Yulong Ran, Evan Ugarte
CS 146
11/14/2018

Project 2 Report

Table of Contents

Maze Class

The Maze class constructs a perfect maze by using the DFS algorithm. This class holds four instances variables: nodeGraph, nodeCount, size and myRandGen. NodeGraph is a two dimensional array of nodes and represents the Maze as a data structure, nodeCount is an integer that count the number of the Node in 2D array, size keep track of the size of the Maze, and myRandGen is a Random generator.

There are two constructors of the Maze class, where the first one takes a single parameter of the maze's size, and second takes a parameter of size and a seed to set myRandGen. Inside each constructor, we declare a new Random Generator a with a seed, either by a parameter or the default seed. We also construct a nodeGraph with the rows to be size+1 and column be size+1. We use a for loop to assign each element in the nodeGraph to a node that have the x and y coordinate as the index of the 2D array.

The Maze class contains a helper method called addEdge which takes two nodes' coordinates as parameters. This method has two local variables, key and key2 which represent the position of the edges. Then we check the x and y coordinates of two nodes, and if they are neighbors to each other, then we add both nodes to the HashMap of neighbors.

The Maze class also contains a helper method called getAllNeighbors which takes a node as parameter. The method constructs an empty arraylist that will contain all of the neighbors of the node. Then we check if the South, East, North and West neighbors exist according to their coordinates.

The Maze class contains two print methods: printMaze and outPutMazeToFile to visualize the Maze. The printMaze simply prints the Maze to the console and outPutMazeToFile will output a maze to a file.

generateMazeDFS

We use Depth First Search to generate Maze in this method. We first create an empty stack and initialize the current cell to be the first node in the nodeGraph and an integer that counts the number of visitedCells. Then we use a while loop to go through every node in the 2D array, inside the while loop, we first construct an arraylist that contains all the neighbors of a current cell, and then we construct another empty array list that should contain the neighbors that don't have edges. Then we have a for loop to go through every element inside the arraylist of neighbors, and if the node has all four walls, we add them to the arraylist of neighborsWithAllWall. After that, we check if the cell has Neighbors with all walls, if it does, we pick a random neighbor among the neighbors who have all four walls, and then we add an edge between the random chosen neighbor and current cell. Then we push the random neighbor to the stack, let current cell be the random neighbor and increment visited Cells. If the cell has no neighbors with all walls up, we pop another cell from the stack and make it as a current cell. The while loop terminates when all the cell have been visited once.

Node Class

The Node class holds instance variables such as x and y coordinates, a Hashmap of edges, a color stored as a string, a boolean variable that tells if the node is part of the path, an integer for the number of steps, and an integer that holds data. The hashmap follows a key value pair where a north has a key of 1, a south node of 2, an east node of 3 and a west node of 4. Every instance variable has a getter and setter method, were the parameter of the set method has the same type as the instance variable, for example a String or an int. The same can be said for the return value of the get methods, where the get methods take in no parameters, but return a value that is of the instance variable.

MazeSolverDFS

MazeSolverDFS is a class that has two instance variables, which is a two dimensional array of nodes called nodeGraph, and an integer that keeps track of the size of the maze. This integer holds the dimensions of the nodeGraph, to make sure we do not go out of bounds when traversing the nodeGraph array. The constructor for MazeSolverDFS takes in two parameters, which is a two dimensional array, and an integer for the array's dimensions. The array passed in has already been turned into a maze, meaning that a path exists from the starting top left node in the maze to the bottom right. The method getNodeAt returns a node based on the parameters it receives, which are x and y coordinates. The method checks if the coordinates sent are in bounds, and then returns a node at the passed coordinates, or null if they were out of bounds. The method getEdges returns all the edges that a node contains, based on the nodes HashMap of neighbors. We check each key in the HashMap, where north = 1, south = 2, east = 3 and west = 4. We add each value that isn't null to an ArrayList, and return the result. The method printSolutionToFile take parameter of a string filename. The method will output two maze into the file name as string filename: one with steps print in the middle of cell, another will print # as the solution path.

SolveMazeDFS

The method that finds the solution path of the maze makes use of a stack, to conduct a Depth First Search. We first add the top left node in the maze, which has an x and y coordinate of 1. From there, we set the node to being part of the solution path. We then enter a while loop that will terminate when either the stack is empty, or if we find the last node, which is located in the bottom right node. In the while loop, we pop a node off of the stack, and then we get all of the edges, which are stored in the ArrayList. We then enter a for loop, that checks each of the neighbors popped off, and sets the color to gray. If the color was initially white, we add the node to the stack, and set the node's steps to how many steps it took to get to the current node. We repeat this process until after we find the final node, and then we terminate the stack. Since all of the nodes in the stack are part of the solution path, we then pop off all of the nodes in the stack, and set their isPath boolean to true.

MazeSolverBFS

Maze Solver BFS is a class that solves the Maze by using Breadth First Search. This class has two instance variables: node Graph and size. node Graph is a two dimensional array of nodes, each element in the node Graph is a cell and whole array represents a prefect Maze as a data structure. Size is an integer that represents the dimension of the Maze. The constructor for MazeSolverBFS takes in two parameters: a two dimensional array, and an integer for the array's dimensions. MazeSolverBFS has two helper methods: getEdges and printSolutionToFile. The getEdges method takes a parameter of a node and find all the edges of this node based on the HashMap, the method returns an arraylist of edges in the order of South, East, North, West if they exist.

PrintSolutionToFile method output a solved Maze into a file. The output file will consist of two solved Mazes: a solved Maze with # , and a BFS Maze with steps. The steps will represents the how the Maze is solved by the BFS algorithm.

SolveMazeBFS

This method solves the Maze based on the Breadth First Search algorithm. The starting node will always be the node with the coordinate (1,1). The beginning node and exit node are initialized as a part of the solution path. The BFS make use of a Queue, where we first add the starting node to the queue, and then we enter the while loop that iterates through the two dimensional array. The while loop will terminate when the Queue is empty or when we find the exit node.

Inside the while loop, we will pop a node from the Queue and call it v. We then go through every neighbor of v that has edges, and connect v to it in a for loop. Essentially, we are finding all neighbors that have open walls. Inside the for loop, we will increment steps and assign to the steps to the node v, and we set the parent of node v to the current node. Then we add the node v to the Queue. At end of the for loop, if node v is the exit node, we will terminate the while loop.

The last step is to traverse back from the exit node. With the while loop, we will start from the exit node, find the parent of the exit node and increment this node as current. For each node we visit, we mark them as part of the solution path. Essentially, this while loop will find all the nodes that are part of the solution path. The while loop terminates when we reach our beginning node which is (1,1).