CS 235 Data Structures and Algorithms

Project 9: 2D Maze Solver Using Recursion

# Purpose:

To achieve a deeper understanding of search algorithms using recursion.

## Key Reading:

Slides for lecture 10, on recursion.

Textbook 7.1 & 7.2: Recursion

Textbook 7.3: Recursive Search

Textbook 7.4: Recursive Problem Solving

Textbook 7.5: Backtracking

# Requirements:

Use the Visual Studio solution provided to complete your assignment. You only need to modify one file for this assignment, which is MySolver.cpp. Follow the algorithm outlined in the slide deck (discussed later).

# How this project works

Here is how your code will work:

1. Each test case will create a maze, and pass the information to the constructor for MySolver.
2. The test case will then call the SolveMaze() function (which you will implement, in MySolver.cpp). The first time SolveMaze() is called, the test case will pass in the starting location.
3. SolveMaze() will then recursively call itself, checking the adjacent cells in clockwise order: up, left, down, then right.

The MazeSolver class has the following utility functions to help you:

* **GetCell():** This method takes an X position and Y position, and returns a single character that represents what is in that location in the map:
  + ‘E’: this letter represents the exit. If you find this letter, you have solved the maze.
  + A space: this represents an empty cell. You can move into this spot. To move into this spot, recursively call SolveMaze at this location.
  + ‘+’: This character means that the cell has been marked as visited (you’ve already been here).
  + ‘#’: This is a wall.
* **MarkCell():** This method takes a Boolean value, plus an X,Y location. It will either mark a cell as visited, or un-mark it (set it back to blank).
* **CanMoveTo():** This method takes an X,Y location, and returns true if the cell is not marked or the cell is the exit. If the cell is marked or if the cell is a wall, or the cell is outside the bounds of the array, this method will return false.

# The algorithm

Search Rules:

1. Your search cannot move outside the boundaries of the array.
2. Your search can only move up, down, left, or right. Diagonal paths are not allowed.
3. If you have to backtrack, clean up your marks. When you’ve solved the maze there should be one solution, with no falsely-marked trails.

Here is an outline of the algorithm. Like all recursive algorithms, it starts by checking for a trivial case. If the trivial case hasn’t been reached, then it will divide and conquer:

1. Trivial case:
   1. Call GetCell() for the current x,y position. If the cell contains an ‘E’, then return TRUE. You’re done.
2. Divide and conquer:
   1. Call MarkCell() at the current x,y position, to mark the current location as visited.
   2. If CanMoveTo (x, y-1) == true then
      1. If SolveMaze(x, y-1) == true then
         1. Return true
   3. If CanMoveTo (x+1, y) == true then
      1. If SolveMaze(x+1, y) == true then
         1. return TRUE.
   4. If CanMoveTo(x, y+1) == true then
      1. If SolveMaze(x, y+1) == true then
         1. return TRUE.
   5. If CanMoveTo(x-1, y) == true then
      1. If SolveMaze(x-1, y) == true
         1. return TRUE.
   6. Call MarkCell at the current x,y position, to set the current cell back to blank.
   7. Return FALSE.

NOTE: if you’re having trouble understanding this pseudocode, you can see a C++ implementation of this algorithm in the slide deck.

The maze coordinate system looks upside down from what you might be used to seeing. The origin is in the upper left corner. The x axis starts at zero and increases as you move right. The y axis starts at zero and increases as you move downward. There are no negative x or y values. Here is a diagram, for clarity (in this case, the test case would start at cell (0, 3):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | # | # | # | # | # | # | # | # | # |
| 1 | # |  |  |  |  |  | # |  | # |
| 2 | # | # | # |  | # | # | # |  | # |
| 3 | ☺ |  |  |  | # |  |  |  | E |
| 4 | # | # | # |  | # |  | # | # | # |
| 5 | # |  |  |  |  |  |  |  | # |
| 6 | # | # | # | # | # | # | # | # | # |

# Restrictions:

Do not change the test cases, please.

This is a fairly simple lab. Please try and do your own work.

# Passing Off:

Function main contains a set of test cases. Run it until you are satisfied with the grade that it reports at the end.

When you’re ready to submit your work, put your solution into a zip file, and upload it to learningsuite.