

18CSC204J Lab Assignment for CLA-P4

Question :

Maze game is a well-known problem, where we are given a grid of 0's and 1's, 0's correspond to a place that can be traversed, and 1 corresponds to a place that cannot be traversed (i.e. a wall); the problem is to find a path from bottom left corner of grid to top right corner; immediate right, immediate left, immediate up and immediate down only are possible (no diagonal moves). We consider a variant of the maze problem where a cost is attached to visiting each location in the maze, and the problem is to find a path of least cost through the maze.

Goal : To find there is a path from bottom left corner of grid to top right corner.

Algorithm Used to solve this : Backtracking (used DFS to implement)

The basic idea is to traversal every possible way from given source cell to the destination cell.If we get any walls just stop over.If we get any out of bound indexes then also stop.whenever we get destination keep track of the path.After completion of all the possible ways just print the result.

Why Backtracking?

For the given problem the solution can be done by DijkstraAlgorithm to find shortest path.But this fails when we have negative cost.Now Backtracking this works fine for bothwhen the cost of the cell having negative and postive values.So we decided to solve this problem using Backtracking.

Source Code :

```
#include<bits/stdc++.h>
using namespace std;

int row,column;

void PrintGrid(int row,int column,vector<vector<int> >grid){
    for(int i = 0; i < row; i++){
        for(int j = 0; j < column; j++){
            cout << grid[i][j] << " ";
        }
        cout << endl;
    }
}

void isValidPath(int i,int j,vector<vector<int> > &grid,string p,vector<string> &Path,vector<vector<int> > &cost,vector<int> &temp,int PathCost,vector<pair<string,int> > &Values){
    // we encounter out of bound then simply stop there
    if(i < 0 || i >= row || j < 0 || j >= column){
        return;
    }
    // if we encounter a wall or already visited cell stop there
    if(grid[i][j] == 1 || grid[i][j] == INT_MIN){
        return;
    }
    // If we reach top right corner (Destination)
    else if(i == 0 && j == column - 1){
        Path.push_back(p);
        temp.push_back(PathCost);
        Values.push_back(make_pair(p,PathCost));
        return ;
    }else{
        // PossiblePaths(i,j,grid,p,Path,cost,temp,PathCost,Values);
        grid[i][j] = INT_MIN;
        isValidPath(i+1,j,grid,p+'D',Path,cost,temp,PathCost+cost[i][j],Values);
        isValidPath(i-1,j,grid,p+'U',Path,cost,temp,PathCost+cost[i][j],Values);
        isValidPath(i,j+1,grid,p+'R',Path,cost,temp,PathCost+cost[i][j],Values);
        isValidPath(i,j-1,grid,p+'L',Path,cost,temp,PathCost+cost[i][j],Values);
        grid[i][j] = 0;
    }
}

void PrintResults(vector<int> temp,vector<pair<string,int> > Values){
    if(temp.size() == 0){
        cout<<"There is no path from Bottom Left Corner to Right Most Corner";
    }
}
```

```

else{
    cout << endl;
    cout << "Total Number of paths possible for the given grid is " << temp.size() << endl
;

    cout << endl;
    cout << "-----\n";
    cout << "All possible Paths -----> There cost as follows\n";
    cout << "-----\n";
    string minPath;
    int min = INT_MAX;

    for(auto i : Values){
        cout << i.first << "\t " << i.second;
        if(min > i.second){
            min = i.second;
            minPath = i.first;
        }
        cout << endl;
    }
    cout << "-----\n";
    cout << "Minimum possible cost required is " << min << "\t" << "( "<< minPath << " )"
<< endl;
    cout << "-----\n";
}
}

void GetPath(vector<vector<int> > &grid,vector<vector<int> > &cost){
    //Path vector stores the path from source to destination
    vector<int>temp;
    vector<pair<string,int> > Values;
    vector<string>Path;
    //temp vector stores the cost of the path
    int PathCost=0;
    //PathCost TO KEEP A TRACK OF THE PathCost FOR RESPECTIVE PATH
    isValidPath(column-1,0,grid,"",Path,cost,temp,PathCost, Values);
    PrintResults(temp,Values);
}

int main(){
    cout << "Enter number of rows and columns respectively\n";
    cin >> row >> column;
    vector<vector<int> > grid(row,vector<int>(column,0)),cost(row,vector<int>(column,0));
    cout << "Enter the positions of each cell in the grid\n";
    // Taking grid matrix as input
    for(int i = 0; i < row; i++){
        for(int j = 0; j < column; j++){
            cin >> grid[i][j];
        }
    }
    cout << "Enter the Cost of each cell in the grid\n";
    // Taking cost matrix as input
    for(int i = 0; i < row; i++){
        for(int j = 0; j < column; j++){
            cin >> cost[i][j];
        }
    }
    cout << "Given positions of grid is :\n";
    PrintGrid(row,column,grid);
    cout << "Given Cost of each cell is :\n";
    PrintGrid(row,column,cost);
    GetPath(grid,cost);
}

```

Main Functions Used And their Description :

1) GetPath :

In this function we declared required data structure. And also called another function isValidPath which plays major role in this code.

2) isValidPath :

In this function we called four recursive calls to find there exist a path from that particular cell to the destination cell. Also there are two bases cases they are if the particular cell is out of grid then we simply stop there, and also if we reach our destination cell then we have to store the path.

3) PrintGrid:

This takes one 2D vector as a parameter and simply print the 2D vector

4) PrintResults :

This function helps to print the results. Which Takes 2 major parameters one vector which is having costs of all paths and another is vector of pairs which stores path and their cost.

Time Complexity :

Time complexity for this approach is 3^{n^2}

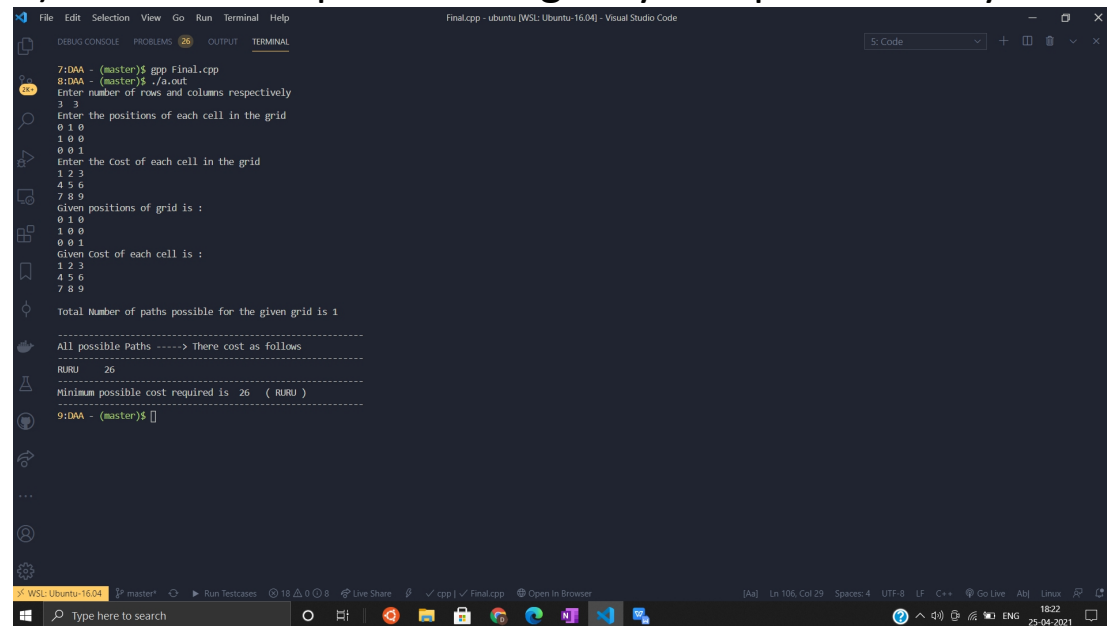
there are n^2 cells from each cell there are 3 unvisited neighbouring cells.

Space Complexity :

Space Complexity is also 3^{n^2}
We have to store the path for each recursive call.

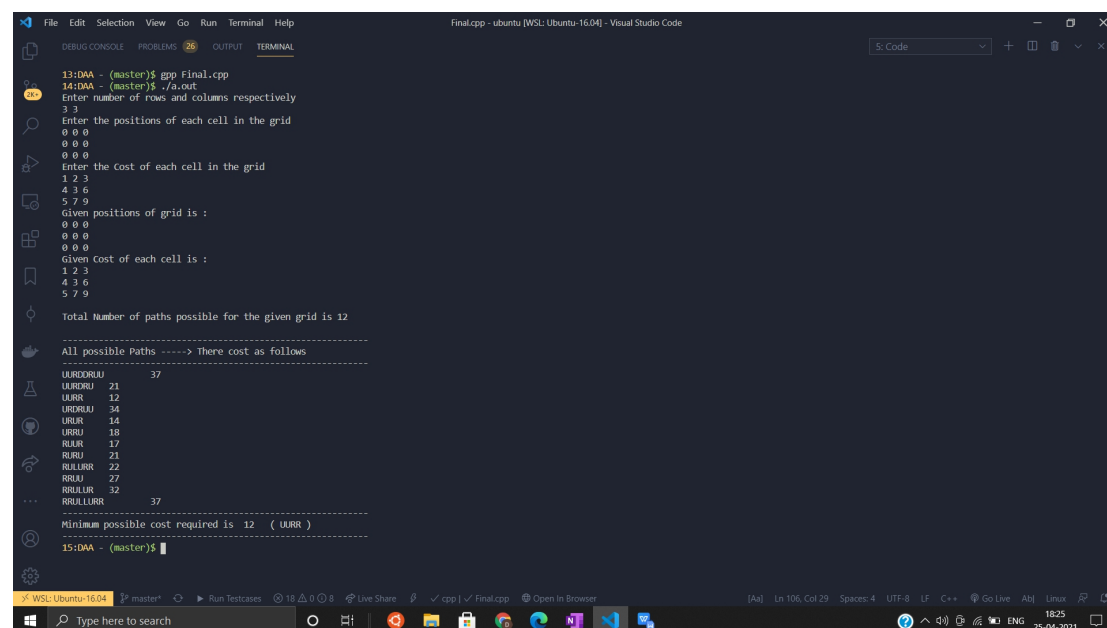
Output :

1) When the input is having only one possible way



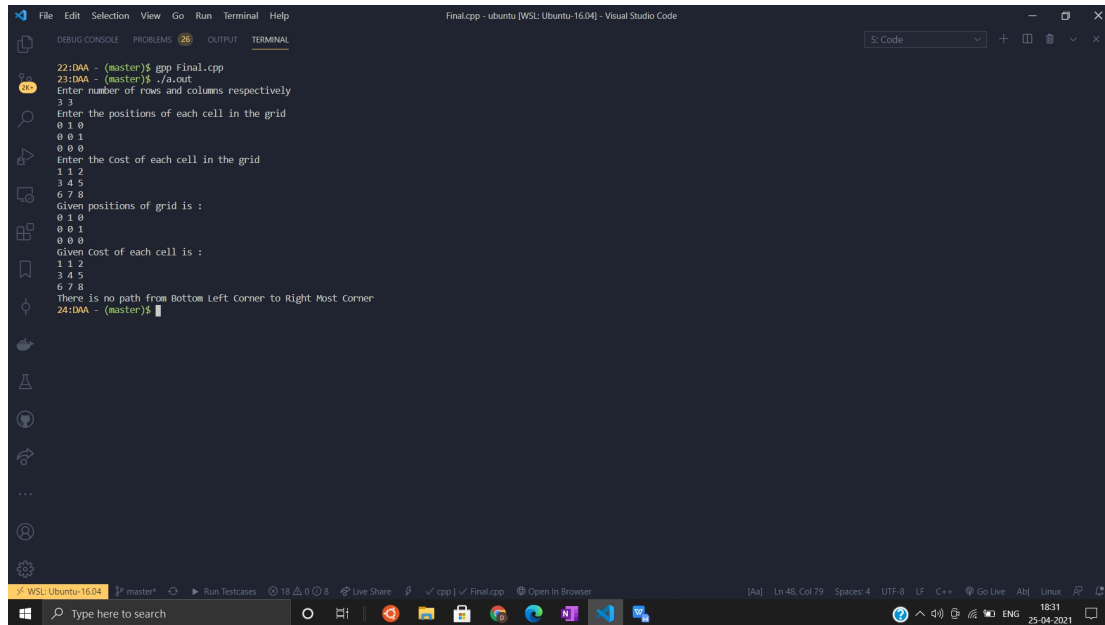
```
7:DAA - (master)$ g++ Final.cpp
8:DAA - (master)$ ./a.out
Enter number of rows and columns respectively
3 3
Enter the positions of each cell in the grid
0 1 0
1 0 0
0 0 1
Enter the Cost of each cell in the grid
1 2 3
4 5 6
7 8 9
Given positions of grid is :
0 1 0
1 0 0
0 0 1
Given Cost of each cell is :
1 2 3
4 5 6
7 8 9
Total Number of paths possible for the given grid is 1
-----
All possible Paths ----> There cost as follows
RURU    26
-----
Minimum possible cost required is  26  ( RURU )
9:DAA - (master)$ []
```

2) When input is having multiple ways:



```
13:DAA - (master)$ g++ Final.cpp
14:DAA - (master)$ ./a.out
Enter number of rows and columns respectively
3 3
Enter the positions of each cell in the grid
0 0 0
0 0 0
0 0 0
Enter the Cost of each cell in the grid
1 2 3
4 3 6
5 7 9
Given positions of grid is :
0 0 0
0 0 0
0 0 0
Given Cost of each cell is :
1 2 3
4 3 6
5 7 9
Total Number of paths possible for the given grid is 12
-----
All possible Paths ----> There cost as follows
UURDDRUU    37
UURDRU    21
UURR    12
URDRUU    34
URUR    14
URRU    18
RUUR    17
RURU    21
RULLUR    22
RRUU    27
RRULUR    32
RRULLURR    37
-----
Minimum possible cost required is  12  ( UURR )
15:DAA - (master)$ []
```

3) Input having no possible way:



```
22:DAK - (master)$ g++ Final.cpp
23:DAK - (master)$ ./a.out
Enter number of rows and columns respectively
3 3
Enter the positions of each cell in the grid
0 1 0
0 0 1
0 0 0
Enter the cost of each cell in the grid
1 1 2
3 4 5
6 7 8
Given positions of grid is :
0 1 0
0 0 1
0 0 0
Given Cost of each cell is :
1 1 2
3 4 5
6 7 8
There is no path from Bottom Left Corner to Right Most Corner
24:DAK - (master)$
```

Done by
Manikanta (RA1911026010033)
Chetan (RA1911026010035)