CSC111 Project Report: Generating and solving maze using the idea of tree and graph

Chenxu Wang, Runshi Yang, Yifei Sun, Haojun Qiu

Tuesday, March 16, 2021

Problem Description and Research Question

The question that we want to explore on in this project is **"how can we generate a maze and solve the maze using the algorithm of trees and graphs?"**. We chose this topic because after learning about the game tree concept in assignment 2, we are being inspired by game chess and interested in exploring some of the other kinds of games using the idea of trees and graphs that we've learned like maze. As we know, maze is a type of puzzle which the solver must find a path from the entrance to exit and it is a very essential part in most of the dungeon themed games. However, manually creating a maze or solving it can be quite hard since it requires the Creator to create a lots of 'dead ends' to confused the player and hide the true path in lots of possible paths. Since trees and graphs can represent almost all kinds of data so it can certainly be used to represent a maze as well! By generating a maze and solving it using them, this process might become a lot easier and useful for game designs. For example, by implementing a maze generator and a AI to solve the maze, it is very helpful to generate random levels in a dungeon themed game and provide solutions to that level. After doing some research, we find out that it is possible to generate mazes using a binary tree algorithm that start with a grid with multiple cells and for each different cell in the grid, it will chose to connect with either its north and west neighbour. This is similar to the idea of left and right attribute we've learned in a binary tree and the neighbour attribute in a vertex! This discovery makes us interested to explore on this topic and try generating a maze using similar ideas of trees and graphs. Furthermore, another reason that we choose this topic is that on assignment 2, we implemented a simple AI to play the mini chess by using game trees and make choices based on the player win rate. Therefore, we are curious to see how we can solve the maze we've generated also by using a simple AI and perhaps it can also adapt based on the path it takes over time.

Computational Overview

First of all, we will use graphs to represent the maze we generate and we built two data classes. The first one is `_Cell` which is the basic unit of the maze and it is going to represent one cell. It is like a single vertex in the graph and it will also has an attribute called neighbour, which stores the other four cells next to it. For the cell that locate on the edge of the maze it will only have 2 or 3 neighbors. On the other hand, We also created three different attributes to record the position of the cell in a maze. The first two attribute are `row_index` and `column_index`, which represent the row index and column index of cell in the a maze, respectively. The last attribute `coordinate` is a tuple that contains these two attributes, representing the coordinates of the cell in a maze, in the form of (row index, column index). Notice that for the row and column index, it actually represent the top left corner position of each cell if we multiply it by the width of each grid. The last attribute that we created for this class is walls which is a list of four numbers that can only take value 1 or 0. The value of the element is 1 if there is a wall in the specific direction and 0 if there is not a wall. The order of direction for this list is 'west', 'south', 'east' and 'north'. There is another attribute `state`, which is what we augmented this class for solving the maze later on, we will leave the detailed explanation when discussing maze solver. For the methods of this class, since cells are very similar to the vertexes in the graph, their methods are similar as well! For example, checking whether two cells are connected and get the neighbours of the cell. The different part is that when getting the neighbours of a cell, there is a direction input so that we can specify which neighbour we want to find. To visualize this class we need a method to draw the cell which is called `draw_cell`. In this method, an input number is required to represent the width of each cell. For the maze and cells, we are using pygame to visualize them and in this method, we are using the `pygame.draw.line` method to draw the top, right, bottom and left walls of the cell when the corresponding wall value is 1 instead of 0. To draw each cell in the correct corresponding position on the screen, it utilizes the

coordinates of the cell, as well as the input width. Notice that the row index mutiply with the width of the cell actually give us the y coordinate correspond to a pygame window since it tells us what row the cell is on. On the other hand, the column index with width can give us the x coordinate correspond to a pygame window. The second class we designed is `Maze`, which is the 'graph' itself. The `BinaryTreeMaze` class have two attributes - `overall_row` and `overall_column` - to keep track of the total numbers of rows and columns in a maze. Furthermore, the last and the most important attribute of the `BinaryTreeMaze` class is `cells` which is a list of `Cell` objects, representing all the cells inside the maze. We initialize `cells` by utilizing already-initialized overall numbers of rows and columns to create corresponding number of cells with their coordinates in the maze being set, and append them to `self.cells`, in order. Notice that at the beginning the cell is not connected with any neighbours and the whole maze is now a graph full of grids. For this project, since we are using a binary tree algorithm to generate the maze, we need a method to use the idea of this algorithm to generate the maze and make connections between cells and its attributes.

The idea of binary tree algorithm is to choose two different directions and let each cell to dig a path for either of these directions. In this project, we are choosing the 'north' and 'west' direction to dig path since it is the most common one. We are going to go through the maze from left to right and top to bottom. When we are getting to a cell in the maze, it can be connected to the cell on the left (west) or the one on the top (north), but it can not chose to connect to none of them or both of them. Because if it connects to nothing, then it will still be an isolated part in the maze and the path and the maze may not has a solution. On the other hand, if it connects to both of them, the width of the path might not be one unit. In this method, for each cell we chose left or top randomly by using the random module and push down one of the walls by modifying the walls attribute (changing the number 1 into 0) in the cell. In order to achieve this, we implemented two methods to search for the cell in north and west directions in the maze. However, in some corners of the maze there is no north and west cells so in these cases we can only chose the cell that is available or completely skip the cell when both directions are unavailable. After removing the wall, we make connections between these two cells by adding them to each other's neighbour attribute. As a result, this algorithm actually creates a graph similar to a binary tree! The root of the tree is the cell where we started which is the north west corner and for each cell in the graph, there is a connection toward the root and one or two connections from other directions since its south and east cells might chose to make connection with this cell. Furthermore, the left and right attribute of the binary tree is like the north and west directions we make connections with in the graph! We've implemented this algorithm as a method in the `BinaryTreeMaze` class called `binary_tree_algorithm`, also we defined a `run_example3` function in the `main.py` file for visualizing a maze generated by this algorithm. For `run_example3`, by inputting a specific size of the screen and width of the grid, the function will generate a new maze object, filled it with cells and using the binary tree algorithm to generate a maze and visualize it. However, this function does not return an actual maze. Therefore, inside the maze class, there is also an in class draw maze method to draw the maze based on the input screen size and it calls the cell.draw cell method for each cell. However, this method assumed that the maze is already being filled with cells and modified using binary tree algorithm so maze's binary tree algorithm should be done before visualizing the maze.

The second is to build a solver to the specific maze we generate. We have added an attribute `state` and method `update_cell_state` to the `_Cell` class to help implement algorithm and visualization for the maze solver. The general idea of our solver algorithm is to traverse all adjacent cell possible and then backtrack the path when it reaches the solution we desire. First, about initializing the attribute: every cell starts as 'naive' if there has been no operation or traverse on it. While the first cell, the start point, will be marked as 'start' independently. Second, about the update cell state, this is a function aimed at updating a cell's state based on the state and connection of the cell on the south and west of it. Intuitively, when both sides are cells that have been denied or cannot reach, we conclude this cell is not possible to exist in the solution path and we mark it as 'blocked'. And when it still has cells on sides(south and east) that are possible to be visited, we mark it as 'start', as after one trial has ended, this is where we may start. For the code, for the condition of 'blocked', we define it as there is either no cell on both sides, has a wall or has a cell that is not reachable and for the condition of 'start', we define it as has a reachable cell no wall between on either one of the sides. And then the main solver. We first create an exception class of `NoSolutionError`. Based on our general idea, we design it in a recursive way. We first append the cell we are working on to the path and then begin recursion.There are two circumstances we jump out of the recursion. Firstly, we we reach the end cell we desire and secondly, when the start cell is marked as 'blocked', which means there are no more valid trials and the maze will have no valid solution, we will raise `NoSolutionError`. Next we come to the recursive steps. When either our cell's neighbour on east or south side is reachable and has no wall between, we record the new cell as 'visited'(which means it is not 'naive' - reachable any longer) And whenever we reach a cell that has walls both on south and east, which means we cannot get to their south and east neighbours, we realize that this is a dead end and we mark it as 'block', what we want to do next is to go back to the cell in the path that still has possibilities to explore, which we have

prepared before, a cell of 'start'. In the code, we mark every cell in the path that is not a 'start' as 'blocked' and pop it from path. When we get to a 'start' cell, we recurse on it. Finally, we obtain a solution. And for the visualizer, what we do is simply painting the cell that is in the solution path. To do this, we add an optional parameter that enables us to draw a solution path when needed.

The library we used is pygame to generate a picture of the maze we built. And we will also let the maze solver to solve the maze after we run our function. Finally, we can change the color of the cells on the path the solver found.

Changes in our plan between proposal and final submission

For the computational path, we stilled used similar algorithms for generating and solving mazes from our proposal but we've added lots of new methods and decided to use pygame to visualize mazes. However, in our proposal our ta suggest that we use the pickle library to save and load mazes as our external library and it proven to be very useful. Since we have two separate filed for maze generation and solver. Moving a maze object that a person really likes is very hard but by using the pickle save and load function we can save the maze we generated in a separate files and load it in our solver file to solve user's favorite maze!

Instructions for running the program

Inside main.py, the first three run example function are being used to visualize a cell, a grid full of cells and a binary tree generated maze. For run example 1, the user need to input a cell object, the size of a screen as a tuple and the width of the cell. Then the runner will visualize a single cell with pygame.

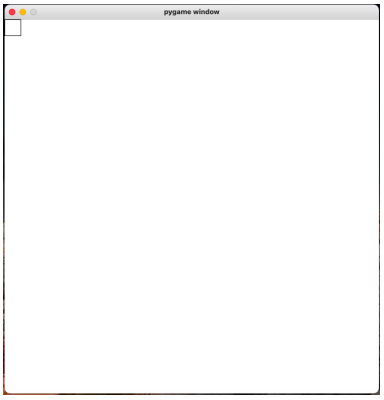

Figure 1: example1 - a single cell in the maze

On the other hand, for run example 2 only the size of a screen and the width of the cell is needed since we are creating a new maze object and visualize it without the binary tree algorithm. As a result, the pygame window would show a grid full of cells. After run this function with a size and width, the result will looks like this:

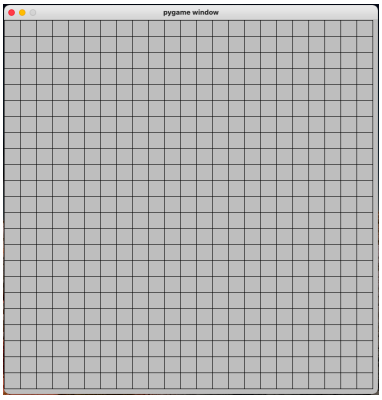

Figure 2: example2 - an unfinished maze

The most important visualize is run example 3 which require a input screen size. It will generate a maze using the binary tree algorithm which modifies the wall attribute in each cell and visualize them using the given size of a screen and the width of the cell. By creating a maze in example2, calling the binary tree algorithm and the draw maze method, we will finally get a maze like this:

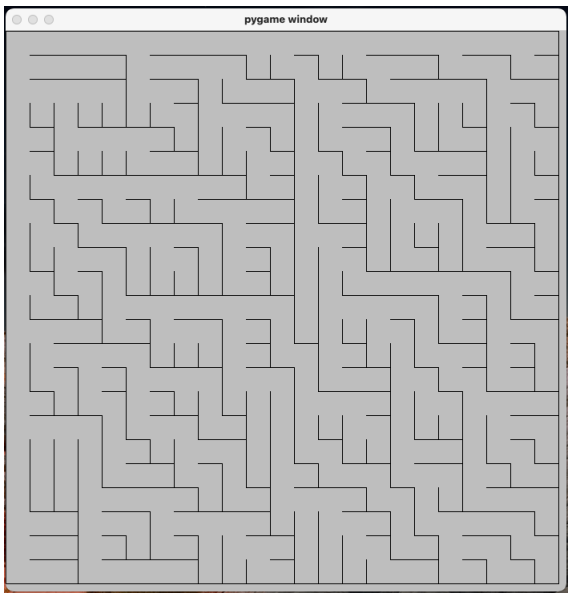

Figure 3: example3 - a maze (with paths)

Notice that resulting width of the grid by dividing screen size and column or row must be the same otherwise the grid will not be a square. We added this as a precondition of this method. At the end of maze generator.py there is a pickle maze function which uses the pickle library to save the maze you really like and you can use it to load it into our maze.solver.py using the load pickle function. For both of these functions, the only input required is the file name that you want to save the byte representation of the maze into. Then, you can use the same file name to load this maze using the load pickle function.

Finally, by running example 4 with an input screen size and cell width you will see a solution to the a maze we generated:

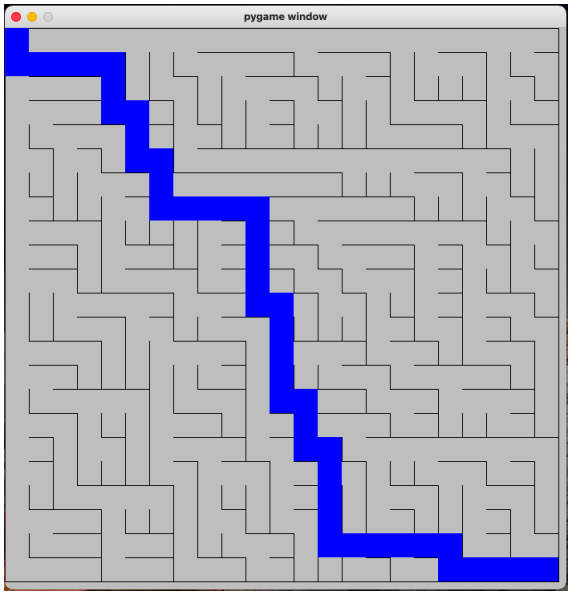

Figure 4: example4 - the solution to a maze

Discussion section

We have successfully generated mazes using tree and graph methods and we are able to find solutions for the generated mazes. So we think we've tentatively solved the problem of generating and solving mazes.
However, there are many limitations. In terms of generating the maze, we can only set the size of the maze, but can not change its shape. And because of the way we generated the maze, the first column and first row of the maze would never have any walls, so the starting point couldn't be manipulated. In terms of the maze solver, we think that the algorithm we used is relatively simple so it came with limitation. It just traverses most part of the maze, which is a depth first search algorithm so the running time of the maze-solving function is relatively long. So, if our users are looking for a maze solver that solves maze relatively quick (probably the one of breadth first search), this solver may not be the one they are looking for. In fact, we can relatively quickly used up the maximum recursion calls when we tried using the Solver after generating a relatively large maze (100×100), and run into a Recursion Error. Although we are aware that we can set a higher maximum recursion limit, this still shows the lack of efficiency our solver algorithm has - we are likely to traverse almost all cells before finding out the solution to the maze, especially when we have a "bad luck". We know there could be several different algorithms for solving the maze that could be much efficient than the one we implemented, but we choose to implement the one that is relatively straightforward, and uses the similar style of recursive search that we learned in this course. Furthermore, because the early-return of our implementation, we stop traversing when a possible solution is found, so we can only find one solution to the maze and it may not be the optimal (shortest) one. Especially when the maze is large, the solution it gives doesn't seem to be the decision of an intelligent person. Finally, a very interesting (or not) limitation we found, is that our maze are easy to solve. How come? Due to the property of binary maze generator algorithm that we use (removing the east or north wall for each cell), we can easily find a "pattern" for solution paths. That is, solution paths always require only the a right or down movement for each step from the starting point, and it never requires a up or left movement, like two examples - Figure 5 and Figure 6 - shown below and the one above - Figure 4. Therefore, it is easy to solve when people find this pattern after playing around with our mazes several times, and then they lose any interest with our mazes (we do not want this to happen though).

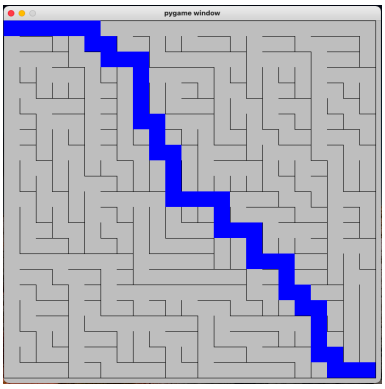

Figure 5

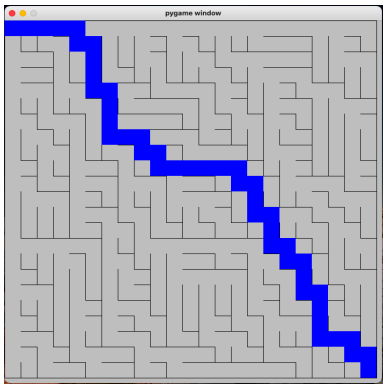

Figure 6

ALWAYS FROM TOP LEFT TO BOTTOM RIGHT

With so many limitations, we also have a lot of ideas for further exploration. The first and also the most urgent one is that we need to optimize the algorithm. One idea is to explore the grid close to the exit first, which will definitely save some time for us, and it is more in line with the decision making method of human when solving the maze problem. In order to do so we may have to add some functions to calculate the distance between the cells. And we're going to have to use a sorting algorithm to put the nearest cell to the end first. Actually, the heap data type satisfies our needs, but we have not learned it yet. Another idea is that we could update the Solver to "ExploringSolver" and add the maze_tree attribute to it, just like what we did in the second assignment, so that it can find the shortest path for us. Second, we want to make the visualization process more vivid. We may draw the maze as we go through the maze generation process. We also plan to animate Solver's position as it explores the maze, so that we can better understand the process. Finally, we can add some content of interaction, so that users can click on a position on the maze to view the solution route starting from that position. We handled this kind of mouse click event in assignment one, and it can't be too hard for us.

References

1. Jeulin-Lagarrigue, Michael. “Binary Tree Maze Generator - Algorithms: H.urna Academy.”H.urna, hurna.io/academy/algorithms/mazegenerator/binary.html.
2. Buck, J. (n.d.). Maze Generation: Binary Tree algorithm. Retrieved April 15, 2021, from http://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm
3. Larson, Dion. “MakeSchool-Tutorials/Trees-Mazes-Python.” GitHub, 7 Oct. 2015,github.com/MakeSchool-Tutorials/Trees-Mazes-Python/blob/master/P1-Solving-the-Maze/content.md