Grid Path

Buzdugă Ionuţ Gabriel
Group 1.1B
Year 1
Computer Science

1 Problem Statement

Grid Path

Consider a field in form of a squared grid of nxn dimensions. Each location of the grid is defined by a positive integer which represents the elevation(height of a point of the field projected on a horizontal reference plane).

Find a path from the leftmost top corner to the corner of the field located in the bottom right such that:

i)Movement along the field can only be made down or to the right.

ii)The cost of the path,calculated as the sum of the absolute values of the difference between to consecutive locations along the path is minimum. Two algorithms will be implemented.

2 Pseudocode

In this section there are presented the two algorithms used for solving the problem.

1. The first Algorithm uses a recursive aproach which is a fairly trivial solution to the problem.Because the same sub-problems are going to be solved multiple times,the time complexity is will be exponential, $\Theta(2^n)$.For every cell there are two options.

2. The second algorithm uses Dynamic programming by creating an aditional matrix in which each element stored holds the value of the minimum cost path up until that element.

   After the matrix is filled,we can start printing the path starting from the top left corner and choosing the minimum value of the two moves we are allowed to make,going right or down.The time complexity for this algorithm is the size of the matrix,$\Theta(n^2)$.

To simplify the Pseudocode appeareance,the names of some variables were shortened:

- $matrix \Rightarrow A$
- $row\_index \Rightarrow i$
- $col\_index \Rightarrow j$
- $MinCost \Rightarrow B$
- The abso(**int** a,**int** b) function which returns the absolute value between two variables is also included in the pseudocode.

2.1 Recursive Algorithm Pseudocode

```
FindPathRec(A, i, j)
 1  if i == 0 and j == 0
 2        return 0
 3  if i == 0
 4        return FindPathRec(A, i, j − 1) + abso(A[i][j − 1], A[i][j])
 5  if j == 0
 6        return FindPathRec(A, i − 1, j) + abso(A[i − 1][j], A[i][j])
 7  a = FindPathRec(A, i − 1, j) + abso(A[i − 1][j], A[i][j])
 8  b = FindPathRec(A, i, j − 1) + abso(A[i][j − 1], A[i][j])
 9  if a < b
10        return a
11  else
12        return b
```

This is the first part of the recursive aproach.

The FindPathRec function returns the minimum cost path for a single location.To be able to print all the locations we also need the printalg1 function.

```
PRINTALG1(A)
1   i = 0
2   j = 0
3   Print the top left location of the path
4   while i ≤A.no_rows-1 and j ≤A.no_cols-1
5       if i ==A.no_rows-1 and j ==A.no_cols-1
6           break
7       if j ==A.no_cols-1
8           i = i + 1
9           Print Current location
10      elseif i ==A.no_rows-1
11          j = j + 1
12          Print Current location
13      elseif FINDPATHREC(A, i + 1, j) < FINDPATHREC(A, i, j + 1)
14          i = i + 1
15          Print Current location
16      else
17              j = j + 1
18              Print Current location
```

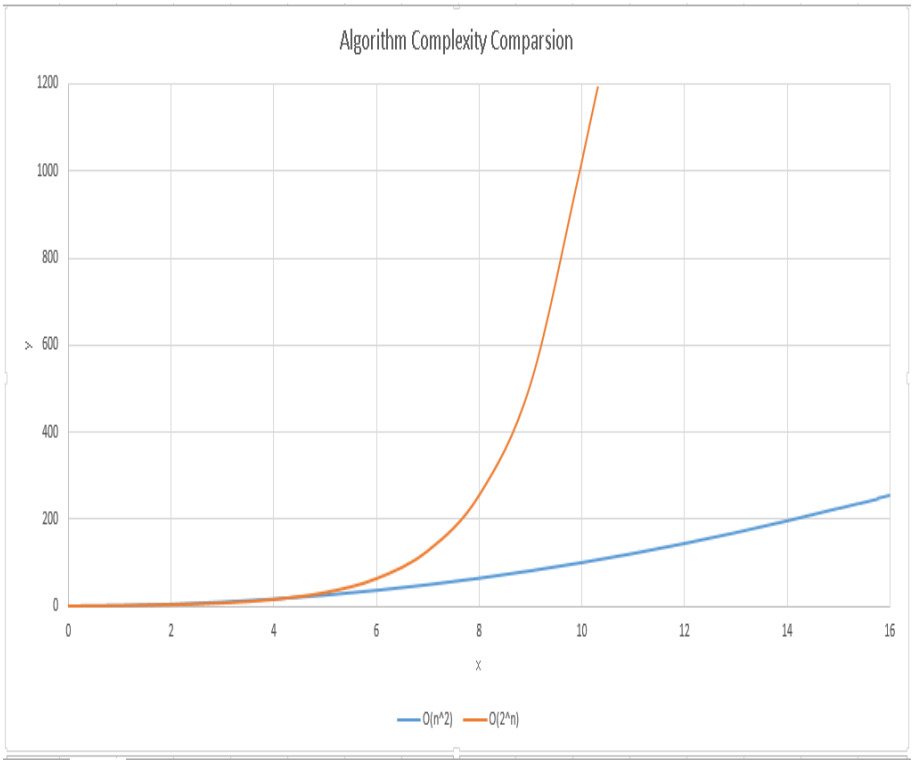
This function prints all the elements of the path by calling the FindPathRec function in a succesive way starting from the top left corner of the grid to the bottom left one .

On the next page we have the Dynamic Programming pseudocode for implementing the algorithm.

2.2 Dynamic Programming aproach

```
FindPathDP(A)
 1  for i = 1 to A.no_rows − 1
 2        B[i][0] = B[i − 1][0] + abso(A[i − 1][0], A[i][0])
 3        B[0][i] = B[0][i − 1] + abso(A[0][i − 1], A[i][0])
 4  for i = 1 to A.no_rows − 1
 5        for j = 1 to A.no_cols − 1
 6              m = B[i − 1][j] + abso(A[i − 1][j], A[i][j])
 7              n = B[i][j − 1] + abso(A[i][j − 1], A[i][j])
 8              if m < n
 9                    B[i][j] = m
10              else
11                    B[i][j] = n
12  i = 0
13  j = 0
14  Print the top left Location
15  while i ≤A.no_rows-1 and j ≤A.no_cols-1
16        if i ==A.no_rows-1 and j ==A.no-cols-1
17              break
18        if j ==A.no_cols-1
19              i = i + 1
20              Print Current location
21        elseif i ==A.no_rows-1
22              j = j + 1
23              Print Current location
24        elseif B[i + 1][j] < B[i][j + 1]
25              i = i + 1
26              Print Current location
27        else
28                    j = j + 1
29                    Print Current location
```

In this graph we can see the comparison of the time complexity of the two algorithms



3 Experimental data

In this section it is presented how the experimental values are generated for our problem.

To create our grid we need to randomly generate a matrix of dimension nxn. The dimension of the matrix is going to be a randomly generated integer represented by the variable $matrix.no\_rows$.

Then we can generate the elements of the grid which have random values between $[0, 10000]$.

Because the recursive algorithm has an exponential time complexity,generating values for both algorithms comes in with a few restrictions.In that case, to avoid getting high time consuming experimental data for the recursive algorithm,we are going to create a matrix that has a dimension of maximum 10x10.

The second algorithm will be run on the same set of values as the first one so we can have a comparsion of running time,but also it will run on a separate set of data that is much larger.

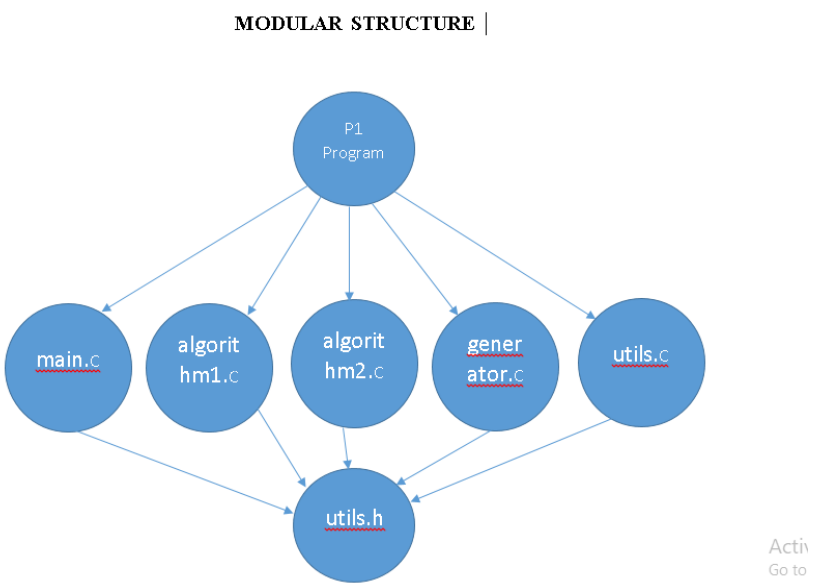
Even then,a restriction of a matrix of a maximum dimension of 300x300 needs to be made because we need to make sure that the file in which we put the experimental values doesn't get too large.

This is essential for testing the algorithms side by side but also for testing on non trivial sets of data.

4 Experimental Application Structure

4.1 High Level Application Structure

In this section we review the modular structure of our application on a high level hierarchy.



Our program uses a modular structure so that it can be easily divided in different work files.

All the ".c" files are part of the project structure and are interconnected by the use utils.h in which all the functions used in the project are located.

4.2 Description of input data

Our input data is relevant to what the algorithm needs to work and is composed from multiple data sets which contain:

- The dimension of the grid represented
  by $matrix.no\_rows$
  $1 \leq$matrix.no_rows≤ 10 Used for the comparison of both types of algorithms
  $1 \leq$matrix.no_rows≤ 300 Used for the Dynamic Programming algorithm
- $matrix.no\_rows^2$ elements which represent the value of each location of the grid.
  Each element is represented by
  $0 \leq matrix[row\_index][col\_index] \leq 10000$

4.3 Description of output data/results

In the results section it is presented how the output data sets of the problem are being made.

Each one of the two algorithms outputs the minimum cost path,starting from the top left corner of the matrix to the bottom right one.

Each location of the path is printed in order as $Mincost[row\_index][col\_index$

In the end a row/multiple rows of such locations are printed on the terminal.

As a method of ease when reading the result a restriction has been implemented for grids with many elements.Only the first and last 10 locations are printed,the other values being replaced by three dots "...".

4.4 List of Application modules

1. The algorithm1.c module

   Here we have implemented the first algorithm using the recursive aproach.

   The module consists of two functions:

   - FindPathRec (which computes the minimum path up to a certain element)
   - printalg1 (which calls the first function in order to print all the minimum cost locations.

2. The algorithm2.c module

   In this module we have implemented the second algorithm based on the Dynamic Programming aproach.

   The module consists of one function:

   - FindPathDP (which computes prints the minimum
     path of the grid).

3. The generator.c module

   This is the module used for the generation of data sets.

   The module consists of one function:

   - *create_ematrix* (which generates a matrix with random elements and dimension).
     It also puts the data sets in a file for use in the python implementaion.
     The functions for both of the algorithms are called here as well on different data sets.

4. The utils.c module

   In this module there are multiple utility functions used in the implementation of the application.

5. The main.c module

   This the main function in which the other modules are called.

   main.c also contains a switch function which is used for easy access of other modules and convenience in comparing the two algorithms as many times as the user wants.

6. The utils.h header

   In this module we have the headers for all the functions used in the application ,the file utils.h being included in all the other .c extension files.

4.5 List of all procedures/functions used in the application

1. algorithm1.c

   - **int** FindPathRec
     (struct *grid_matrix* matrix,int *row_index*,int *col_index*)
     This function is used for the recursive algorithm.
     It computes the minimum cost necesarry to reach a certain location.The function works its way back the grid calculating each time what is the minimum sum of absolute values to location A[0][0]. The path it can take is limited by two moves up and to the left.
     If the current location calculated is either on the first row of the matrix or on the first column it will have only one path to take so it will go straight to the first element.
     Parameters:
     - The first parameter is **struct** *grid_matrix* **matrix**
       This is the matrix which holds the randomly generated values. It has a struct form so we can easily allocate values and free the matrix.
     - The second parameter is **int** *row_index*
       It holds the row of the location.
     - The third parameter is **int** *row_column*
       It holds the column position of the location.

     The function returns the minimum cost value up until the location defined by the row and column indexes.

- **void** printalg1(struct *grid_matrix* matrix)

  The role of this function is to start printing the locations of the shortest path on the grid.

  It starts with the first location of the grid and then check to see which of the two options:going to the right or down has the minimum cost value.

  If it reaches the last column or last row it will only count one of those two directions.

  The function stops once it reaches the bottom right location.

2. algorithm2.c

   This module only contains one funtion:

   - int FindPathDP(struct *grid_matrix* matrix)

     This function represents the second algorithm implemented and it uses a Dynamic Programming aproach.

     The difference between this function and the last one is that we use a newly created matrix (MinCost) to save the minimum cost path to each location.

     This way we don't have to call the function for each element when printing.The printing procedure stays the same,only we start from the first element of the new matrix and search the minimum path possible.

     The only parameter used is the randomly generated matrix which is used to create the new one.

3. generator.c

   This module contains only one function which has two roles:

   - Creating a randomly generated matrix
   - Executing the two algorithms on this particular set of data.

   The function **void** *create_matrix*(**int** *grid_dimension*) has only one parameter.

   This parameter determines if the geneator creates a matrix of maximum dimension of 10x10 or a one with a maximum of 300x300.

   These sets of data are relevant for comparing the two algorithms as the first one can give a decent running time only if the matrix dimension is around 10x10.

   So depending on what algorithm we want to show this function is going to call the other two functions from algorithm1.c and algoritm2.c

4. utils.c

   This module is where our utility functions are located.

   - The first function is:
     int min(**int** a,**int** b)

     It returns the minimum value between the two parameters.

     This function is used while finding the minimum of two consecutive locations.

- The second function is:
  int abso(**int** a,**int** b)
  It returns the absolute value between two elements.
  This function is used to get the absolute value between two consecutive locations.
- The third function is:
  void *set_value* which has 4 parameters:
  - **struct** *grid_matrix* matrix
  - **int** *row_index*
  - **int** *columm_index*
  - **int** *element_value*

  This function is used to set the value of each element in the matrix.
- The fourth function is:
  void *get_value* which has 3 parameters:
  - **struct** *grid_matrix* matrix
  - **int** *row_index*
  - **int** *columm_index*

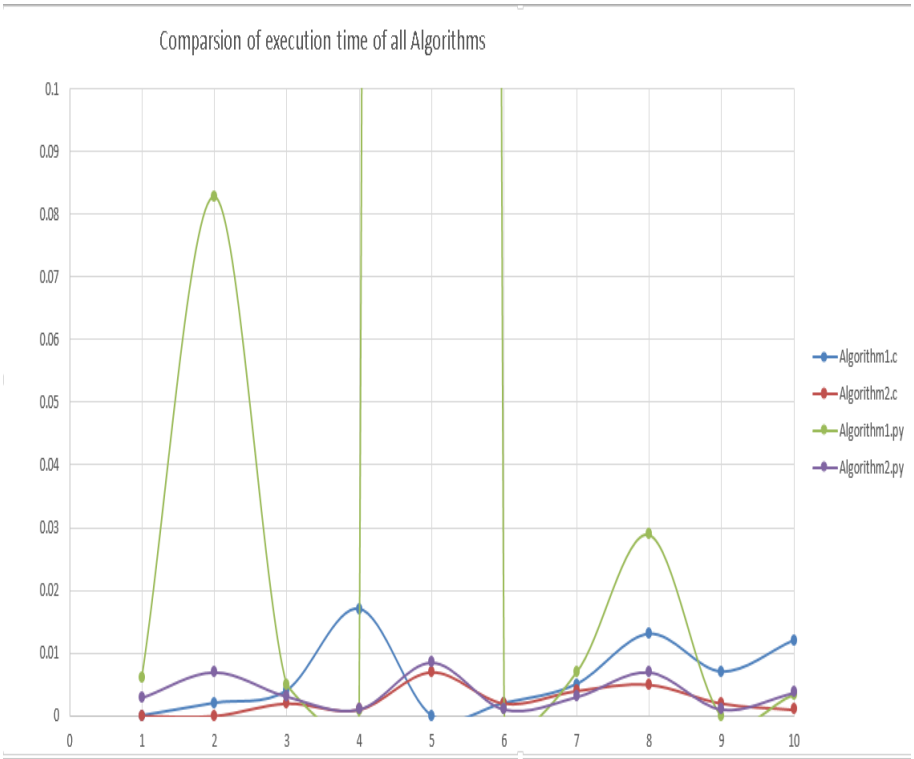
  This function is used to get the value of each element in the matrix.
- The last function in this module is:
  void *print_matrix*(**struct** *grid_matrix* matrix)
  This function has only one parameter,a matrix of type struct.
  The role of this function is to print the values in the text file.

5. utils.h -Contains all the headers for the functions from above

4.6 Results and Conclusions

In this section we will conduct two sets of tests to evaluate how out algorithms differ in running time.

1. The first test will be made on 10 different data sets comparing the two algorithms in both C and Python language.



On three of the algorithms the results are very similar.

One massive difference is on the Algorithm1.py.

We can already see that the first algorithm has a higher running time even on the C implementation,and when impemented in python the differences are even higher.

Algorithm1.c	Algorithm2.c	Algorithm1.py	Algorithm2.py
0	0	0.006118	0.002859
0.002	0	0.08292	0.006981
0.004	0.002	0.004983	0.003073
0.017	0.001	0.000915	0.000997
0	0.007	4.925523	0.008538
0.002	0.002	0.000997	0.000996
0.005	0.004	0.006982	0.002991
0.013	0.005	0.028895	0.006978
0.007	0.002	0	0.000997
0.012	0.001	0.003332	0.003671
Average time	Average time	Average time	Average time
0.0062	0.0024	0.5060665	0.0038081

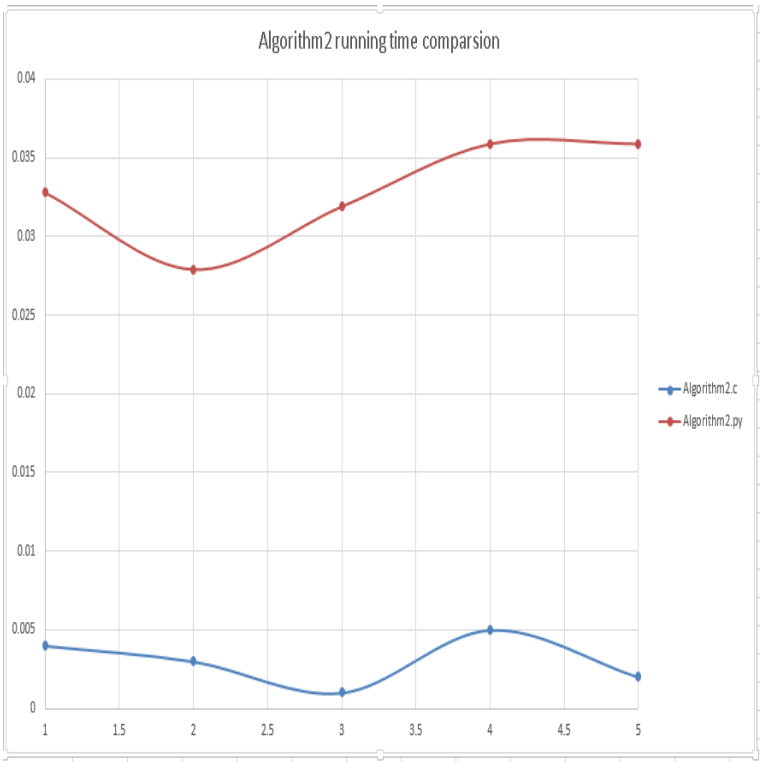
Here is the table of results.

We can see that the fifth set of data in python is way higher than the rest.

The average time of each implementation is also shown,the dynamic programming algorithm implemented in C(Algorithm2.c) having the best running time average.

The main part about these sets of data is that they were run on a matrix of a maximum dimension of 10x10 because,otherwise the first algorithm would take too long to run because it has a exponential running time.

2. The second set of tests will be made on a larger matrix,but the results will only be computed for the implementations of the second algorithm.



In this graph we can see how the Dynamic Programming algorithm(Alogrithm2) runs in both C and Python.

the algorithm implemented in C is much faster than the one in python.

The diffence in running time comes from the fact the Python is an interpreted language,where as C is a compiled language.

1	Algorithm2.c	Algorithm2.py
2	0.004	0.0328
3	0.003	0.0279
4	0.001	0.0319
5	0.005	0.0359
6	0.002	0.0359
7	Average Time	Average Time
8	0.003	0.03288

In this table we can see what values we got from using 5 data sets.

The average time on the algorithm implemented in C is much lower than the algorithm implemented in python.

Conclusions

After implementing both algorithms and comparing them in both C and Python language we can tell the differences and draw conclusions.

Most importantly is how much time the algorithms take to run.

In this case, it was a challenging task to compare the two algorithms,because of how they behave on large sets of data.

Due to how the Recursive algorithm has an exponential running time it was hard to compare on a matrix that is larger than 10x10.

Regarding this problem,I chose to review both algorithms on a smaller scale,and show only how the Dynamic Programming algorithm behaves on sets of bigger matrices.

After the tests,results showed that even though all the algorithms produce the same output values,the Dynamic programming algorithm implemented in C language has the fastest running time.

4.7 References

(a) GeeksforGeeks.com

(b) StackOverflow.com

(c) Dynamic Programming-Wiki