18CSC204J Lab Assignment (CLA4)

Name: YASH TIWARI - RA1911026010073
Name: Soumyadeep Mukherjee - RA1911026010078

Question:

Maze game is a well-known problem, where we are given a grid of 0's and 1's, 0's corresponds to a place that can be traversed, and 1 corresponds to a place that cannot be traversed (i.e. a wall or barrier); 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 (positive value) or profit (negative value) is attached to visiting each location in the maze, and the problem is to find a path of least cost through the maze.

You may solve the problem after imposing/relaxing other restrictions on the above problem on

- Values of cost/profit (but not same cost/profit for all traversable cells in the grid)
- Moves possible (but you cannot trivialize the problem by making the grid linear or partly linear in any way)
- No. of destinations possible

Choose the most efficient algorithm possible for your specific case.

Hints: Convert the maze to a weighted graph G (V, E). Each location (i, j) in the maze corresponds to a node in the graph. The problem can have multiple solutions. Students can use any design technique such as greedy method, backtracking, dynamic programming. Students can choose their own conditions, positive or negative costs for the graph.

APPROACH:

- Represent the maze with a 2D matrix. Start from top left square, cell (0, 0). Try to move to the square on the right, if it's not possible to move to the square on the right because there is obstacle, try to move down. Keep doing this until we reach to the bottom right square, cell (N-1, N-1), where N is the size of the maze. On making a move to valid cell (row, column) mark the value in solutionMatrix[row][column] and increment the cost by one.
- If we reach a dead end, a cell(row, column)from where we can neither go right nor down then backtrack(mark the value in solutionMatrix[row][column] = 0 and return false).
- Similarly in this way all the possible paths are found and the path which gives minimum cost is displayed.

DESIGN TECHNIQUE AND REASON:-

For this problem we are using **BACKTRACKING** method for building the solution because of the advantage of getting accurate solution and on assumption that maze size will not be too large . Backtracking is algorithm-technique for solving problem recursively by trying to build a solution incrementally, one piece at a time , removing that fail to satisfy the constraints of problem.

Algorithm:

1. Create a solution matrix, initially filled with 0's.
2. Create a recursive function, which takes initial matrix, output matrix and position of M(i, j).
3. If the position is out of the matrix or the position is not valid then return.
4. Mark the position output[i][j] as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.
5. Recursively call for position (i+1, j) and (i, j+1).
6. Unmark position (i, j), i.e output[i][j] = 0.

Source Code:

Case-I:

```
#include <bits/stdc++.h>

#define MAX 5

using namespace std;



int cost[100];

int a=0;

string str[100];
```

```
bool isSafe(int row, int col, int m[][MAX],int n, bool visited[][MAX])
{
   if (row == -1 || row == n || col == -1 ||col == n || visited[row][col]|| m[row][col] == 1)
      return false;
   return true;
}



void printPathUtil(int row, int col, int m[][MAX],int n, string& path, vector<string>&possiblePaths,
bool visited[][MAX])
{
   if (row == -1 || row == n || col == -1|| col == n || visited[row][col]|| m[row][col] == 1)
      return;
   if (row == n - 1 && col == n - 1) {
      possiblePaths.push_back(path);
      return;
   }

   // Mark the cell as visited
   visited[row][col] = true;

   // Try for all the 4 directions (down, left,
   // right, up) in the given order to get the
   // paths in lexicographical order

   // Check if downward move is valid
   if (isSafe(row + 1, col, m, n, visited))
   {
      path.push_back('D');
      printPathUtil(row + 1, col, m, n,path, possiblePaths, visited);
      path.pop_back();
```

```
}

// Check if the left move is valid
if (isSafe(row, col - 1, m, n, visited))
{
    path.push_back('L');
    printPathUtil(row, col - 1, m, n,path, possiblePaths, visited);
    path.pop_back();

}

// Check if the right move is valid
if (isSafe(row, col + 1, m, n, visited))
{
    path.push_back('R');
    printPathUtil(row, col + 1, m, n,path, possiblePaths, visited);
    path.pop_back();

}

 // Check if the upper move is valid
if (isSafe(row - 1, col, m, n, visited))
{
    path.push_back('U');
    printPathUtil(row - 1, col, m, n,path, possiblePaths, visited);
    path.pop_back();

}

// Mark the cell as unvisited for
```

```
    // other possible paths
    visited[row][col] = false;
    }

// Function to store and print
// all the valid paths
void printPath(int m[MAX][MAX], int n)
{
    // vector to store all the possible paths
    vector<string> possiblePaths;
    string path;
    bool visited[n][MAX];
    memset(visited, false, sizeof(visited));

    // Call the utility function to
    // find the valid paths
    printPathUtil(0, 0, m, n, path,possiblePaths, visited);
    int p=1,j;



    // Print all possible paths
    for (int i = 0; i < possiblePaths.size(); i++)
      {
          p=1;
          str[a]="";
          cost[a]=0;
          str[a]=possiblePaths[i];
          for(j=0;j<possiblePaths[i].size();j++)
          {
              cost[a]+=p;
              p++;
```

```
            }
            a++;
        }

}

// Driver code
int main()
{
    int m[MAX][MAX] = { { 0, 0, 0, 0, 0 },
                        { 0, 1, 1, 0, 0 },
                        { 0, 1, 1, 0, 0 },
                        { 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0 } };
    int n = sizeof(m) / sizeof(m[0]);
    printPath(m, n);
    int min_cost=cost[0],pos=0,i,j;
for(i=0;i<5;i++)
    {
        for(j=0;j<5;j++)
        {
            cout<<m[i][j]<<" ";
        }
        cout<<endl;
    }
    for(i=1;i<a;i++)
    {
        if(cost[i]<min_cost)
        {
        min_cost=cost[i];
        pos=i;
```

```
        }
    }
    if(min_cost!=0)
        cout<<"Cost: "<<min_cost<<endl<<"Path "<<str[pos];
    else
        cout<<"No traversable path";
return 0;
}
```

<u>Output:</u>

Cost: 8

Path DDDDRRRR

<u>Test case(explanation)-</u>

{ { 0, 0, 0, 0, 0 },

{ 0, 1, 1, 0, 0 },

{ 0, 1, 1, 0, 0 },

{ 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0 } };

After marking possible paths

{ { 0, 0, 0, 0, 0 },

{ D, 1, 1, 0, 0 },

{ D, 1, 1, 0, 0 },

{ D, 0, 0, 0, 0 },

{ D R, R, R, R } };

So as the marked path which shortest is DDDDRRRR

As 8 steps are taken to each the final point so the total cost = 8 (number of characters)*1

=8

Time complexity Is O(n^2)

Case-II:

```
#include <bits/stdc++.h>

#define MAX 5

using namespace std;



int cost[100];

int a=0;

string str[100];

bool isSafe(int row, int col, int m[][MAX],int n, bool visited[][MAX])

{

   if (row == -1 || row == n || col == -1 ||col == n || visited[row][col]|| m[row][col] == 1)

       return false;

   return true;

}



void printPathUtil(int row, int col, int m[][MAX],int n, string& path, vector<string>&possiblePaths, bool visited[][MAX])
```

```
{
    if (row == -1 || row == n || col == -1|| col == n || visited[row][col]|| m[row][col] == 1)
        return;
    if (row == n - 1 && col == n - 1) {
        possiblePaths.push_back(path);
        return;
    }

    // Mark the cell as visited
    visited[row][col] = true;

    // Try for all the 4 directions (down, left,
    // right, up) in the given order to get the
    // paths in lexicographical order

    // Check if downward move is valid
    if (isSafe(row + 1, col, m, n, visited))
    {
        path.push_back('D');
        printPathUtil(row + 1, col, m, n,path, possiblePaths, visited);
        path.pop_back();

    }

    // Check if the left move is valid
    if (isSafe(row, col - 1, m, n, visited))
    {
        path.push_back('L');
        printPathUtil(row, col - 1, m, n,path, possiblePaths, visited);
        path.pop_back();
```

```
    }

    // Check if the right move is valid
    if (isSafe(row, col + 1, m, n, visited))
    {
        path.push_back('R');
        printPathUtil(row, col + 1, m, n,path, possiblePaths, visited);
        path.pop_back();

    }

     // Check if the upper move is valid
    if (isSafe(row - 1, col, m, n, visited))
    {
        path.push_back('U');
        printPathUtil(row - 1, col, m, n,path, possiblePaths, visited);
        path.pop_back();

    }

    // Mark the cell as unvisited for
    // other possible paths
    visited[row][col] = false;
    }

// Function to store and print
// all the valid paths
void printPath(int m[MAX][MAX], int n)
{
    // vector to store all the possible paths
    vector<string> possiblePaths;
```

```
    string path;
    bool visited[n][MAX];
    memset(visited, false, sizeof(visited));

    // Call the utility function to
    // find the valid paths
    printPathUtil(0, 0, m, n, path,possiblePaths, visited);
    int p=1,j;



    // Print all possible paths
    for (int i = 0; i < possiblePaths.size(); i++)
      {
          p=1;
          str[a]="";
          cost[a]=0;
          str[a]=possiblePaths[i];
          for(j=0;j<possiblePaths[i].size();j++)
          {
             cost[a]+=p;
             p++;
          }
          a++;
      }

}

// Driver code
int main()
{
    int m[MAX][MAX] = { { 0, 0, 0, 0, 0 },
```

```
                    { 1, 1, 1, 0, 0 },
                    { 0, 1, 1, 0, 0 },
                    { 0, 0, 0, 0, 0 },
                    { 1, 0, 0, 0, 1 } };
    int n = sizeof(m) / sizeof(m[0]);
    printPath(m, n);
    int min_cost=cost[0],pos=0,i,j;
for(i=0;i<5;i++)
    {
        for(j=0;j<5;j++)
        {
            cout<<m[i][j]<<" ";
        }
        cout<<endl;
    }
    for(i=1;i<a;i++)
    {
        if(cost[i]<min_cost)
        {
        min_cost=cost[i];
        pos=i;
        }
    }
    if(min_cost!=0)
        cout<<"Cost: "<<min_cost<<endl<<"Path "<<str[pos];
    else
        cout<<"No traversable path";
return 0;
}
```

Output:

No traversable path.

Test case-2(explanation)-

{ { 0, 0, 0, 0, 0 },
{ 1, 1, 1, 0, 0 },
{ 0, 1, 1, 0, 0 },
{ 0, 0, 0, 0, 0 },
{ 1, 0, 0, 0, 1 } };

As in the given matrix that destination is marked as 1 so it can be reached, so the output is no traversable path

Time Complexity Analysis

Code does have many function and do work on basic idea of backtracking (calling function recursively till we get desired result)

Mostly condition statement are used but there are two loops running in the code which makes time complexity as O(n^2)

Space complexity of the code is O(N^2) as well as we are storing cost and the path.

Result:

Hence, we have found an optimal path to traverse through the maze and reach the destination from the source. WITH TIME COMPLEXITY OF 0(N^2)