,c)}+C[m][n];
                return table[m][n];
                }
    }
    def findingPath(table[][],m,n,pathArray[]){
        // table is the array that we create with maxNumberOfWeed function
            if ( (m<0) || (n<0) ) {return;}
            if ( table[m-1][n] > table[m][n-1] ) {
                pathArray.append("m-1,n");
                    findingPath(table,m-1,n,pathArray);
            }
            else{
                pathArray.append("m,n-1");
                    findingPath(table,m,n-1,pathArray);
            }
    return pathArray;
        }
```

3 Asymptotic time and space complexity analysis

For the algorithm with memoization:

- Finding the max number weed part have the time complexity of O(mn) since we travel all the cells and make calculations for each cell only once because we put them in the array. We do not need to re-calculate the same amount again and again
- finding the path part: Since we choose one path in each iteration and ignore the other one we can write the recursion like T(mn)= T(mn/2) + O(1). If we solve this recursion time complexity will be O(lgmn)
- Total time complexity will be: O(mn) + O(lgmn) = O(mn)
- For the space complexity since we use two different 2d arrays in order to store values the space complexity will be 2*O(mn)

4 Experimental evaluations of the algorithm & Test Cases

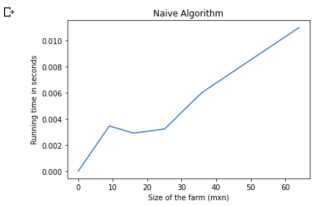

Figure 2: Result of naive algorithm is almost linear and taking much more time than the dynamic programming approach with memoization

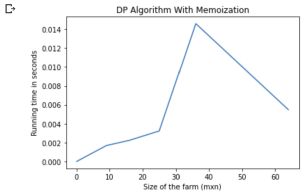

Figure 3: Result of dynamic programming with memoization is almost linear even if decreasing when farm area increasing for some value

Figure 4: TestCase1: For a big farm(Result of 20x20 farm)

```
Farm
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
Path
[['p' '.' '.' '.' '.']
 ['p' '.' '.' '.' '.']
 ['p' '.' '.' '.' '.']
 ['p' '.' '.' '.' '.']
 ['p' 'p' 'p' 'p' 'p']]
Cells that we need to visit in order to collect maximum number of weed
['[0][0]', '[1][0]', '[2][0]', '[3][0]', '[4][0]', '[4][1]', '[4][2]', '[4][3]', '[4][4]']
Execution time: 0.0019335746765136719 seconds
```

Figure 5: TestCase2:When there is not any weed

```
Farm
[[1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]]
Path
[['p' '.' '.' '.' '.']
 ['p' '.' '.' '.' '.']
 ['p' '.' '.' '.' '.']
 ['p' '.' '.' '.' '.']
 ['p' 'p' 'p' 'p' 'p']]
Cells that we need to visit in order to collect maximum number of weed
['[0][0]', '[1][0]', '[2][0]', '[3][0]', '[4][0]', '[4][1]', '[4][2]', '[4][3]', '[4][4]']
Execution time: 0.015130281448364258 seconds
```

Figure 6: TestCase3:When there is a weed in every cell

```
Farm
[[0 0 0 0 1]
 [0 0 0 0 1]
 [0 0 0 0 1]
 [0 0 0 0 1]
 [0 0 0 0 1]]
Path
[['p' 'p' 'p' 'p' 'p']
 ['.' '.' '.' '.' 'p']
 ['.' '.' '.' '.' 'p']
 ['.' '.' '.' '.' 'p']
 ['.' '.' '.' '.' 'p']]
Cells that we need to visit in order to collect maximum number of weed
['[0][0]', '[0][1]', '[0][2]', '[0][3]', '[0][4]', '[1][4]', '[2][4]', '[3][4]', '[4][4]']
Execution time: 0.009818792343139648 seconds
```

Figure 7: TestCase4: When there is a weed in just one column

```
Farm
[[1 1 1 1 1]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
Path
[['p' 'p' 'p' 'p' 'p']
 ['.' '.' '.' '.' 'p']
 ['.' '.' '.' '.' 'p']
 ['.' '.' '.' '.' 'p']
 ['.' '.' '.' '.' 'p']]
Cells that we need to visit in order to collect maximum number of weed
['[0][0]', '[0][1]', '[0][2]', '[0][3]', '[0][4]', '[1][4]', '[2][4]', '[3][4]', '[4][4]']
Execution time: 0.0017795562744140625 seconds
```

Figure 8: TestCase5: When there is a weed in just one row

```
Farm
[[1 0 0 0 0 0]
 [0 0 0 1 0 0]
 [0 1 0 0 0 0]
 [0 0 0 0 1 0]
 [0 0 1 0 0 0]]
Path
[['p' '.' '.' '.' '.' '.']
 ['p' '.' '.' '.' '.' '.']
 ['p' 'p' '.' '.' '.' '.']
 ['.' 'p' '.' '.' '.' '.']
 ['.' 'p' 'p' 'p' 'p' 'p']]
Cells that we need to visit in order to collect maximum number of weed
['[0][0]', '[1][0]', '[2][0]', '[2][1]', '[3][1]', '[4][1]', '[4][2]', '[4][3]', '[4][4]', '[4][5]']
Execution time: 0.001828908920288086 seconds
```

Figure 9: TestCase6: When there is one weed for each row

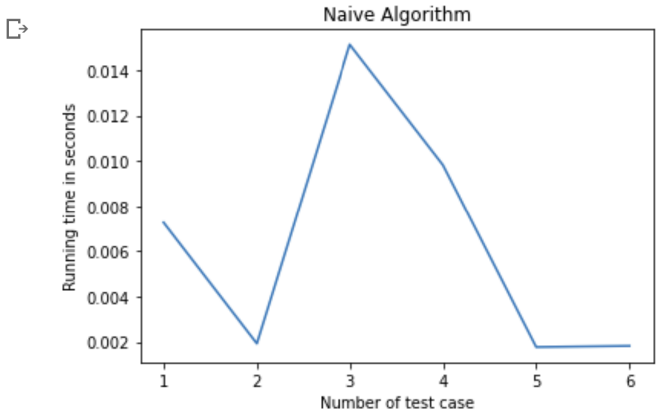

Figure 10: Time that is taken by each test case