LABYRINTH SOLVER

OSLOMET

FINAL PROJECT

Problem Solving and Scripting (ACIT4420)

Candidate number: 237

Table of Contents

1. Introduction

As the final project of the course “Problem solving and scripting”, there were five project options with different subjects to implement. This project is on option number three “Labyrinth Solver”.

The purpose of this project is to implement a python code in order to generate and solve a 2-dimensional maze of a given size. The project should satisfy a couple of specifications:

- It is required to receive the size of the maze from the user.
- A 2D maze should be generated and displayed in a GUI.
- Users can select a starting point for the solution of the maze.
- The solved maze should be displayed on the GUI.
- The user should have the option to save and open a maze.

A detailed description of the maze will be given in the following sections of this report. In addition, an overview of different maze generation and solution algorithms will enable the reader to have a clear understanding of the various types of mazes that can be generated using each of these algorithms. Furthermore, it provides a good understanding of how different algorithms can be used to solve mazes and gives some insight into their pros and cons of them.

In the following sections, there will be a description of the selected approaches and an explanation of the implementation details. As part of the testing process, examples of how the code performs will be presented as well. Lastly, we will present an evaluation of the project as well as a conclusion to the work.

2. Problem statement

2.1. Maze

Generally speaking, a maze is a puzzle that consists of some complex and branching passages and walls that are interconnected in a complex way. In order to solve the maze, the solver must follow a path from the starting point to the endpoint which can be the exit point or a specific location inside the maze, and avoid dead ends along the way. Despite the fact that the words maze and labyrinth are used

interchangeably, they are not the same thing. When it comes to puzzles, a labyrinth is a sort of puzzle that has a single path with no branches, while a maze is a puzzle with multiple paths and dead ends which need to be solved.[1][2] In this project, the implementation of a maze has been considered.

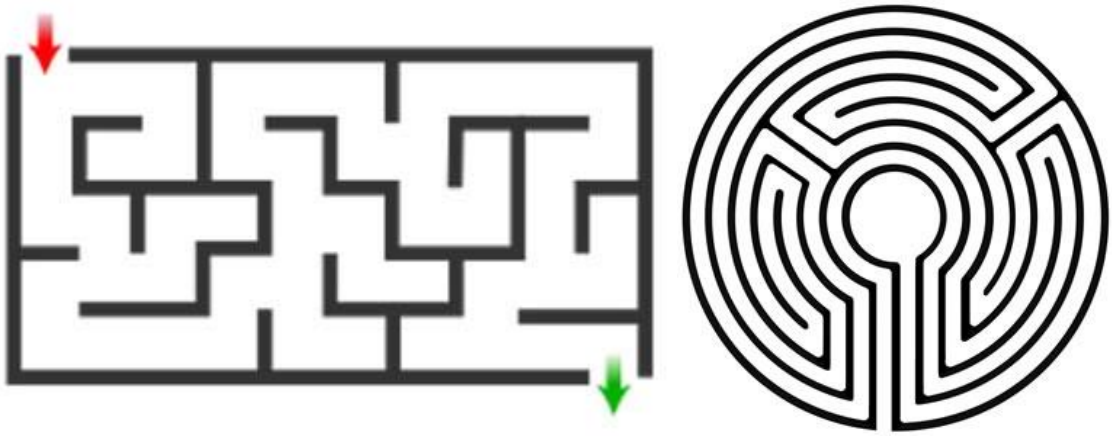

Figure 1 Schematic of maze and Labyrinth[2]

There are various types of mazes that can be created with different algorithms. The structure of mazes can be seen to be heavily dependent upon the algorithm that has been used to generate them. This can be seen in figure 2. The next section of this report will discuss some of these algorithms and their advantages and weaknesses.

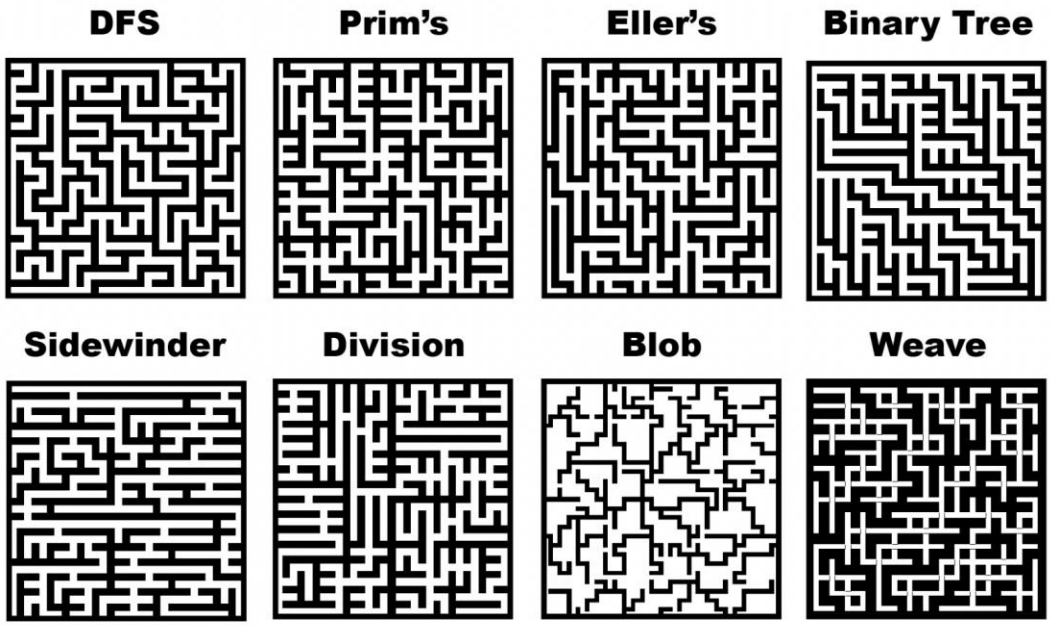

Figure 2 Schematic of maze structure generated with various algorithms[3]

2.2. Maze generation algorithms

There are several methods for generating mazes some of which are discussed in this section.

2.2.1 Depth-first search

A simple algorithm for generating mazes is the depth-first algorithm, also known as the "recursive backtracker" algorithm, which is based on the depth-first search principle. This algorithm produces mazes with long passages and few branches due to the fact that it explores the maze along each passage before it backtracks.

This search will start from a random cell and then explores all the cells in its neighborhood and once it has found a neighbor cell that is not visited, it will be added to a stack of cells. It also removes the walls between them, the newly visited neighbor, and the previous cell. The process will continue by moving to the next unvisited neighbor cell until all neighbors are visited that is the case with dead ends. It then uses the saved path in the stack and backtracks all the way back to find another unvisited neighbor. This process will continue until all of the cells have been visited.

There are two different approaches that can be taken to implement this algorithm, namely recursive and iterative implementations. It is preferable to implement the algorithm in an iterative manner since the recursive implementation might go beyond the maximum depth of the recursion stack while exploring a large number of areas at once.[1, 2, 4]

2.2.2 Randomized Kruskal's algorithm

This method is a kind of greedy algorithm that generates mazes based on the concept of a minimum-spanning tree. A minimum spanning tree is a set of edges and vertices in an undirected graph where edges connect the vertices with no forming cycles that have a minimum summation of the weights of the edges.

In simpler words, In this algorithm cells are in different sets initially. Then by selecting a wall randomly, it will check if both cells are not in one set, it will remove the wall and join the sets.

This process will continue until all cells belong to only one set. Unlike the depth first search, this algorithm generate mazes with short passages which are fully connected. Mazes generated with this approach is easier to solve relative to the other methods.[1, 4, 5]

2.2.3 Randomized Prim's algorithm

This algorithm is based on finding a minimum spanning tree similar to Kruskal's algorithm. It is important to note that the main difference between these two methods is the order in which the walls are added to the maze. In Kruskal's algorithm, the walls of the maze are added to the maze in random order, but in Prim's algorithm, the walls of the maze are added to the maze in order of their length. As a result, Prim's mazes will have a few branches and long paths, similar to the depth-first algorithm that Prim uses to create mazes.

In order to implement this algorithm, it is necessary to create a grid of cells and walls. Next, a starting cell will be randomly selected, and walls will be added to the queue in order of their length, then the walls will be popped off the queue, and a new set of cells and walls will be created to include the new cells. As a result, all of the cells inside the maze will continue to be covered by this method.[1, 4]

2.2.4 Recursive division method

As one can find based on its name, the recursive division algorithm divides the grid of cells vertically and horizontally into two regions, following its purpose. It then selects a randomly selected region and divides it into two smaller sub-regions by creating a wall between them. There is a recursive process involved in these divisions, where all regions are divided up until a set of criteria such as a specific size of the cells is met.

It is relatively simple to implement this method, and it produces mazes that have a regular structure, with straight walls, and a high density of walls.[1, 4]

2.2.5 Wilson's algorithm

By using loop-erased random walks, Wilson's algorithm generates an unbiased sample from a uniform distribution over all mazes.
Essentially, Wilson's Algorithm involves walking randomly around a starting cell,

erasing loops that it makes until it reaches a part of the maze that it has already constructed. Next, it chooses the first cell that hasn't yet been included in the maze as the new starting point. [1, 2, 4, 6]

2.2.6 Aldous-Broder algorithm

This is an easy algorithm that generates a random maze from a random starting point and searches all of its neighbors to find its solution. A cell that has not been visited by a neighbor will be the next cell in the algorithm and is marked as visited if it is not visited by one neighbor. The process will continue to move from random unvisited neighbors to random unvisited neighbors in a repetitive manner. As soon as there are no more unvisited neighbors left, the algorithm will stop and a maze will be generated.

This algorithm has several advantages, including its simplicity as well as the fact that the generation process is very intuitive, which makes it an excellent starting point for beginners as it is easy to understand. In order to create mazes, this algorithm has the main disadvantage of being very inefficient in terms of its efficiency.[1, 4, 6]

2.2.7 Cellular automaton algorithm

Cellular automata are discrete models that consist of a grid of cells, each with a discrete number of states and a set of rules that govern when cells change states. Based on the neighboring cells or neighborhood of a given cell, the rules determine when a given cell will change state.
In each time step, the rules are applied to each cell of the grid once before any changes to the state of the cell occur.
Maze and Mazectric are two well-known examples of cellular automata. In the first case, cells survive if they have at least one and no more than five neighbors. In the second case, one to four neighbors are necessary for cells to survive. Three neighbors are required for a cell to be born.[1, 7]

2.3.Maze solution algorithms

In this section, an overview of some of the important maze solution algorithms is presented.

2.3.1 Random mouse algorithm

This algorithm is a very basic and trivial method to solve the maze. The solution is like considering a mouse searching in the grids of the maze from a given starting point. When the mouse reaches a joint cell, it will randomly select one passage and move in that direction. It continues the search until it finds the exit point.

The disadvantage of this model is that, although it finally finds the solution, it might take a long time to reach the endpoint. Therefore, it is not suitable for mazes with a lot of dead ends or loops. There is always the possibility of improving the performance of this method by incorporating some rules.[8]

2.3.2 Wall follower

According to this algorithm, the maze is solved by following the walls of the maze in a certain direction. The visited cells of the maze will be recorded as it moves through the passages within the maze. This method guarantees finding the solution without getting lost. In the following image, the wall following the solution of the maze is displayed.[8]

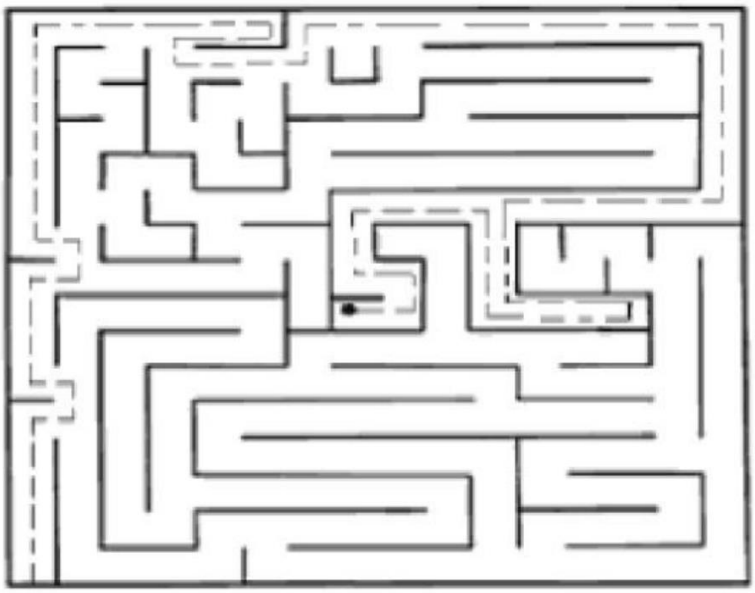

Figure 3 Wall follower maze solution[[9]]

2.3.3 Pledge algorithm

If the entrances and exits to the maze are on the outer walls, the wall follower method can still be used to solve disjoint mazes. It is possible, however, that the solver may start inside the maze in a section separated from the exit. As a result, wall followers will keep moving around their ring. This problem can be solved using the Pledge algorithm.

To circumvent obstacles, the Pledge algorithm requires an arbitrarily chosen direction. A hand is walked along an obstacle (say the right hand) while the angles turned are counted. The solver leaves the obstacle when the angular sum of the turns made is zero, and the solver continues in the original direction.

There is no way this algorithm can be applied to finding the way from an entrance on the outside of the maze to an endpoint inside the maze.[8]

2.3.4 A* search

In this method, a path with the least cost will be found from a random starting point. This algorithm follows the path of the lowest expected total distance and saves the path segments that are sorted based on the priority in a queue. The order of visiting the cells in the maze is calculated by a heuristic function of cost. The first cell which has the lowest sum of the cost for its path and the remaining cost will then be selected. [8]

2.3.5 Dead-end filling

This algorithm will examine the entire maze at once, and the first step is to identify the dead ends. The algorithm traverses the maze like a matrix and marks all the cells with three walls, for example, by stacking them. In the second step, dead-ends will be filled up until a junction is reached; only the correct passages will remain unfilled. In order to accomplish this, a dead-end is retrieved from the stack and the corridor is followed. The last step, from entry to exit point, is already the solution.[8, 10]

2.3.6 Maze-routing algorithm

An algorithm for finding the path between two locations in a maze in a low-overhead manner is known as the maze-routing algorithm. Initial implementations of the algorithm were designed for chip multiprocessors (CMPs). However, it has been proven to be applicable to any grid-based maze. According to this scheme, a

source cell sends a message to each of its four neighbors. Messages propagate as waves between nodes. The connecting path is determined by the first wave front that reaches the destination. Furthermore, the algorithm is capable of detecting when there is no path between two locations in the grid (maze). Additionally, the algorithm is intended to be used by a person without any prior knowledge of the maze and has a fixed memory complexity regardless of the maze's size. This requires the use of four variables to discover the path and detect locations that are unreachable. [8, 11]

2.3.7 Breadth-first algorithm

The breadth-first search algorithm is one of the algorithms that find the shortest path in the maze. It starts at a random cell in the grid and explores all the neighboring cells and uses a queue to save the visited cells. Then it explores the cells that are in the next level by increasing the distance from the starting cell. When the final point is reached it will use the path all the way back to the starting point as the shortest path through the maze.[8, 12]

3. Methodology

3.1 Maze generation

3.1.1 Recursive division

The recursive division method has been selected for random maze generation. This is a very simple technique that creates a perfect maze with no dead cells and guarantees that there is always a solution available. It works as follows:

1) Start with a plain grid of a given size with no walls.

2) Set boundary walls on all outer sides.

3) Start with the whole grid and recursively divide it into two rectangular sub-areas, either horizontally or vertically, leaving only one opening in the created walls.

4) Repeat step 3 on the resulting smaller rectangles until they can’t be divided any further.

3.2. Solution algorithms

In this study two different solution methods have been implemented; Breadth-first search and Depth-first search (DFS).

3.2.1 Breadth-first search

Breadth-first search guarantees that the solution will be the shortest possible (although for perfect mazes with only one solution, the guarantee doesn't matter). It starts at any random starting point in the grid and explores all the neighboring cells and uses a queue to save the visited cells. Then it explores the cells that are in the next level with the shortest distance to the starting cell. When the final point is reached it will use the path all the way back to the starting point as the shortest path through the maze.[8, 12]

3.2.2. Depth-first search

Depth-first search (DFS) is an algorithm similar to BFS. It starts at some arbitrary node of the graph like BFS but explores as far as possible along each branch before backtracking. For a DFS non-recursive implementation, we are using a stack instead of a queue to store nodes which will be explored. This way we check the nodes first which were last added to the stack.

DFS is not guaranteed to find the best/shortest solution. This means DFS might not be the optimal choice to find a path in a maze, but it has other applications in finding connected components or maze generation. BFS is generally more suitable for finding the shortest path, while DFS is better at exploring all possible paths.[8, 12]

4. Implementation

The overall design of the code uses object-oriented principles to modularize the maze generation, bookkeeping and solution logics, and graphical user interface and visualization into separate classes, allowing them to be easily used and reused in different contexts.

The above code defines three classes: `Maze`, `MazeGenerator`, and `MazeGUI`. The Maze class which is the main type representing a maze provides methods for solving the maze using different algorithms, as well as tools for saving and loading the maze to/from a file.

The `MazeGenerator` class uses the "recursive division" algorithm to generate mazes. It accepts a size and returns a randomly generated maze with a random exit point.

The `MazeGUI` class is responsible for taking input parameters from the user (namely size, starting point, method of solving the maze, etc), and then visualizing the generated maze and drawing the final solution. It also provides an interface to load and save mazes from/to files with UI dialogs.

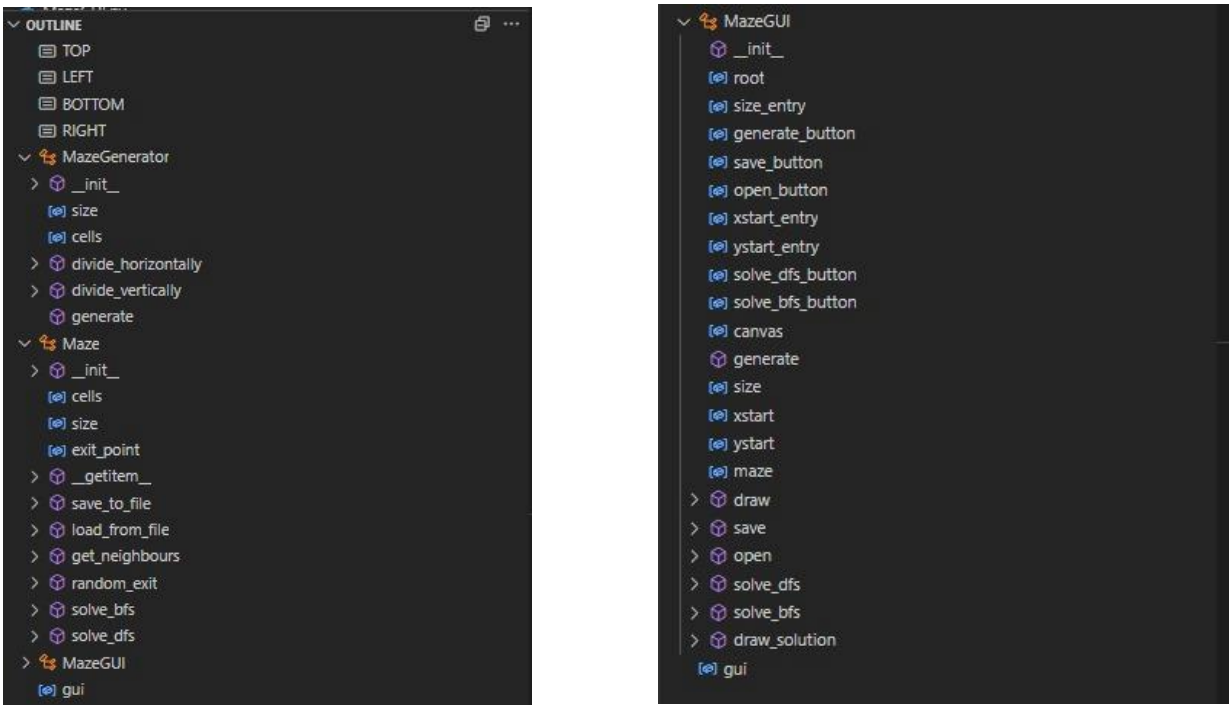

Figure 4 Diagram of the classes in the project

The following sections will provide the implementation details for each of these classes with figures of the code snippets to clarify the explanations.

4.1 Maze generator class

The class `MazeGenerator` is implemented to generate mazes based on the recursive division algorithm.

The grid of the maze is represented as a three-dimensional NumPy array. The first two dimensions correspond to the location of the cell and the last dimension

represents 4 walls for each cell. It means that if a cell has a wall, the last dimension of the array for that cell will be equal to one in a certain index.

The walls are numbered based on their location relative to the cell. The top wall is assigned a value of 1, the left wall is assigned a value of 1, and similarly, the Bottom and Right walls correspond to 2 and 3.

The `MazeGenerator` has a constructor `__init__()` which has an argument for the size of the grid for the maze. It initializes the maze grid as a NumPy array of zero values with the size of (size*size*4).

In the next step, the constructor sets the boundary of the maze by setting the walls on the top, bottom, left and right sides of the corresponding cells in the boundary. Therefore, the edges of the array will be equal to 1 which ensures that the maze has a complete outer wall.

```
def __init__(self, size: int):
    self.size = size
    self.cells = np.zeros([size, size, 4], dtype=np.uint8)

    # initializing outer walls
    self.cells[0, :, TOP] = 1
    self.cells[size - 1, :, BOTTOM] = 1
    self.cells[:, 0, LEFT] = 1
    self.cells[:, size - 1, RIGHT] = 1
```

Figure 5 code of the __Init__ method

There are two methods in the class for dividing the maze into two sections repeatedly by using horizontal and vertical lines.

The `divide_horizontally` method as its name is self-explanatory divides the maze horizontally and leaves a gap open to let two sections be connected.

In the first part, the method checks the size of the region before division, and if the difference between the `y_start` and `y_end` parameters is smaller than one, the method will return with no division.

In case the size of the region is big enough to be divided, the method generates two random numbers for determining the position of the divider and the opening within the divider.

The method then sets the bottom walls of all the cells of the divider by assigning the value of one to these cells. It will skip the gas position and let an opening between two sections.

As the region is divided into two sections, each section should be divided again. Therefore, by calling the method `divide_vertically`, for each section, the maze will recursively be divided into smaller regions until all passages in the maze have a width of one.

```python
def divide_horizontally(self, x_start: int, x_end: int, y_start: int, y_end: int):
    """Divide the maze horizontally at a random point, leaving one gap open"""

    # Stop dividing if the region is too small
    if y_end - y_start < 1:
        return

    divide_at = random.randint(x_start, x_end - 1)
    gap_pos = random.randrange(y_start, y_end)

    for i in range(y_start, y_end + 1):
        if i == gap_pos:
            continue

        self.cells[divide_at, i, BOTTOM] = 1


    self.divide_vertically(x_start, divide_at, y_start, y_end)
    self.divide_vertically(divide_at + 1, x_end, y_start, y_end)
```

Figure 6 Code of the divide_horizontally method

The `divide_vertically` method is similar to the `divide_horizontaly` method with the difference in dividing into two regions vertically at a random point.

The generate method triggers the division by calling the `divide_horizontally` method on the whole grid. the resulting maze which is an array of cells will be returned by the method.

4.2 Maze class

The Maze class represents a maze and provides methods to solve it using either breadth-first search (BFS) or depth-first search (DFS). It also provides methods to save and load the maze to/from a file.

The `__init__` method is the constructor of the class, takes a NumPy array of cells as an argument, and assigns it to the cells attribute of the object. It also calculates the size of the maze as the first dimension of the cells array and generates a random exit point for the maze by calling the `random_exit()` method.

The `__getitem__` method is a "dunder" (double underscore) method that allows the cells array to be accessed directly from a maze object as if it were an indexable container, such as a list. For example, maze[0,0] would return the first cell of the maze object.

The `save_to_file` method saves the Maze object to a file using the pickle module, which serializes the object and stores it in a binary format. The filename argument specifies the name of the file to save the object to. In the `MazeGUI` class, this method gets called after the user chooses a filename in the save dialog.

```python
class Maze:
    """
    Maze class represents a maze and provides methods to solve it.
    It also provides methods to save and load the maze to/from a file.
    """

    def __init__(self, cells: np.ndarray):
        self.cells = cells
        self.size = cells.shape[0]
        self.exit_point = self.random_exit()
        print("Exit point:", self.exit_point)

    def __getitem__(self, index):
        """dunder method to directly access cells by index"""
        return self.cells[index]

    def save_to_file(self, filename: str):
        """Save the maze to a file using pickle"""
        pickle.dump(self, open(filename, "wb"))

    @classmethod
    def load_from_file(cls, filename: str) -> "Maze":
        """Load the maze from a file"""
        return pickle.load(open(filename, "rb"))
```

Figure 7 Code of the Maze class

The `load_from_file` method is a class method, as indicated by the `@classmethod decorator`. Class methods are methods that are bound to the class and not the instance of the class. This means that they can be called on the class itself, as well as on any instance of the class. In this case, the method uses the pickle module to load a Maze object from a file and returns it. This is handy because it acts similarly to a factory method to build a maze object by reading the contents of a file.

The `get_neighbours` method takes the coordinates x and y of a cell in the maze and returns a set of tuples containing the coordinates of the neighbors of that cell that are not blocked by walls.

The `random_exit` method generates a random exit point for the maze. It first chooses a random side of the maze (top, bottom, left, or right) and then selects a random point along that side to be the exit point. It then leaves a gap in the wall at the exit point and saves that in the cells array.

```
def get_neighbours(self, x, y) -> set[tuple[int, int]]:
    """Return a set of neighbours for a given cell not blocked by walls"""
    neighbours = set()
    if self.cells[x, y, BOTTOM] == 0 and x < self.size - 1:
        neighbours.add((x + 1, y))
    if self.cells[x, y, RIGHT] == 0 and y < self.size - 1:
        neighbours.add((x, y + 1))
    if self.cells[x - 1, y, BOTTOM] == 0 and x > 0:
        neighbours.add((x - 1, y))
    if self.cells[x, y - 1, RIGHT] == 0 and y > 0:
        neighbours.add((x, y - 1))

    return neighbours
```

Figure 8 Code of the get_neighbors method

The `solve_bfs` method uses a breadth-first search to solve the maze. Breadth-first search is an algorithm for traversing or searching a graph, which consists of starting at a root node and exploring all of the neighbor nodes first, before moving on to the next-level neighbors.

The `solve_bfs` method takes two optional arguments: `x_start` and `y_start`, which represent the starting coordinates of the search. These default to 0, 0 if not specified (in `MazeGUI` users will specify them).

The method begins by initializing a queue to store the path and a set to store the visited cells. It also initializes an empty list to store the solution path. It then adds the starting cell to the queue and the visited set.

Next, the method enters a loop that continues as long as the queue is not empty. Inside the loop, it takes the first element from the queue and marks it as visited. It also adds this cell to the solution path. If the current cell is the exit cell, the method returns the solution path.

If the current cell is not the exit cell, the method checks if it has any unvisited neighbors. It does this by taking the difference between the set of neighbors and the set of visited cells. If there are any unvisited neighbors, the method adds them to the queue and marks them as visited. In the next iteration, next-level neighbors will be popped from the stack and this will continue until we reach the exit point.

```python
def solve_bfs(self, x_start: int = 0, y_start: int = 0) -> list[tuple[int, int]]:
    """Solve the maze using breadth first search"""

    x_exit, y_exit = self.exit_point

    # queue to store the path
    queue = deque([(x_start, y_start)])
    # visited cells
    visited = set()
    # solution paths
    solution = []

    while queue:
        x, y = queue.popleft()
        solution.append((x, y))

        # if the current cell is the exit cell, then we are done!
        if x == x_exit and y == y_exit:
            print("Reached to the exit!")
            return solution

        # check if the current cell has any unvisited neighbours
        # since both neighbors and visited are sets, we can use subtraction to find the difference
        unvisited_neighbours = list(self.get_neighbours(x, y) - visited)

        # if it has, then add all of them to the queue
        if unvisited_neighbours:
            queue.extend(unvisited_neighbours)
            visited.update(unvisited_neighbours)

    return solution
```

Figure 9 Code of the Breadth First method

The `solve_dfs` method uses a similar approach to solve the maze using a depth-first search. Depth-first search is an algorithm for traversing or searching a graph, which consists of starting at a root node and exploring as far as possible along each branch before backtracking.

The `solve_dfs` method takes two optional arguments: `x_start` and `y_start`, which represent the starting coordinates of the search. These default to 0, 0 if not specified (in `MazeGUI` users will specify them).

The method begins by initializing a stack to store the path and a set to store the visited cells. It also initializes an empty list to store the solution path. It then adds the starting cell to the stack and the visited set.

Next, the method enters a loop that continues as long as the stack is not empty. Inside the loop, it takes the top element from the stack and marks it as visited. It

also adds this cell to the solution path. If the current cell is the exit cell, the method returns the solution path.

If the current cell is not the exit cell, the method checks if it has any unvisited neighbors. It does this by taking the difference between the set of neighbors and the set of visited cells. If there are any unvisited neighbors, the method pushes them to the stack and marks them as visited. It continues doing that until a branch reaches its end.

```python
# write function to solve the maze using depth first search
def solve_dfs(self, x_start: int = 1, y_start: int = 1) -> list[tuple[int, int]]:
    """Solve the maze using depth first search"""

    x_exit, y_exit = self.exit_point

    # stack to store the path
    stack = [(x_start, y_start)]
    # visited cells
    visited = set()
    # solution paths
    solution = []

    while stack:

        x, y = stack.pop()
        solution.append((x, y))

        # if the current cell is the exit cell, then we are done!
        if x == x_exit and y == y_exit:
            print("Reached to the exit!")
            return solution

        # check if the current cell has any unvisited neighbours
        # since both neighbors and visited are sets, we can use subtraction to find the difference
        unvisited_neighbours = list(self.get_neighbours(x, y) - visited)

        # if it has, then add all of them to the queue
        if unvisited_neighbours:
            stack.extend(unvisited_neighbours)
            visited.update(unvisited_neighbours)

    return solution
```

Figure 10 Code of the Depth First method

One key difference between the `solve_bfs` and `solve_dfs` methods is the data structure used to store the path. `solve_bfs` uses a queue, which follows a first-in, first-out (FIFO) approach, while `solve_dfs` uses a stack, which follows a last-in, first-out (LIFO) approach. This means that the order in which cells are visited and added to the solution path is different for the two methods. BFS is generally more suitable for finding the shortest path, while DFS is better at exploring all possible paths.

4.3 MazeGUI class

`MazeGUI` class is providing the graphical user interface of the maze using the `Tkinter` package. `Tkinter` is the built-in module in Python that provides a simple interface for creating (GUI) applications.

The `__init__` method creates a `Tkinter` form and adds some buttons and a canvas of size 500 *500 to it. The method adds an entry on the form which allows the user to provide the size of the maze. Then by clicking the “Generate Maze” button it will generate the maze. Other buttons are save maze, open maze, and solve maze which is added to the form.

The generate method of the `MazeGUI` class stores the size value that the user is provided in the entry. Then, It calls the generate method of the `MazeGenerator` to create a maze and forward the maze object to the draw method to display the maze on the canvas.

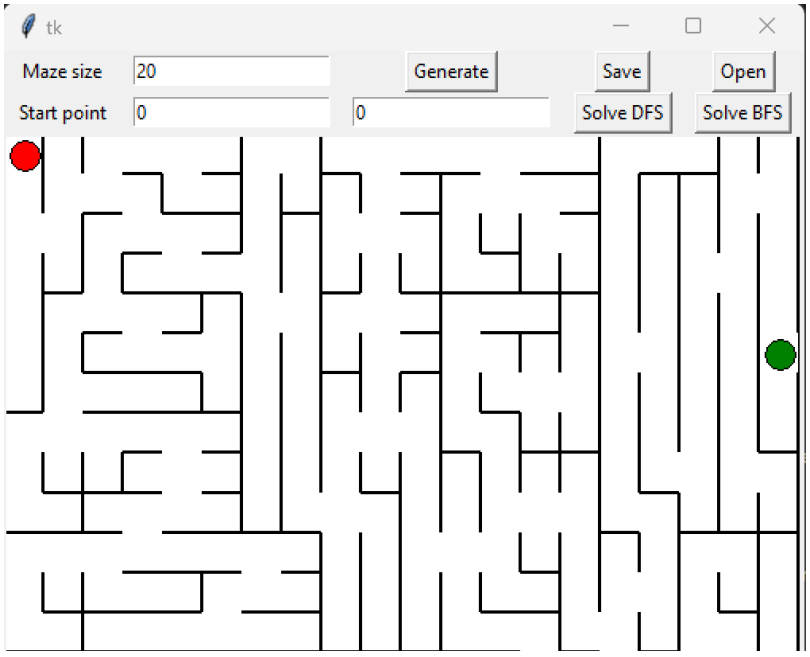

Figure 11 Schematic of the GUI

The `solve_dfs` and `solve_bfs` methods are called when the "Solve DFS" and "Solve BFS" buttons are clicked, respectively. Both methods get the starting coordinates from the text entry fields and call the `solve_dfs` and `solve_bfs` methods of the Maze class, respectively, to solve the maze. They then draw the solution path on the canvas using the `draw_solution` method.

The draw method takes a canvas object and a maze as input and draws the maze on the canvas by the `create_line` method of the canvas object.

First, the cell-size constant is used to determine the size of the cells in the maze given its size attribute. Then, the `create_rectangle` method of the canvas object is used to draw a white background on it.

```
# Draw the maze on a Tkinter canvas
def draw(self, canvas, maze, linewidth=2):
    # Constants for the size of the cells in the maze
    CELL_SIZE = 500 // self.size

    # Draw the white background
    canvas.create_rectangle(0, 0, 500, 500, fill="white")

    # Draw the walls in the maze
    for y in range(0, self.size):
        for x in range(0, self.size):
            # Draw the top wall
            if maze[y][x][TOP]:
                canvas.create_line(
                    x * CELL_SIZE,
                    y * CELL_SIZE,
                    (x + 1) * CELL_SIZE,
                    y * CELL_SIZE,
                    fill="black",
                    width=linewidth,
                )

            # Draw the bottom wall
            if maze[y][x][BOTTOM]:
                canvas.create_line(
                    x * CELL_SIZE,
                    (y + 1) * CELL_SIZE,
                    (x + 1) * CELL_SIZE,
                    (y + 1) * CELL_SIZE,
                    fill="black",
                    width=linewidth,
                )
```

Figure 12 Code of the Draw method

The code then iterates over the cells in the maze to draw the walls of the cells. Using two for loops each cell in the maze is traversed and walls of the cells in the directions that have a value of one will be drawn. The indexes of the cell and the cell size are fed to the `create_line` method for drawing the walls on canvas.

In addition, two red and green circles are displayed using the `create_oval` method of the canvas to show the start and end points of the maze.

The save function of the `MazeGUI` class asks the user for the file name that will store the maze. By the `asksaveasfilename` method of file dialog, it receives the name with an extension of "`.maze`". Then, the method `save_to_file` is called with the file name to store the maze using pickle.

The open function in the maze similarly asks the user for the file name to open the maze. This will be done with the `askopenfilename` method of the `filedialog`.

Given the file name, the `load_from_file` method of the maze class will be called which returns the maze object. As the maze also has an attribute size, it is required to save the maze size in this attribute as well.

```
# Open a maze from a file
def open(self):
    # Ask the user for the file to open
    file_name = filedialog.askopenfilename()
    # Open the maze from the file
    self.maze = Maze.load_from_file(file_name)
    self.size = self.maze.size
    # Draw the maze on the canvas
    self.draw(self.canvas, self.maze)

# Solve the maze using depth-first search
def solve_dfs(self):

    # Get the starting point from the entry fields
    xstart = int(self.xstart_entry.get())
    ystart = int(self.ystart_entry.get())

    solution = self.maze.solve_dfs(x_start=xstart, y_start=ystart)
    self.draw_solution(self.canvas, solution)
```

Figure 13 Code of the open and save methods

The final step in this function is to display the opened maze. It will be done with the draw function which receives the canvas and maze objects as its input parameters.

The solve method calls the `solve_dfs` method of the maze and receives the solution which is then fed to the `draw_solution` method to be drawn on the canvas.

The `draw_solution` method receives the canvas object and the path as parameters and draws the solution on the canvas in the `Tkinter` form.

It first assigns the value of `cell_size` and then calls the method draw to display the initial state of the maze on canvas.

The `next_step` function is defined and called inside the `draw_solution` method. Once it is called, the next step of the solution will be displayed on the canvas.

```
# Draw the solution on the canvas
def draw_solution(self, canvas, path):
    # Constants for the size of the cells in the maze
    CELL_SIZE = 500 // self.size

    # Draw the initial state of the maze
    self.draw(canvas, self.maze)

    # Function to draw the next step of the solution
    def next_step():
        # If there are more steps in the solution
        if path:
            # Get the next step
            x, y = path.pop(0)
            # Draw the step on the canvas
            canvas.create_rectangle(
                (y + 0.30) * CELL_SIZE + 1,
                (x + 0.30) * CELL_SIZE + 1,
                (y + 0.70) * CELL_SIZE,
                (x + 0.70) * CELL_SIZE,
                fill="yellow",
            )
            # Call this function again after a short delay
            self.root.after(20, next_step)

    # Start animating the solution
    next_step()
```

Figure 14 Code of the draw solution method

First, it checks the path to see if there is more step in the solution and gets the first step as x and y values by method pop. Each step is shown on the maze by drawing a yellow rectangle utilizing the `create_rectangle` method of the canvas. It recursively calls itself after a short delay to display the next step.

5. Testing and examples

Below there are 3 examples of randomly generated mazes with different sizes and different exit points, all generated using the recursive division method. The red dot shows the start points while the Green dot shows the final exit point.

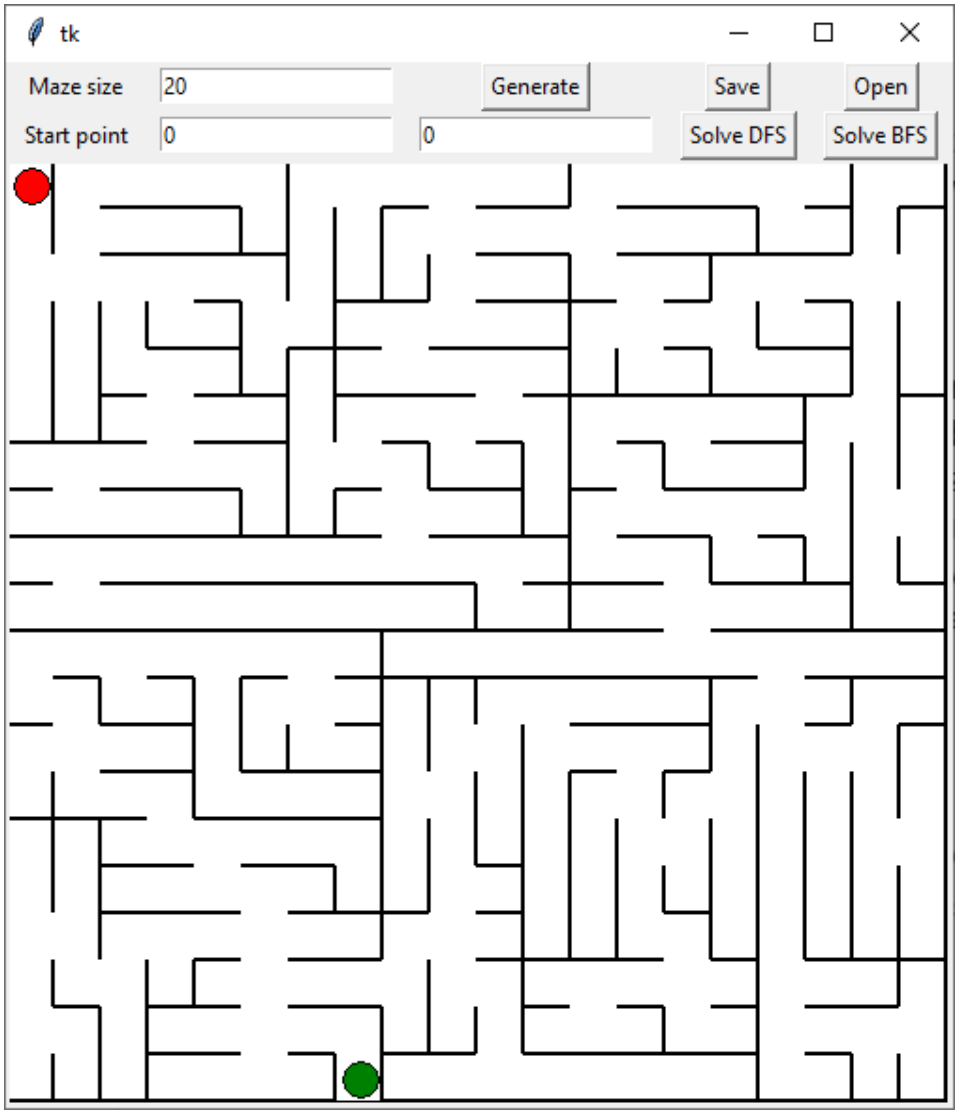

Figure 15 Maze-1: Size=20, start=(0,0), exit=(19,7)

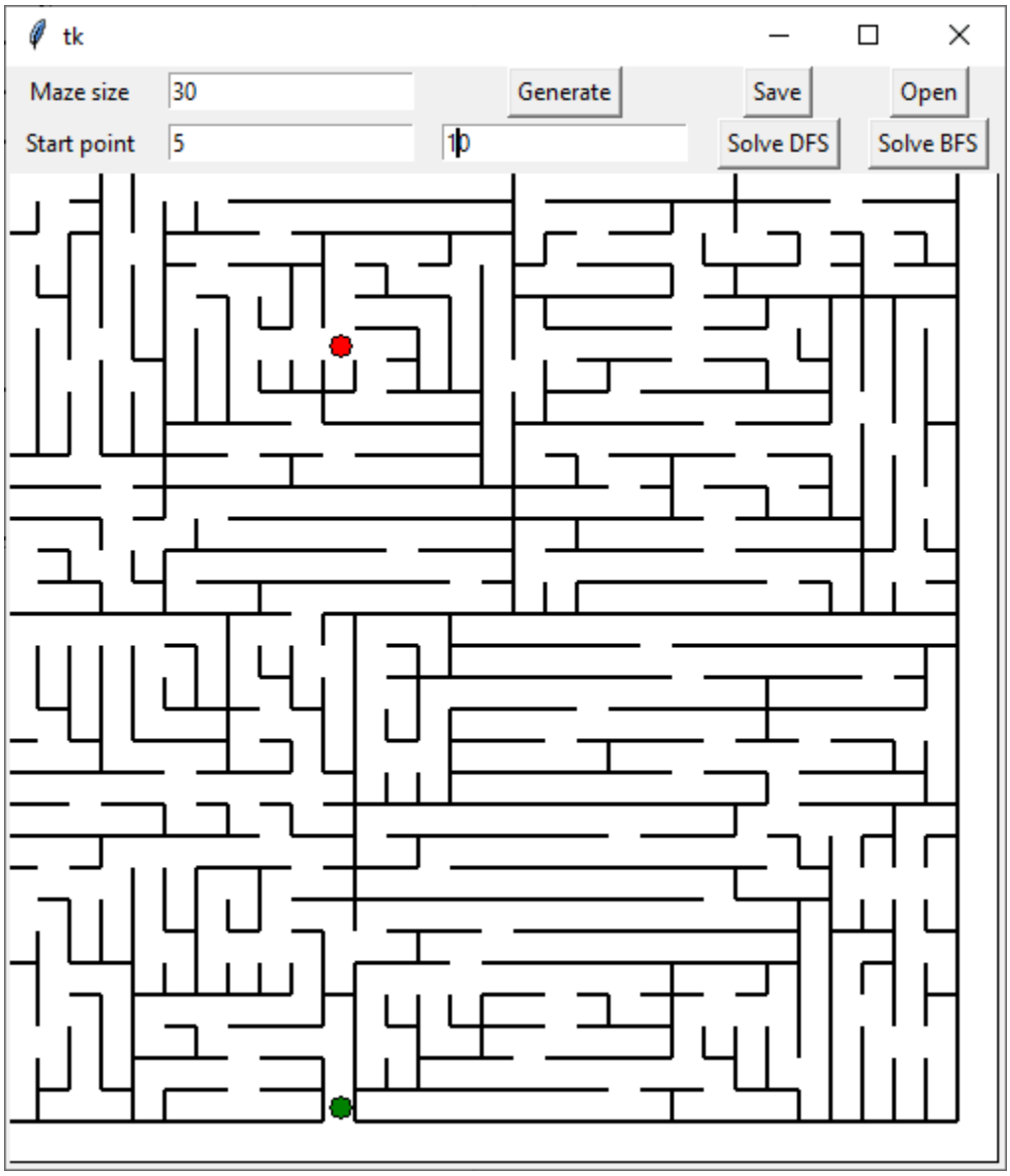

Figure 16 Maze-2: Size=30, start=(5,10), exit=(29,10)

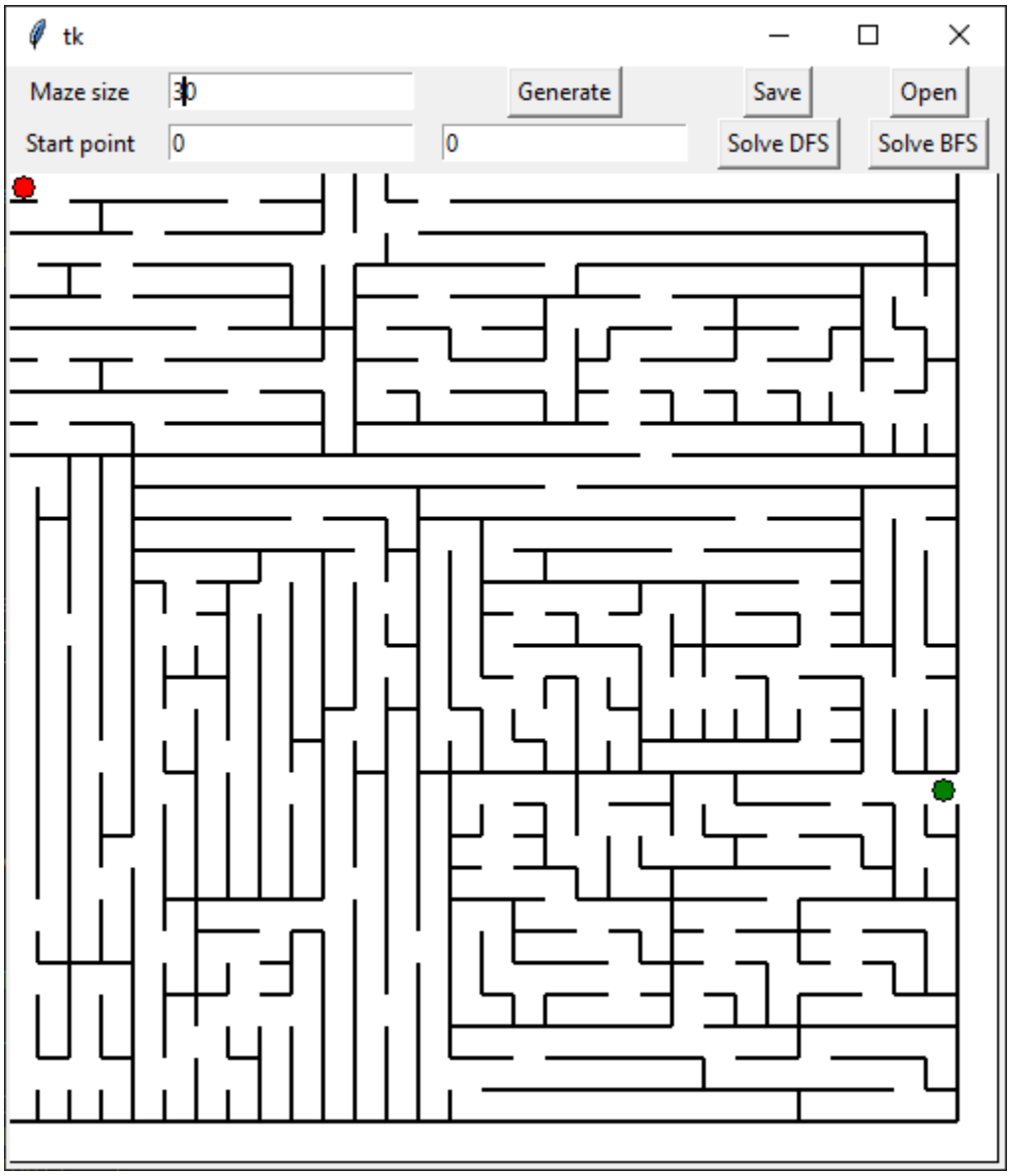

Figure 17 Maze-3: Size=30, start=(0,0), exit=(19,29)

The following two figures display the solved maze with the breadth first and the depth first algorithms.

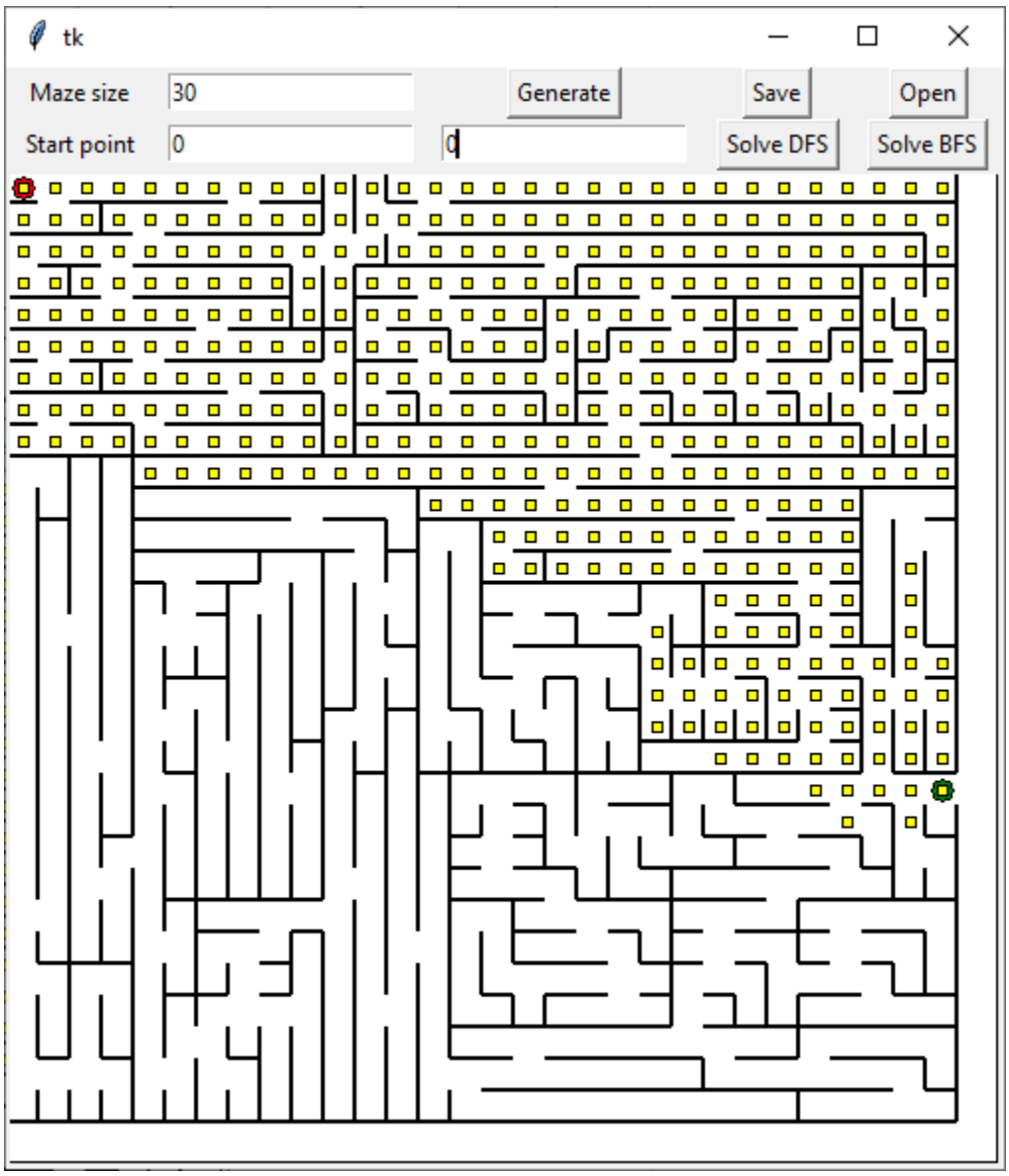

Figure 18 Solution for maze-3 using Breadth first search

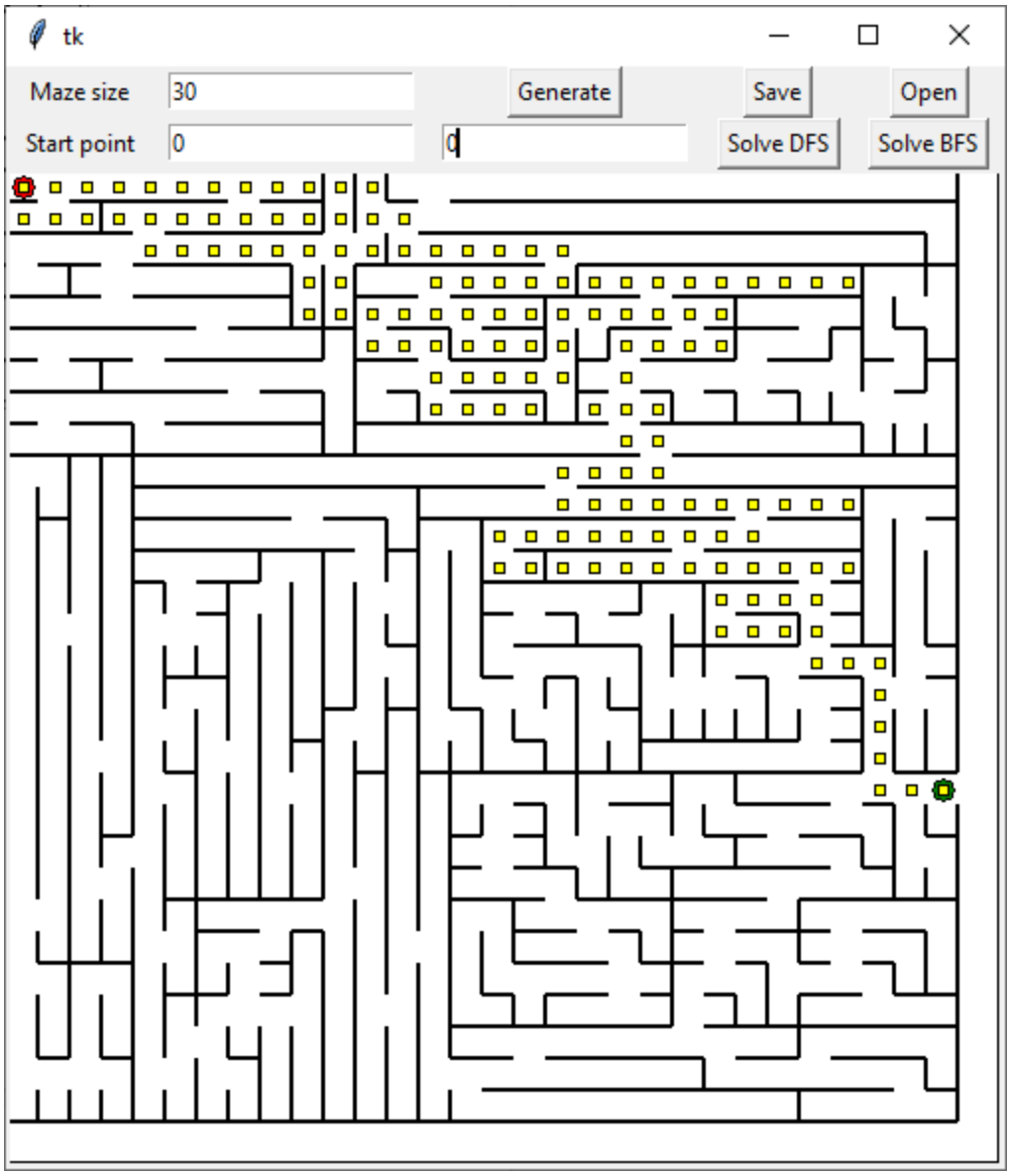

Figure 19 Solution for maze-3 with Depth-first search

6. Evaluation and conclusion

As an overall evaluation of this project, the goal was to try and implement different types of algorithms for maze generation and solution. The idea of reviewing the literature was to get an understanding of different approaches that would help to implement them. Personally, I would like to have implemented more complex algorithms to see the creation and solution of mazed in the animated form. Since I believe the animation gives a clear idea of

how they are different from the others. This is also the reason that I have made the solution of the maze animated. It was also possible to make the generation animated as well.

For the graphical user interface, there were other options like using the Pygame package instead of Tkinter, but as the focus of this course and the assignments were on the utilization of the Tkinter, I prioritized implementing the GUI in Tkinter. The GUI is not free of errors and weaknesses like if the maze size is given a large number, it might be difficult to see the paths in the maze. It could have been solved by making the size of the GUI flexible and dependent on the size of the maze.

I also would like to have implemented some unit testing methods to check the maze generation and solution, but as this was a small project with limited algorithms I could test it manually. In case of the addition of more algorithms, it would be something to consider as a priority.

All in all, I believe it was a good experience with such a fun subject like maze generation and solution and I have learned a lot during the implementation process.

7. References

1. Wikipedia. *Maze generation algorithm*. Available from: https://en.wikipedia.org/wiki/Maze_generation_algorithm#Randomized_depth-first_search.
2. ; Available from: https://www.differencebetween.info/difference-between-maze-and-labyrinth.
3. *Fundamentals of Programming and Computer Science*. [cited 2022; Available from: https://www.cs.cmu.edu/~112/.
4. *Maze Generation Algorithms - An Exploration*. 2020; Available from: https://professor-l.github.io/mazes/.
5. Williams, L.K. *Kruskal Algorithm Maze Generation*. 2019; Available from: http://www.integral-domain.org/lwilliams/Applets/algorithms/kruskalmaze.php.
6. Nunes, I., G. Iacobelli, and D.R. Figueiredo, *A transient equivalence between Aldous-Broder and Wilson's algorithms and a two-stage framework for generating uniform spanning trees.* arXiv preprint arXiv:2206.12378, 2022.
7. Adams, C., *Evolving Cellular Automata Rules for Maze Generation*. 2018, University of Nevada Reno.
8. Wikipedia. *Maze-solving algorithm*. Available from: https://en.wikipedia.org/wiki/Maze-solving_algorithm#Tr%C3%A9maux's_algorithm.
9. Sadik, A.M., et al. *A comprehensive and comparative study of maze-solving techniques by implementing graph theory*. in *2010 International Conference on Artificial Intelligence and Computational Intelligence*. 2010. IEEE.
10. Hendrawan, Y. *Comparison of Hand Follower and Dead-End Filler Algorithm in Solving Perfect Mazes*. in *Journal of Physics: Conference Series*. 2020. IOP Publishing.
11. Zhou, H.; Available from: http://users.eecs.northwestern.edu/~haizhou/357/lec6.pdf.
12. Bryukh, V.B.a. 2016; Available from: http://bryukh.com/labyrinth-algorithms/.