(class2) 22-03-2023

Backtracking

Ques Rat in a maze.

This an example of the problem that can be solved using Backtracking.

→ A maze is given as n*n binary matrix of blocks where source block is the upper left most block, i.e., maze[0][0] and the destination block is bottom right most block, i.e., maze[n-1][n-1].

→ A rat starts from source & has to reach the destination. The rat can move only 4 directions, i.e., Left, Right, Down, Up.

source

	0	1	2	3
0	1	1	1	1
1	0	0	1	0
2	1	0	1	1
3	0	0	0	1

→ destination

Here, 0 means that the cell is blocked, and the rat cannot enter. 1 means that the cell is open, and the rat enters easily.

For example :-

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	0	0
3	0	1	1	1

(0, 0) is a starting position of a rat.

(3, 3) is the ending position of the rat.

* we have to print all the existing paths to reach the given destination.

The constrains are :-

i) The rat can only move on the places where 1 is written, because 1 means that the cell is open & the rat can enter, and 0 means the rat can not enter, i.e., the path is blocked.

ii) The possible movements are up, Down, Left, Right.

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	0	0
3	0	1	1	1

Initially, we are at (0, 0) position. and the possible movements are, Up, Down, Left, Right.

not possible

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	0	0
3	0	1	1	1

From (0, 0) we can't go in upward direction, because its out of bound.

~~Up~~, Down, Left, Right

Now, we check for Down direction,

possible

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	0	0
3	0	1	1	1

From (0, 0), we can go in downward direction.

So, we go down & store the path in a string "D".

Now, we are at (1,0) position, & the possible movements are, up, Down, Left, Right.

possible

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	0	0
3	0	1	1	1

From (1,0), we can go upward direction again, but if we repeatedly do the same process then we stuck in an Infinite LOOP.

The solution to this problem is, we maintain a visited boolean array, whenever we visit a place we mark that place as visited, i.e., "true". So that we will not visit it again.

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	0	0
3	0	1	1	1

visited array

	0	1	2	3
0	T	F	F	F
1	F	F	F	F
2	F	F	F	F
3	F	F	F	F

Initially, we are at (0, 0) position, and the possible movements are, up, Down, Left, Right.

not possible

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	0	0
3	0	1	1	1

Now, we check for Downward direction,

possible

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	1	0	0
3	0	1	1	1

From (0,0), we can go downward direction, i.e., (1, 0)
So, we go down & store the path in the string "D", and along with that, we mark (1, 0) as visited in the visited array.

	0	1	2	3
0	1↓	0	0	0
1	1↓	1	0	0
2	1	1	0	0
3	0	1	1	1

mark this as visited

	0	1	2	3
0	T	F	F	F
1	T	F	F	F
2	F	F	F	F
3	F	F	F	F

By doing the same approach we find our all paths :-

"DDRDRR" & "DRDDRR"

what is the Base case ?

The base case is when we reach the destination, i.e:, we have found one solution & then return to explore other solutions & while returning back we need to make the corresponding position in the visited array as False, & this the main Backtracking line.

Important :-

- Down → (i+1, j)
- Left → (i, j-1)
- Right → (i, j+1)
- Up → (i-1, j)

	j-1	j	j+1
i-1		↑	
i	←	(i,j)	→
i+1		↓	

code:-

```
#include<bits/stdc++.h>
using namespace std;
bool isSafe( int x, int y, int rows,
        int columns, int arr[][4],
        vector<vector<bool>> &visited)
{
    if(((x >= 0 && x < rows) && (y>=0
        && y < columns)) &&
        (arr[x][y] == 1) &&
        (visited[x][y] == 0)
    ){
        return true;
    }
    else{
        return false;
    }
}
void solveMaze(int arr[][4], int rows,
        int columns, int i, int j,
        vector<vector<bool>> &visited,
        vector<string> &path, string
        output)
{
    //base case
    if(i == rows-1 && j == columns-1)
    {
        path.push_back(output);
        return;
    }
```

```
//down direction → (i+1, j)
if(isSafe(i+1, j, rows, columns,
                         arr, visited))
{
   visited[i+1][j] = true;
   solveMaze(arr, rows, columns,
          i+1, j, visited, path,
                         output + 'D');
   visited[i+1][j] = false;
}
//left direction → (i, j-1)
if(isSafe(i, j-1, rows, columns,
                      arr, visited))
{
   visited[i][j-1] = true;
   solveMaze(arr, rows, columns,
          i, j-1, visited, path,
                         output + 'L');
   visited[i][j-1] = false;
}
//right direction → (i, j+1)
if(isSafe(i, j+1, rows, columns,
                      arr, visited))
{
   visited[i][j+1] = treu;
   solveMaze(arr, rows, columns,
          i, j+1, visited, path,
                         output + 'R');
   visited[i][j+1] = false;
}
```

```
    // up direction → (i-1, j)
    if(issafe(i-1, j, rows, columns,
                        arr, visited))
    {
        visited[i-1][j] = true;
        solveMaze(arr, rows, columns,
                  i, j-1, visited, path,
                          output + 'U');
        visited[i-1][j] = false;
    }
}
int main()
{
    int arr[4][4] = { {1, 0, 0, 0},
                      {1, 1, 0, 0},
                      {1, 1, 0, 0},
                      {0, 1, 1, 1}
                    };
    int rows = 4;
    int columns = 4;
    vector<vector<bool>> visited(rows,
        vector<bool>(columns, false));
    visited[0][0] = true;
    vector<string> path;
    string output = "";
    solveMaze(arr, rows, columns, 0,
          0, visited, path, output);
    cout << "possible paths are:";
    for(auto i : path){
        cout << i << " ";
    }
    return 0;
}
```

Explanation of code :-

i) The base case, is that, when we reach at the bottom right corner, i.e., row-1 & column-1 cell, it means we find the path, then we store the path in the vector & return back to find more paths.

ii) isSafe() function, this function return true if going at the particular is safe or not.
There are 3 conditions to check the safe cell :-
a) The cell should be inbound, i.e., x >= 0 and x < rows, and y >= 0 and y < rows.
b) In our maze array, there should be 1 present at that cell, 1 means that the cell is open.
c) In our visited array, there should be false marked at that cell, it means this cell is not visited previously.
If all the above 3 conditions are true, then we go on that cell. Before going on that cell, we mark that cell as visited, & that call the recursive function for that cell.

iii) For each direction, solve one case for each direction & then recursive call will automatically handle all the cases for that direction.

iv) Save & print path, after every recursive call we save the direction in our output string & when the base case is hit we save that output string in our string vector & return back to find the next path.