CS301 HW3

Adnan Kaan Ekiz

April 2019

1 Introduction

In Homework 3 of the Algorithms course, main idea is to implement 3 different solutions for the traffic lane problem. During the process of implementing the code, it is important to recognize and briefly mention some of the key parts. For that reason, Homework 3 report will present some idea about the algorithms implemented. It will also show the results of these problems with graphs as well as identifying different subproblems, observing the optimal substructure property and the overlapping computations.

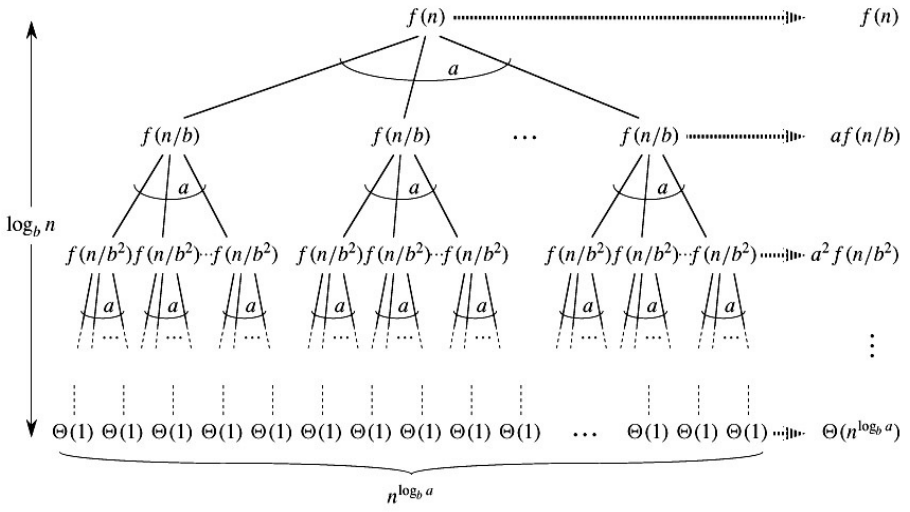
2 Recursive Formulation of the Problem

Recursive formulation of the algorithm requires to call the same recursive function for each lane in our road matrix. Basic structure of the recursive formulation starts with the stopping conditions that is needed for all of the subproblems. General formulation then starts to call the same recursive function for the next 3 lane. Subproblems can be listed as follows:

- Calling 3 new functions for each possible cost value for lanes in the next level
- Calculate the minimum cost for the return value of each function
- Control condition to check if computation reach to an end

Algorithm is implemented in a way that at each call of the function, it will call 3 subproblems related with 3 lanes at the next level. Computation of the lane changes may overlap due to lack of memorization and this will cause inefficiency for the Naive Recursive Algorithm that is going to be mentioned later. Overlaps are caused by the unnecessary computations of the same lane in any given level. For example, one of the overlapping condition may happen when columns in level l_k c_0, c_1 and c_2 tries to compute the column c_1 in level l_{k+1} which will cause the same point to be computed more than once.

General formulation of the recursive algorithm tree can be shown in below:



During the recursive iteration, algorithm calls the same recursive function until it reaches to the end of the road which is the last level in road matrix. After returning the cost value in the last level, it starts to increment the cost for points that are diagonal to each other while returning the original cost value without incrementing for adjacent points in the matrix. Implementation of each algorithm will be described in the next section of the report.

3 Pseudocode and Complexity of Algorithms

There are 3 different implementations to solve the given traffic lane problem in homework. These implementations are:

1. Naive Recursive Algorithm
2. Recursive Algorithm with Top-Down Memoization
3. Iterative Algorithm

3.1 Naive Recursive Algorithm

Algorithm does not use any memorization during the computation step. Therefore, pseudocode of the algorithm is a bit shorter than the other algorithms even though it has a high time and space complexity compared to other algorithms.

Naive Recursive one takes 3 parameters at the beginning of algorithm. These are starting starting point represented as x and y and also our road matrix to determine whether next point is an obstacle or not. Pseudocode of the algorithm is as follows:

Algorithm 1 Naive Recursive Solution for Traffic Lane Problem

```
1: procedure NaiveRecursive [Matrix Road, Int x, Int y]
2:     if x equals len(road)-1 then return 0
3:     costs ← list[]
4:     if y > 0 and road[x+1][y-1] equals 0 then
5:         costs ← add NaiveRecursive(road, x+1, y-1)+1
6:     if road[x+1][y] equals 0 then
7:         costs ← add NaiveRecursive(road, x+1, y)
8:     if y < 2 and road[x+1][y+1] equals 0 then
9:         costs ← add NaiveRecursive(road, x+1, y+1)+1
       return min(costs)
```

For the Naive Recursive Algorithm, space and time complexity creates a problem for mediocre n values. To analyze the asymptotic time in this algorithm we can look at the last level of the road matrix. We know that even though cost of one point was calculated before, algorithm calculates the value of that point again and again for at most $O(3^l)$ where l is the number of total levels in the road matrix. At this point, columns in the last level will be called for 3^l times, as a result not using any memory table. When we analyze the algorithm for space complexity, algorithm passes the road matrix by reference for future calls of the same function. We also have a list called costs in order to determine the minimum cost for the next 3 columns. Hence, algorithm creates the same costs list 3^l times for only the last level of the road matrix which is a huge problem for big l values.

3.2 Recursive with Top-Down Memorization

Top-down memorization uses an extra parameter called memory in order to store the calculated points in the road matrix. During the implementation of the algorithm, I have decided to use dictionary such that the combination of x and y becomes a key and total cost for that point in the matrix is the value of the key.

For some column in level l_k, algorithm checks if any of the available columns in level l_{k+1} exists in the dictionary before the computation of that specific point. Available columns in next level are decided by these three conditions:

- Algorithm checks to make sure that column c_k in next level and column c_x in current level has at most 1 index difference between them.
- If column in next level is an obstacle, then don't calculate that point recursively.
- Index value of column c_k should be in between the following interval : [0,2].

Algorithm then calls it self recursively while checking if column in the next level is available. When it reaches to the last level of the road matrix it returns 0 and for every column change in the matrix increments the cost of reaching to that point by 1.

When analyzing the algorithm in terms of space complexity, it passes both road matrix and memory table by reference. There are also cost arrays for each function that is called. When number of levels are not a really big number, algorithm does okay since each of these cost functions will store at most 3 elements gathered from 3 different columns. Hence, algorithm does not cover much space for small road matrices. On the other hand, time complexity of top-down recursive algorithm seems really similar to asymptotic time of iterative one. This is due to the fact that algorithm tries to learn the minimum cost value of the column in next lane by making use of the memory table that was inserted. Depending on the implementation of the algorithm, during the first call of the function, algorithm adds almost all of the future values to memory table. Hence, during the later calls we can retrieve the computed values from our memory table. Retrieving the values from our table causes the algorithm to behave very similar to iterative algorithm. That is why algorithm has O(n) time complexity in general. Pseudocode of the algorithm:

Algorithm 2 Top-down Recursive Solution for Traffic Lane Problem

```
1:  procedure TopDownRecursive [Matrix Road, x, y, MemoryTable]
2:      if x equals len(road)-1 then return 0
3:      costs ← list[]
4:      if y > 0 and road[x+1][y-1] equals 0 then
5:          if point in MemoryTable then
6:              costs ← add MemoryTable[point]+1
7:          else
8:              costs ← add TopDownRecursive(road,x+1,y-1,MemoryTable)+1
9:      if  road[x+1][y] equals 0  then
10:         if point in MemoryTable then
11:             costs ← add MemoryTable[point]
12:         else
13:             costs ← add TopDownRecursive(road,x+1,y-1,MemoryTable)
14:     if  y < 2 and road[x+1][y+1] equals 0  then
15:         if point in MemoryTable then
16:             costs ← add MemoryTable[point]+1
17:         else
18:             costs ← add TopDownRecursive(road,x+1,y+1,MemoryTable)+1
        return min(costs)
```

3.3 Iterative Algorithm

In the iterative version of the algorithm, we do start from the beginning of the road matrix and iterating through the end of the matrix while computing the minimum cost. Assuming that we have a column c_k in level l_k, minimum cost is computed in a way that it takes the minimum of column values c_j, c_{j-1}, c_{j-2} if and only if these columns have at most 1 index difference from c_k for level l_{k-1}.

At the end of iteration, algorithm uses the minimum function to compute the lowest cost in the last level which represents the lowest value for the road matrix. In terms of time and space complexity, algorithm iterates each of the k levels of matrix. Therefore, algorithm processes 3*n elements in total which has O(n) complexity in terms of time. When it comes to space complexity, algorithm passes both of the matrices by reference, hence space complexity is only limited to two arrays which holds the road map and the memory.

Algorithm 3 Iterative Solution for Traffic Lane Problem

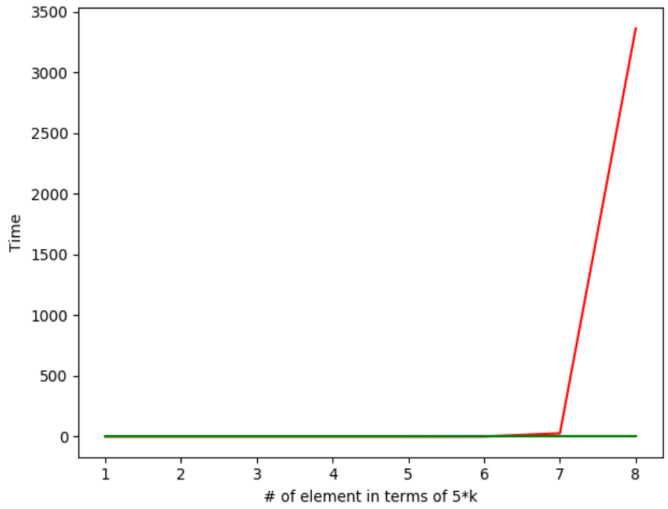
```
1:  procedure IterativeAlgorithm [Matrix Road, Matrix Memory]
2:      ct ← 0
3:      while counter < len(road) - 1 do
4:          ct ← ct + 1
5:          if ct = 1 then
6:              if road[ct][0] not equal to 1 then
7:                  memory[ct][0] ← 1
8:              if road[ct][1] not equal to 1 then
9:                  memory[ct][1] ← 0
10:             if road[ct][2] not equal to 1 then
11:                 memory[ct][2] ← 1
12:         else
13:             if road[ct][0] not equal to 1 then
14:                 memory[ct][0] ← min(cost of adjacent columns in level ct-1)
15:             if road[ct][1] not equal to 1 then
16:                 memory[ct][1] ← min(cost of adjacent columns in level ct-1)
17:             if road[ct][2] not equal to 1 then
18:                 memory[ct][2] ← min(cost of adjacent columns in level ct-1)
        return min(costs of the last level)
```

4 Experimental Values and Benchmark Suites

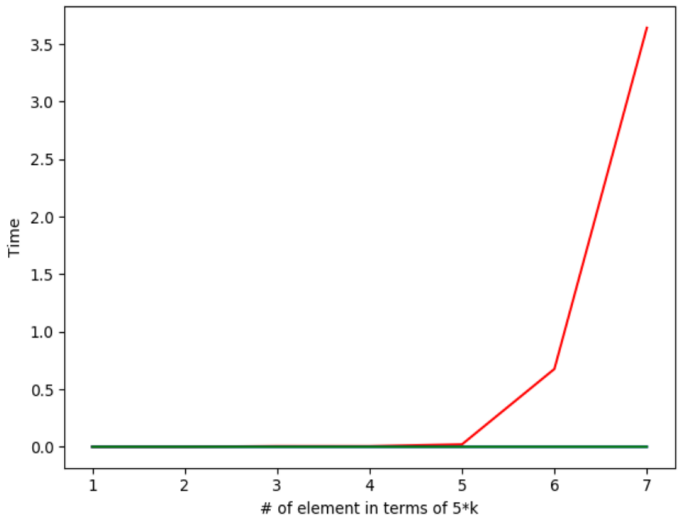
4.1 Experimental Values

Experiments handled with input size in format 5*k where k is changing over time. At the beginning I have implemented a function that creates road matrices

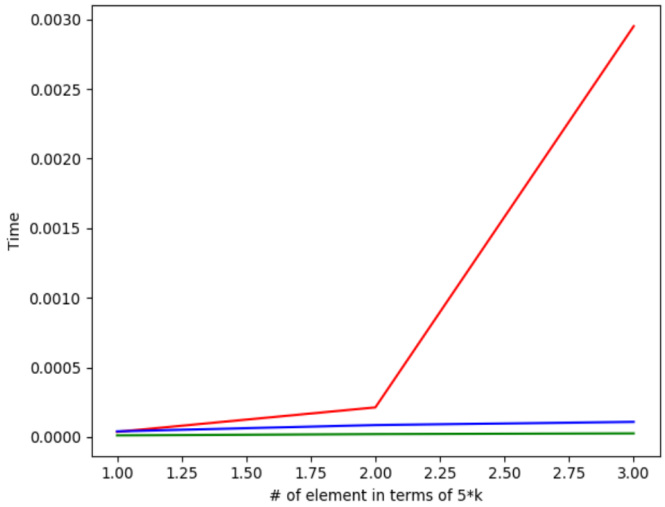
where each level has only 1 obstacle in it. Then conducted experiments with road matrices that has a level size of [5,10,15,20,25,30,35,40] and results showed the following graph where red denotes the time of Naive Recursive, blue denotes the time of Top-Down Recursion with Memorization and green denotes the time of Iterative Algorithm:



In order to show more detailed behavior of the time curve, it was important to decrease the level size of the input. Therefore, plotted results of these 3 algorithms are given below with inputs with level sizes. Input with level sizes [5,10,15,20,25,30,35]:



Input with level sizes [5,10,15]:



Results indicate that in terms of time, iterative algorithm has the minimum time while naive recursive algorithm has a high complexity that changes dramatically with the input size. When plotting the results there seems to be no difference between Iterative and Top-Down algorithms. We can see the difference between times from the table given below:

Algorithm Type	Naive Recursive	Top-Down Recursive	Iterative
Level size(l)		Algorithm Time(s)	
l = 5	1.2159347534e-05	2.3126602172e-05	6.9141387939e-06
l = 10	0.0001940727	5.2928924560e-05	1.4066696166e-05
l = 15	0.0002868175	7.5101852416e-05	1.9073486328e-05
l = 20	0.0084428787	0.0001089572	2.6702880859e-05
l = 25	0.1789031028	0.0001361370	3.2901763916e-05
l = 30	1.6840751171	0.0001811981	5.2928924560e-05

In general, results were not surprising since I expected Naive Recursive algorithm to be the slowest one due to the computation of single item more than once. Iterative algorithm was also expected to have the smallest computation time since same function and instances were not created more than once. Even though Top-Down Recursive function has similar behavior to Iterative one, it calls the same function and creates costs array for each call of the function that causes some overhead in terms of time complexity.

4.2 Benchmark Suites

During the benchmark test, all algorithms made several assumptions based on given homework document. These assumptions are the following:

- Beginning point was assumed as the middle column of the first level, therefore all algorithms will try to compute the minimum cost to last level based on this fact.
- Starting level was assumed to have no obstacles since it may also affect the starting point.
- All levels have only 1 obstacles that is placed randomly to one of the columns in level l.

After doing the black-box testing for algorithms with several random road matrices, I have decided to control the algorithms with several edge cases in order to prove that algorithm is working properly. These edge cases are defined to show different and unexpected road matrices. Hence white-box testing includes the following:

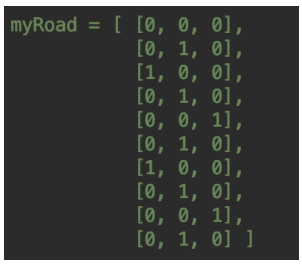


Figure 1: Expected output: 5

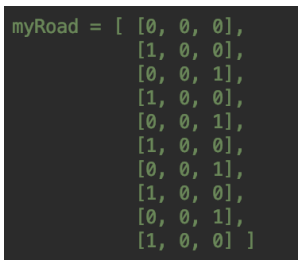


Figure 2: Expected output: 0

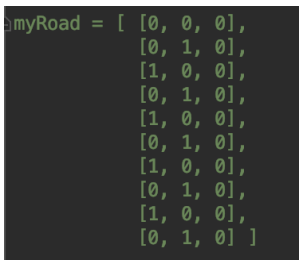


Figure 3: Expected output: 1

After running the code for these 3 different road matrices, results were similar to the expected outcome of each matrix:

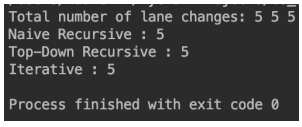


Figure 4: Actual output: 5

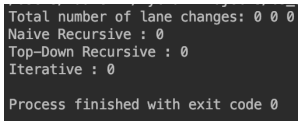


Figure 5: Actual output: 0

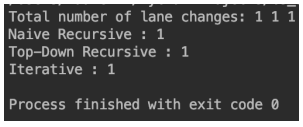


Figure 6: Actual output: 1