Patrician's Path - SWDP

Problem Definition

Known Facts	User Requirements	Necessary Processing	Alternative Solutions
• Program navigates (i.e., traverses) a maze, searching for a solution/goal • Program reads in the contents of a maze from a text file named maze.txt • In a text file, a maze can be constructed using X characters for walls, space characters for paths, and a $ character for the goal • The first line of the text file is a number denoting how many rows are in the maze; the second line is a number denoting how many columns are in the maze; the subsequent lines contain the symbols that constitute the maze itself • The maze begins at position (1,1),	• Define a custom maze, saved in an external text file called maze.txt, where Line 1 is a number specifying how many rows are in the maze, Line 2 is a number specifying how many columns are in the maze, and all subsequent lines are rows built from the accepted characters of the maze. The defined maze must have a border on all sides, and the starting point must be location (1,1) • Respond to console questions by typing in the appropriate numbers shown on-screen	Introduce and familiarize user with program's operation. Ask user if they are ready to begin. While answer is not 0 or 1, throw error and ask again. If answer is 1, then exit program. Assign (1,1) as starting coordinates. Load maze from external text file into 2-dimensional vector. Traverse maze recursively via depth first search: • Mark current cell as visited • If current cell is goal, push coordinates to sequence of positions and return to caller that goal is found **(true)** • If **up** adjacent cell is both not a wall and not visited before, perform depth first search again with	Introduce and familiarize user with program's operation. Ask user if they are ready to begin. While answer is not 0 or 1, throw error and ask again. If answer is 1, then exit program. Assign (1,1) as starting coordinates. Load maze from external text file into 2-dimensional vector. Traverse maze recursively via depth first search: • Mark current cell as visited • Push coordinates to *explicit* stack instead of call stack • If current cell is goal, return to caller that goal is found **(true)** • If **up** adjacent cell is both not a wall and not visited before, perform depth

and the maze is assumed to have a border on all sides composed of X characters • Program outputs the sequence of ordered pairs in a path from the starting position to the goal or indicates if there is no solution • Program must be able to solve a diversity of mazes—any given maze providing it satisfies the maze definition criteria • Program must be robust in that it handles invalid user inputs		coordinates of that cell • If **right** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell • If **down** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell • If **left** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell • If goal has been found, push coordinates to sequence of positions and return to caller that goal is found **(true)** • Return to caller that there are no available directions left **(false)**	first search again with coordinates of that cell • If **right** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell • If **down** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell • If **left** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell • If goal has been found, return to caller that goal is found **(true)** • Pop current coordinates off explicit stack and return to caller that there are no available directions left

		If sequence of positions is empty, output that no path is available. Otherwise, iterate through sequence of positions from ending index to starting index, outputting the ordered pair at each index. End of program.	**(false)** If explicit stack is empty, output that no path is available. Otherwise, copy stack contents to an array and iterate through the array from ending index to starting index, outputting the ordered pair at each index. End of program.

Analysis (IPO Chart)

Input Data	Processing Steps	Output Data
• A 0 or 1 depending on whether the user is ready to begin • A custom maze, saved in an external text file called maze.txt, where Line 1 is a number specifying how many rows are in the maze, Line 2 is a number specifying how many columns are in the maze, and all subsequent lines are rows built from the accepted characters of the maze. The defined maze must have a border on all sides, and the starting point must be location (1,1)	Introduce and familiarize user with program's operation. Ask user if they are ready to begin. While answer is not 0 or 1, throw error and ask again. If answer is 1, then exit program. Assign (1,1) as starting coordinates. Load maze from external text file into 2-dimensional vector. Traverse maze recursively via depth first search: • Mark current cell as visited • If current cell is goal, push coordinates to sequence of positions and return to caller that goal is found **(true)** • If **up** adjacent cell is both not a wall and not visited	• A series of messages introducing the user to Patrician's Path and explaining how it works • A message asking the user if they are ready to begin, listing possible responses 0 for "Anything for my dear patrician!" and 1 for "Flee" • An error message for invalid inputs • A message listing the path as a sequence of ordered pairs, or a message indicating that the path is unavailable

	before, perform depth first search again with coordinates of that cell ● If **right** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell ● If **down** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell ● If **left** adjacent cell is both not a wall and not visited before and goal is not found, perform depth first search again with coordinates of that cell ● If goal has been found, push coordinates to sequence of positions and return to caller that goal is found **(true)** ● Return to caller that there are no available directions left **(false)** If sequence of positions is empty, output that no path is available. Otherwise, iterate through sequence of positions from ending index to starting index, outputting the ordered pair at each index. End of program.	

Design Using Pseudocode

main.cpp

```
Include Graph.h

Using standard namespace

Int main():
    Initialize constant string variable FILE_NAME to “maze.txt”

    Initialize constant integer variable START_X to 1
    Initialize constant integer variable START_Y to 1

    Instantiate Graph class as instance maze

    Call maze.loadGraph function, passing in argument FILE_NAME

    Initialize boolean variable pathExists to maze.depthFirstSearch(START_X, START_Y)

    Output “Path: ”
    Initialize vector of COORDs path to maze.getSolutionSet()

    If(path vector is empty):
        Output “Unavailable”
    Else:
        For(integer i starting at final index of path vector; i is greater than or equal to 0; i decreasing by 1 each time):
            Output (X coordinate at current index, Y coordinate at current index)
        Output end line

    Return 0
```

Graph.h

```
Ifndef GRAPH_H
Define GRAPH_H

Include iostream
Include fstream
Include vector
Include windows.h

Include Cell.h

Class Graph:
    Public:
        Graph constructor function()
        Virtual Graph destructor function()
        Void function loadGraph(constant string &FILE_NAME)
        Boolean function depthFirstSearch(integer row, integer column)
        Function getSolutionSet() which returns a vector of COORDs
    Private:
        Integer attribute height
        Integer attribute width
        Char attribute buffer
        Char attributes wall, treasure, path
        Attribute, called maze, that is a vector of vectors of Cell pointers
```

Attribute, called rowOfCells, that is a vector of Cell pointers
Attribute, called sequenceOfPositions, that is a vector of COORDs

Endif

Cell.h

```
Ifndef CELL_H
Define CELL_H

Class Cell:
    Public:
        Cell constructor function(char theSymbol)
        Virtual Cell destructor function()
        Char function getSymbol()
        Boolean function getVisitedStatus()
        Void function markVisited()
    Private:
        Char attribute symbol
        Boolean attribute visitedStatus

Endif
```

Graph.cpp

```
Include Graph.h

Graph::Graph():
    Set wall attribute to 'X'
    Set treasure attribute to '$'
    Set path attribute to ' '

Void Graph::loadGraph(constant string &FILE_NAME):
    Ifstream mazeFile(FILE_NAME)

    Pass contents of first line of mazeFile into height attribute
    Pass contents of second line of mazeFile into width attribute

    Do not skip white spaces

    For(integer row starting at 0; row is less than height; row increasing by 1 each time):
        For(integer column starting at 0; column is less than width; column increasing by 1 each time):
            Pass current character to buffer attribute

            If(buffer is a new line character):
                Pass next character to buffer
            rowOfCells.push_back(pointer to newly constructed Cell with buffer as its symbol)

        Push rowOfCells to maze 2D vector
        Clear rowOfCells

    Close mazeFile

Boolean Graph::depthFirstSearch(integer row, integer column):
    Declare boolean variable foundTreasure

    Call markVisited function on dereferenced pointer to current cell
```

```
        If(getSymbol() on dereferenced pointer to current cell equals treasure):
                Push current coordinates to sequenceOfPositions vector
                Return true

        If(getSymbol() on dereferenced pointer to above cell != wall AND !(getVisitedStatus() on dereferenced pointer to above cell)):
                Set foundTreasure variable to depthFirstSearch(row-1, column)

        If(!foundTreasure AND (getSymbol() on dereferenced pointer to right cell != wall AND !(getVisitedStatus() on dereferenced pointer to right cell))):
                Set foundTreasure variable to depthFirstSearch(row, column+1)

        If(!foundTreasure AND (getSymbol() on dereferenced pointer to below cell != wall AND !(getVisitedStatus() on dereferenced pointer to below cell))):
                Set foundTreasure variable to depthFirstSearch(row+1, column)

        If(!foundTreasure AND (getSymbol() on dereferenced pointer to left cell != wall AND !(getVisitedStatus() on dereferenced pointer to left cell))):
                Set foundTreasure variable to depthFirstSearch(row, column-1)

        If(foundTreasure):
                Push current coordinates to sequenceOfPositions vector
                Return true

        Return false

Graph::getSolutionSet() which returns a vector of COORDs:
        Return sequenceOfPositions

Graph::~Graph():
        // Destructor (empty body)
```

Cell.cpp

```
Include Cell.h

Cell::Cell(char theSymbol):
        Set symbol attribute to theSymbol
        Set visitedStatus attribute to false

Char Cell::getSymbol():
        Return symbol

Void Cell::markVisited():
        Set visitedStatus attribute to true

Boolean Cell::getVisitedStatus():
        Return visitedStatus

Cell::~Cell():
        // Destructor (empty body)
```

Testing/Verification

Test Case 1: Handling invalid inputs

This test case involved entering invalid inputs for each of the questions posed by the console. Whenever an invalid input was entered, the console promptly displayed the error message: “ERROR: Invalid entry. Please try again.” Several combinations of invalid input were tried, including different casings, letter-number sequences, space tabs, blank inputs, and more. No matter the nature of the invalid input, the program always managed to handle the error and only proceed once a valid input was typed in by the user.

Test Case 2: Background music

This test case involved running the program with computer speakers enabled. As expected, the predetermined program soundtrack began playing as soon as the console window opened.

Test Case 3: Exiting the program

The user should have the ability to exit the program both at the beginning and at the end. At the beginning, when faced with the question “Ready to begin?”, I entered “1” as my input; immediately, the program execution stopped, confirming the expected behaviour that an input of “1” should cease the program. At the end, exiting the program should simply be a matter of pressing any key. When this was done, the program appropriately ended, confirming that our program successfully allows the user to exit at both the beginning and end.

Test Case 4: Listening for key press

Occasionally, the program should wait for the user to press any key before proceeding to the next slide (or exiting if on the final slide). In slides where the user is prompted to press any key, I followed along and pressed a random key. Irrespective of what key was pressed, the program successfully listened for it and executed the promised behaviour—either moving on to the next slide or exiting the program.

Test Case 5: Settings

Throughout the program, the user is presented with many opportunities to customize their experience. These customization options are divided into Algorithmic Settings, Theme Settings, Hero Settings, and Treasure Settings. It is imperative that whatever options the user selects are reflected in their experience. To test that this is the case, I performed many trials of the program, using a different combination of settings each time, and looked to see if my chosen settings indeed customized the behaviour of the program. For example, did choosing a fast animation speed actually speed up the maze animation? Did the program correctly refer to the hero by the name I entered? What about the hero's role? Trial by trial, I exhausted all of the possible combinations of settings to confidently confirm that the settings work as advertised!

Test Case 6: Default Hero Name

In the case that the user leaves the input field blank for the hero name question, the program should default to a hero name of Moist Von Lipwig. To test this, I did not type anything for the hero name, simply pressing the Enter key. Later on, when the program needed to refer to the hero by name, it always used the name of Moist Von Lipwig. This confirms the program defaults to Moist Von Lipwig when the hero name input is left blank.

Test Case 7: Solving the Maze

There is nothing more important to be tested than ensuring that the DFS algorithm works and a valid path is reliably obtained. To test this, I ran the program against multiple different mazes, using the maze provided in the Problem Overview as a starting point and, subsequently, branching out to more complex, labyrinthine mazes. In cases where the maze was solvable, the program successfully outputted a valid path as both a sequence of positions and an animated trail of white breadcrumbs; in cases where the maze was unsolvable, the program accordingly indicated that the path was unavailable, and no breadcrumbs were shown. Additional maze metrics—namely, the length of path and cells visited—were also accurate, further proving the robustness and efficacy of the program's DFS algorithm.