# Final Project 2020 Autumn

# Labyrinth Solver

*Candidate 252*

*Oslo Metropolitan University*

*Faculty of Technology, Art and Design*

*December 17th, 2020*

# Contents

# 1 Introduction

As my final project I have completed assignment number 3: Labyrinth Solver. The project description states that I was to write a program using Python which does the following:

- Designs a random 2D Labyrinth.
- Can find the exit from any point inside the labyrinth.
- Takes in user defined values to determine the size of the labyrinth.
- Takes in user defined values to determine where the solution shall start from when looking for the exit.
- The solving algorithm should utilize brute-force or Q Learning.

In accordance with this assignment, I have produced a Python 3 application with a graphical user interface (GUI) which lets the user input their preferences for size, solution start location, what algorithm should be used to solve the labyrinth and whether the graphics should be animated when drawing and solving the labyrinth. This document serves as documentation and a user-guide on the project and should without the reader having access to the source-code provide enough insight to evaluate the project. It is however recommended to have the source code available.

In short terms the project consists of a structure and code based on object-oriented principles, as I prefer OOP for larger projects such as this. For drawing the labyrinth, I have utilized a depth-first algorithm, while the solution is calculated using a either a recursive-walk or A* algorithm, depending on user choice. I will be going further into the details of different algorithms in chapter 2 and code implementations in chapter 3.

## 1.1 Running the application

Note that the entire project has been developed on computers running Windows 10, using the PyCharm IDE. The project has not been tested on other operating systems, nor by running it through a command line. It is therefore recommended to run the application through an IDE on a Windows computer to ensure no unexpected errors occur.

### Dependencies and Running

For drawing the labyrinth, the project utilizes a graphical library called pygame. This will have to be installed before trying to run the project, and can be done through the package installer for Python by running the following command:

*pip install pygame*

If the reader has any problems installing the library, see pygame's documentation on installation.

The application is then started by running Main.py.

## Graphical User Interface

Upon running the application, two separate windows will appear: The Control Panel and the Labyrinth Window.

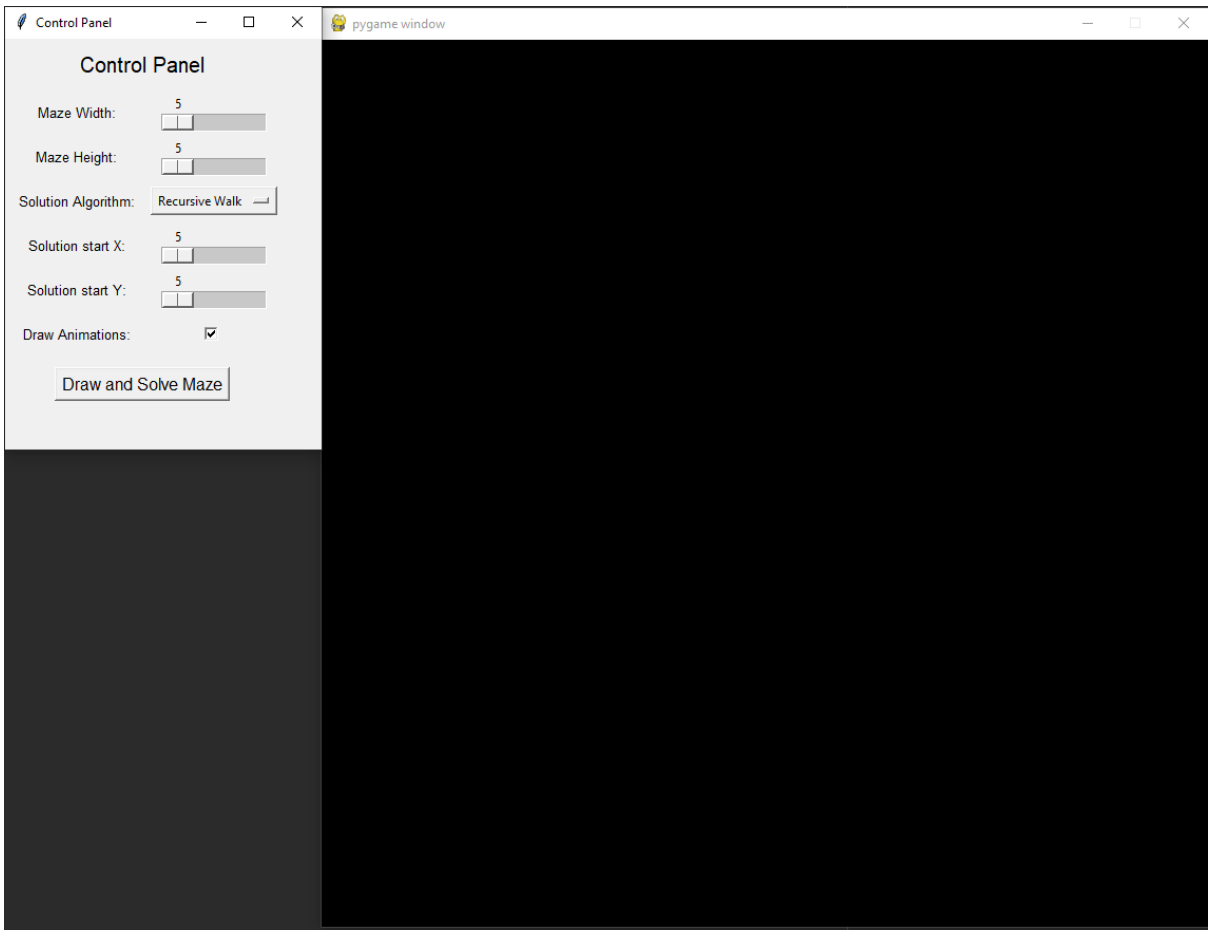


*Figure 1: The GUI as it initially appears when the application is first launched.*

At the ***control panel*** the user can use sliders to adjust the size of the maze, with height and width being restricted to values within 5 and 40, the upper limit being there to not exceed the window's size. Below the sliders is a dropdown menu for the user to select what algorithm should be utilized when solving the maze, followed by sliders for placing the starting coordinates for the solution algorithm. These sliders are bound to the current value of the maze's height and width sliders, as not to let the user place the starting point outside of the maze's boundaries. Finally, there is check-button for enabling and disabling drawing animations (the program takes considerably longer to execute when they are enabled, as the labyrinth is drawn almost instantly when they are disabled.). The labyrinth and its solution path will be drawn once the user clicks "Draw and Solve Maze". Note that a new labyrinth will be drawn each time the button is pressed.

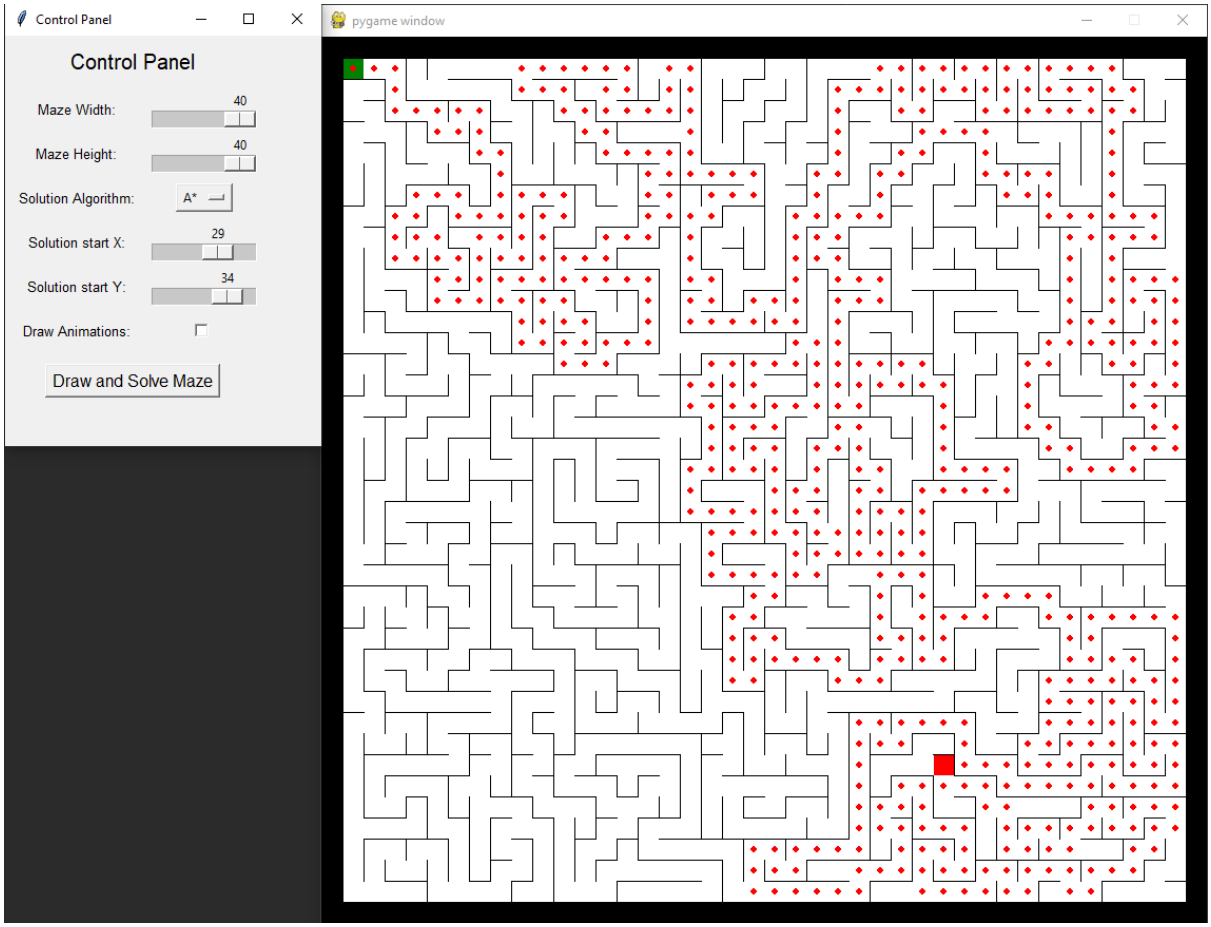


*Figure 2: The solved labyrinth after it is finished drawing. Red square marks the solution start coordinates, and the green square is the exit.*

## 1.2 Known Errors and Weaknesses

### Fixed window size and freezing

Since the project task itself does not include any requirements for a graphical user interface, making the GUI robust and flexible has not been a priority over code quality and structure in other parts of the project. The labyrinth window is therefore locked in size and trying to use other open programs while the labyrinth is being drawn may result in the program going into a non-responsive state until its task is completed, sometimes freezing up entirely. This is due to how animations are simulated by sleeping the program thread between draw-operations, and a more elegant solution was not prioritized. It is therefore recommended to not click outside the application window or actively use other applications once the drawing process has started and until it is finished.

### Recursive Depth Error

When using the recursive walk algorithm to solve a large maze (error has only been observed while having a maze of size 35+ in both dimensions), the program sometimes exceeds Python's recursive depth limit if the path back to the exit is properly complex. This has been left in to demonstrate one of the weaknesses of a recursive walk versus the A* algorithm. Note that error messages are not displayed in the GUI, only in the command line.

If the recursive depth is exceeded, it is apparent in the GUI if a path back from the solution starting point to the exit is not being drawn and should be the only case where this occurs[1].

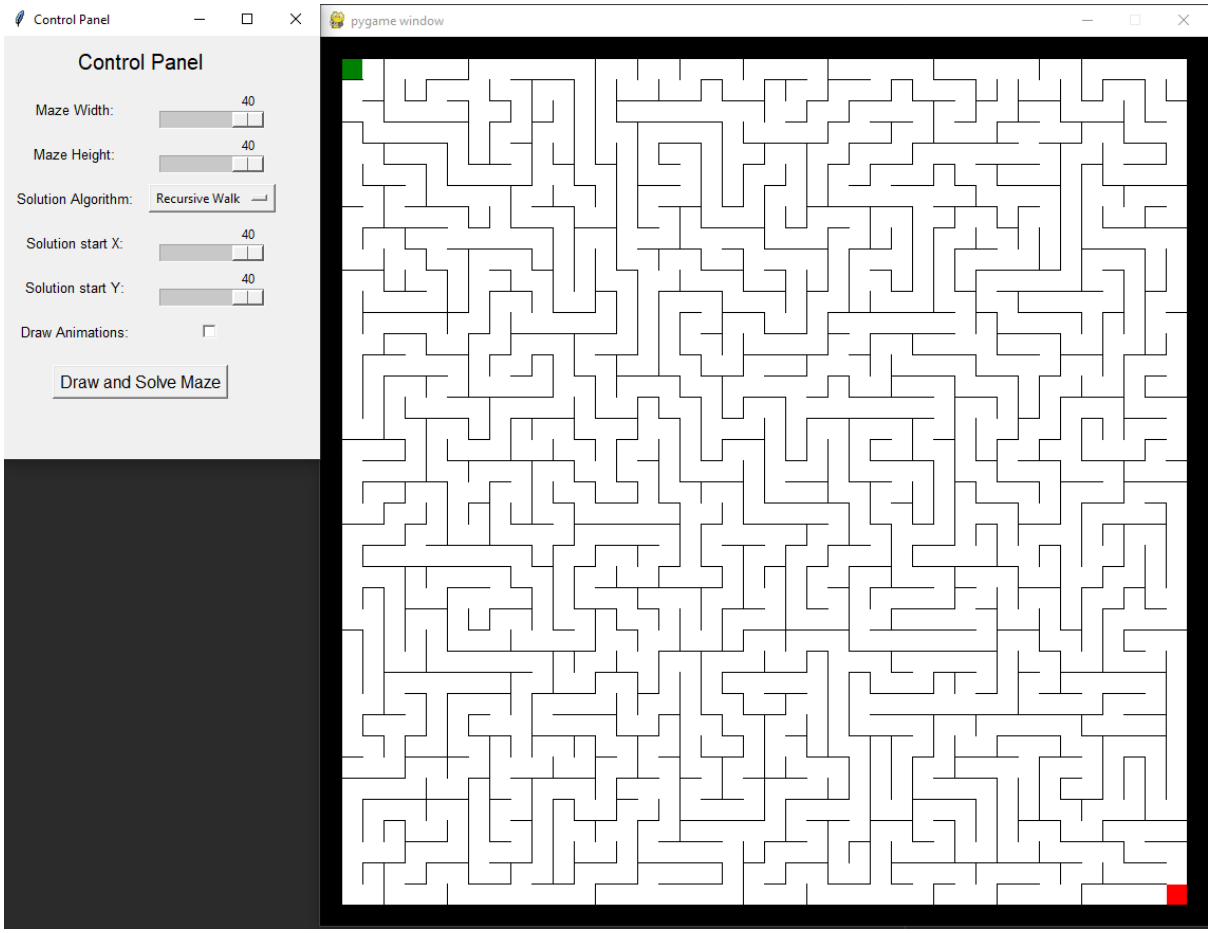


*Figure 3: What the GUI looks like in case of a RecursionError with the recursive walk algorithm*

# 2 Algorithms

When it comes to creating and solving the maze, I have utilized three algorithms which should be explored theoretically before diving into the code implementations.

## 2.1 Iterative randomized depth-first search

For generating the randomized maze an iterative, randomized depth-first search algorithm (Wikipedia, 2020) was implemented in order to make sure the maze would not be restricted to a certain size, as would happen if I had implemented a recursive version (which is exactly what happened with the recursive walk solution algorithm, see chapter 2.2). I first considered implementing a randomized version of Prim's (Wikipedia, 2020) or Kruskal's algorithm (Wikipedia, 2020) to generate the maze, as the general consensus of my research into viable algorithms for creating mazes indicated those to be the most effective when

[1] *Right before handing in the assignment, I learned that one may increase Python's recursion limit manually. It has been implemented in the code, and the error should not occur anymore.*

dealing with larger, random data structures. However, I was already familiar with depth-first algorithms for traversing graph structures and it proved to be effective enough for the project[2].

A depth first search traverses a graph by starting at an arbitrary node and then follows a branch of the graph until its end. The algorithm then backtracks to the last intersection of branches and then follows the other branches to their end. My depth-first search being iterative entails that keeping track of explored and unexplored branches are done by using a stack to keep track of unvisited nodes, removing nodes from the stack once all of the node's neighbouring paths have been explored to their end. Being randomized means that the sequence of which neighbour's path is explored is random.

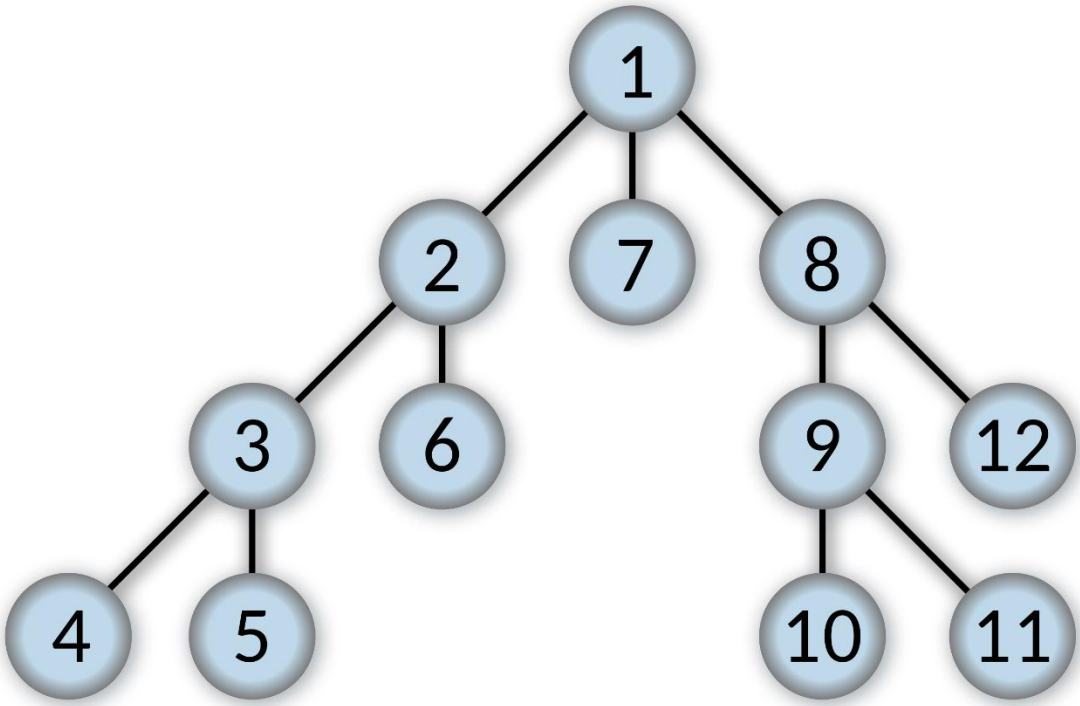


*Figure 4: Example of the exploration sequence in a pre-order, depth-first search in a tree-graph. Source: Wikipedia*

In the maze solver application, the algorithm starts at the cell[3] which is designated to be the maze's exit. Before creating the actual maze, a grid of cells has been generated and represents the graph to be traversed. Each cell has an attribute called walls, which is a dictionary[4] with a direction as key, and a Boolean as value, indicating whether the cell has a wall in the given direction. When initialized, each cell has a wall in all four directions, and as the algorithm traverses the grid, the walls are removed to create paths. The algorithm keeps track of visited and unvisited nodes, starting to backtrack whenever a node has no more unvisited neighbours.

[2] «Effective enough» being arbitrary defined as spending a maximum five seconds to generate and graphically draw the maze. It currently finishes in roughly three seconds after the draw button is pressed for a 40x40 maze.

[3] I will be using cells instead of nodes when explaining the maze implementations.

[4] List of key-value pairs

```
while (stack.length > 0)

    neighbours = list of unvisited neighbouring cells

    if (neighbours.length > 0)

        random_path = choose random neighbouring cell in neighbours list

        //check what direction the chosen neighbour has in respect to the current cell
        //remove appropriate walls
        //draw the current maze cell and its chosen neighbour
        //set the chosen neighbour to be the current cell and add to visited-list and stack

    else
        stack.pop
```

*Figure 5: Pseudo-code written while implementing the iterative depth-first algorithm*

When all possible paths are explored, the exit cell is popped from the stack, the algorithm terminates, and the maze is done being generated.

## 2.2 Recursive walk

A recursive walk is a brute-force type of algorithm to find the solution by walking through each cell in a pre-determined pattern and is a form of depth-first search algorithm, albeit done recursively instead of iteratively (as the name implies).

```
recursive_walk(x = -1, y = -1)

    if (x or y == -1)
        set x and y to the solution start coordinates

    if ((x, y) == exit_coordinates)
        return true //solution has found the exit

    else if (cell.already_visited)
        return false

    if  (unchecked neighbour in horizontal direction and recursive_walk(x +/- cell_size, y)) or
        (unchecked neighbour in vertical direction  and recursive_walk(x, y +/- cell_size))
        return true

    return false
```

*Figure 6: Pseudo-code written while implementing the recursive walk algorithm*

The implemented algorithm in my application starts searching from the given coordinates for solutions to start from. Before checking any neighbouring cells, it makes sure it is not already at the exit, and that the cell at the given coordinates (x, y) has not already been visited.

The algorithm then goes left, right, up, and then down when checking for adjacent cells without a wall between them and the current cell. If a neighbour is found, the function calls itself with the coordinates of the adjacent cell as parameters. Any inputs resulting in the function returning false will make the algorithm go up one level in the function call-stack and continue checking the neighbours at the previous intersection.

While recursive solutions are rather elegant in terms of code readability, they are not preferred when aiming for efficiency, with iterative solutions known to be more efficient in terms of time-complexity (Elton, 2016). In this particular use-case of solving a maze, the algorithm also sometimes exceeds Python's default recursive depth limit of 1000 when applied to a larger maze with a not so straight-forward exit path. While the limit can be adjusted, having to do so is generally considered a bad code smell and should be avoided (Python Central, 2016). However, to prevent a RecursionError from crashing the application, I have done so in this project.

## 2.3 A* Algorithm

A* (A Star) is a popular path-finding algorithm, often utilized in map applications and video game AI for finding optimal routes. The algorithm is performance-wise deemed to be very efficient, but is based on storing all nodes in memory, thus making it less optimal to run on devices with restricted memory, which is not a concern for modern computers and mobile devices.

The concept of A* is based around calculating the cost of moving from one position to another, with each node having a fixed value indicating how much it costs to move to that node. The algorithm aims to reach the given end point with a movement cost as low as possible by trying out different paths.

At each iteration, the algorithm determines what paths to continue extending based on the cost of the path, and the estimated sum of the entire end path. Mathematically, the algorithm tries to minimize

$$f(n) = g(n) + h(n)$$

where *n* is the next node on the path, *g(n)* is the cost of moving from the start node to *n* and *h(n)* is a heuristic approximation of the most optimal movement cost.

In order to implement A* in the maze application, the already existing cell class was required to hold g(n) and h(n) as attributes, and to know which Cell is the parent node on the last path it was included in (see figure 7).

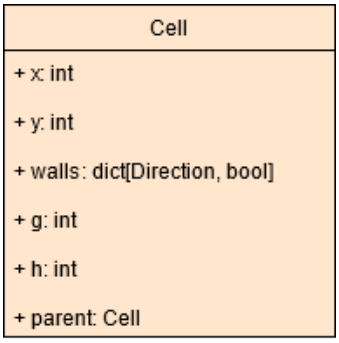


*Figure 7: Class-diagram including necessary attributes in Cell-class to implement A* algorithm*

The algorithm keeps track of two lists of cells: one for cells which has already been visited and one for those who has not, dubbed closed and open cells, respectively. When first called, the algorithm marks the cell located at the coordinates for the solution to start from as open. The list of open cells is then iterated over until it is empty, executing the following operations at each iteration:

1. Remove the current cell from open list, mark it as closed.
    a. If the current cell is the exit cell, terminate function and return a list of coordinates generated by tracing the parent cells back to solution start.
2. Find all adjacent cells
3. For each adjacent cell:
    a. That is not closed, is already marked as open and has a movement cost larger than the current cell's movement cost + one more step, update both cell's movement cost and set the current cell as the adjacent cell's parent.
    b. That is not closed and not marked as open, update both cell's movement cost and set the current cell as the adjacent cell's parent, mark the adjacent cell as open.

```
a_star()

    closed_cells = []
    open_cells = []

    while (open_cells.length > 0)

        cell = open_cells.pop()
        closed_cells.append(cell)

        if cell is exit_cell
            return path

        adjacent_cells = list of all neighbouring cells

        for (adjacent_cell in adjacent_cells)

            if (adjacent_cell not in closed_cells)
                if (adjacent_cell in open_cells)
                    if (adjacent_cell.g > (cell.g + movement_cost))
                        adjacent_cell.g = cell.g + movement_cost
                        adjacent_cell.h = calculate_heuristic_cost() //find a decent way to do this
                        adjacent_cell.parent = cell

                else
                    adjacent_cell.g = cell.g + movement_cost
                    adjacent_cell.h = calculate_heuristic_cost()
                    adjacent_cell.parent = cell
                    open_cells.append(adjacent_cell)
```

*Figure 8: Pseudo-code written when implementing the A* algorithm*

## 2.4 Simple Performance Test

To test whether one of the solution algorithms were superior in terms of execution speed, a simple test was performed where the execution time of the solution algorithms were measured one hundred times on different, random 40x40 mazes with the solution start coordinates set to (40, 40). The results are presented in figure 9:

| | Recursive Walk | A* |
|---|---|---|
| **Fastest** | 0.001000016 | 0.001999985 |
| **Slowest** | 0.010000285 | 0.011002540 |
| **Average** | 0.005201382 | 0.004901316 |

*Figure 9: Results from a simple timing test of the solution algorithms*

From these results it looks like the A* algorithm perform better on average but has worse edge cases than the recursive walk. However, due to the random nature of the test and negligeable differences, this test should be considered inconclusive on the matter of which algorithm is the most optimal for this project.

# 3 Project Structure and Implementation Details

This chapter will be covering the overarching project structure as well as the implementation details and thought process behind key classes and functions in the application. The project is split into four separate Python packages based on functionality, as well as having the GUI-class and Main-file located at root level. Each subchapter will cover one such package.

## 3.1 Values

The values package contains three dumb[5] enum classes dubbed Colour, Direction and SolutionType, and a script file named Constants. All these classes contain values like the RGB values for different colours or the root-coordinates for the maze, which all have in common that they are not supposed to change and are used in multiple places throughout the application.

These classes have been implemented to make the code more readable, and to ensure no errors are made by the programmer when utilizing them.

```
if chosen_neighbour == "RIGHT":
    self.__grid[(x, y)].toggle_wall("RIGHT")
    self.__grid[x + 20, y].toggle_wall("LEFT")

    draw_maze_cell(x, y, self.__screen, "RIGHT")

    x = x + 20
    visited.add((x, y))
    stack.append((x, y))
```

```
if chosen_neighbour == Direction.RIGHT:
    self.__grid[(x, y)].toggle_wall(Direction.RIGHT)
    self.__grid[x + Constants.CELL_SIZE, y].toggle_wall(Direction.LEFT)

    draw_maze_cell(x, y, self.__screen, Direction.RIGHT)

    x = x + Constants.CELL_SIZE
    visited.add((x, y))
    stack.append((x, y))
```

*Figure 10: Cut-out from a function with and without the usage of the value classes*

In Figure 10 we can on to the left see the code before utilizing the value classes. Here, the programmer must remember that left and right are always written in capital letters every time, and one spelling or capitalization error would lead to a bug in the program. The literal integer value 20 is also quite diffuse in terms of what it represents and is dependent on the programmer and whoever reads the code knows that it is supposed to be the size of a cell.

[5] The exact definition of a dumb class varies but is generally known as classes or files only containing attributes/constants and methods/functions to retrieve or change said values.

Typing in the wrong number would also lead to bugs. The example to the right in figure 10 demonstrates code that utilizes no value literals and is also more comprehensive in terms of what each value represents.

## 3.2 Maze

The maze package consists of three classes, Cell, Grid and MazeDrawer, which responsibilities entails keeping track of data associated with the maze and rendering it in the GUI.

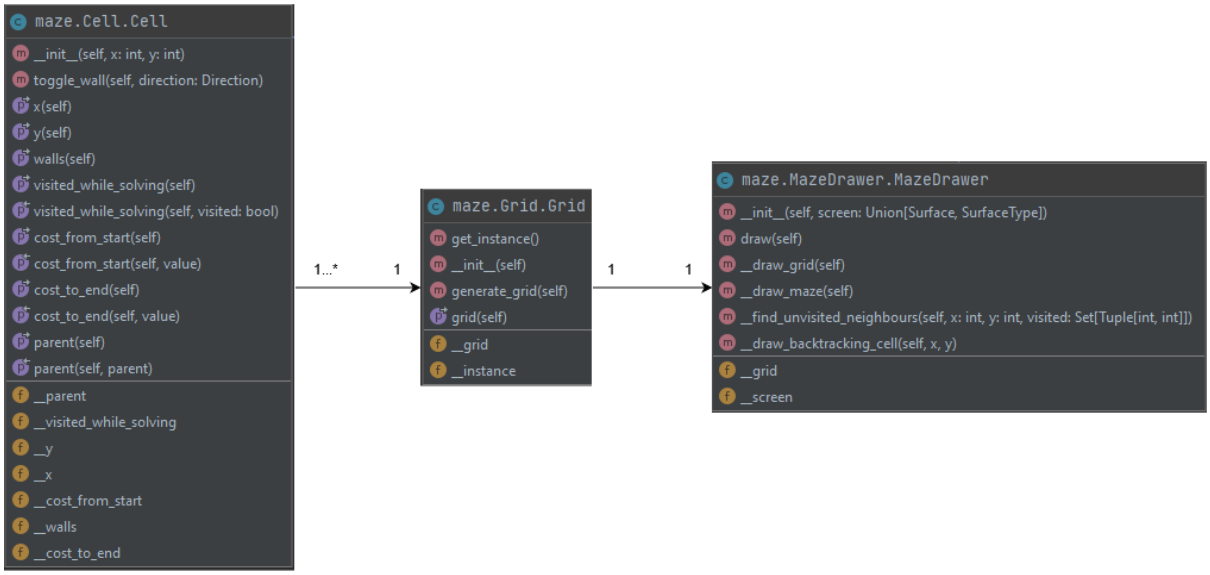


*Figure 11: Class diagrams for Maze package*

### Cell

A cell is the equivalent of a node in the maze graph and is a dumb class primarily responsible for holding values indicating its position in the grid and in which direction it has walls enabled. When initialized, the class receives its x and y-coordinates as parameters in the constructor, and all walls are enabled. Any other attributes the class has, like parent and cost_from_start, are related to the solution algorithms, and has already been explained in the subchapter detailing the A* algorithm. All those values are initialized as 0 for integers, false for Booleans and None for objects.

### Grid

Grid is a singleton class[6] responsible for generating a *n* x *m* grid of cells, represented by a dictionary with coordinates as key and a cell as value, holding that grid as a property. Having the grid object using the coordinates of each cell as a key when the Cell-objects themselves also holds the same coordinates as attributes results in storing the same values twice but is done so for the sake of convenience when working with the grid and cells in other parts of the application. Any attempts at alternative solutions like having a list of Cells instead of a

[6] If the reader is unfamiliar with this design pattern, read this article for insights.

dictionary, resulted in less readable code or extra iterations over the grid to retrieve the correct cells.

The class was created following the singleton pattern due to how multiple classes utilize the grid, and they are all supposed to be working on the same grid object at runtime. Having a singleton class enforces this.

### MazeDrawer

MazeDrawer is responsible for graphically rendering the maze. When first initialized the class retrieves an instance of the grid-class and uses it to generate a new grid. The drawing process is then started by calling the only public facing method named draw. This function first renders the grid to the GUI, runs the randomized depth-first algorithm to update the wall attributes of all the cell-objects in the grid and then filling out the maze's paths with a colour.

## 3.3 Utilities

The utilities package only holds one script file named DrawUtils, which contains two functions utilized by the Maze- and SolutionDrawer classes when performing graphical operations on the GUI. These functions are not of particular note, and the reader can examine their code and respective docstrings to get a grasp of their functionalities if relevant.

## 3.4 Solutions

The solutions package contains the classes named Solution, AStar, RecursiveWalk and SolutionDrawer, responsible for solving and then drawing the solution onto the maze in the GUI.

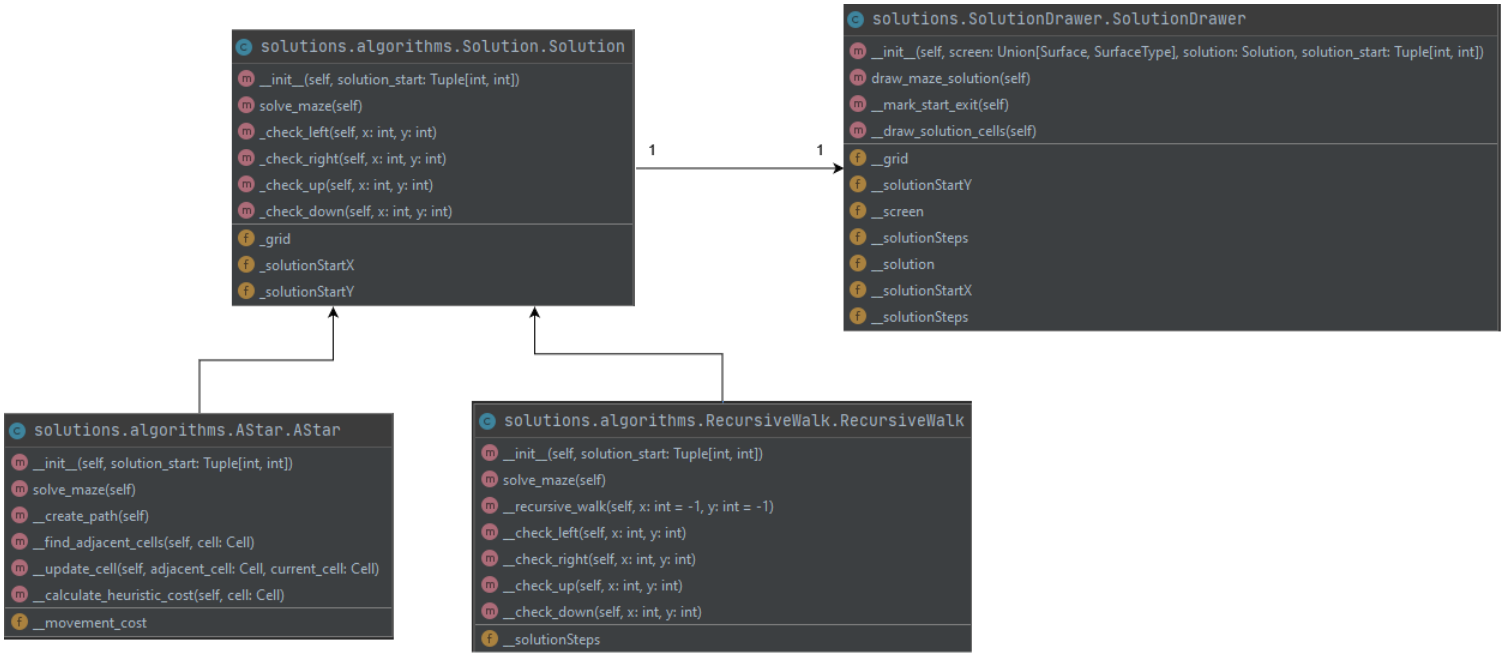


*Figure 12: Class diagram for Solutions package*

### Solution

In order to easily implement multiple solutions and dynamically inject them into the SolutionDrawer, an abstract base class for all solutions, aptly named Solution, was implemented. An abstract base class was implemented instead of an interface due to how it should hold values relevant for all solutions (grid and start coordinates) and have some implemented utility functions used in all solutions.

### RecursiveWalk and AStar

These classes contain the logic for solving the maze using the Recursive Walk or A* search algorithms, respectively. Aside from the heuristic cost for the A* algorithm being calculated by

$$h(n) = a * |(x_n - x_0)| + |(y_n - y_0)|$$

with $a$ being the movement cost for each cell, $(x_n, y_n)$ the coordinates of the cell $n$ and $(x_0, y_0)$ the coordinates of the exit cell, there is not much to add to the explanations of these algorithms which was not already covered in chapter 2.

In both classes the public facing method solve_maze functions as the entry point for the algorithms, with all the private methods being utility functions to make the code more readable and prevent repetition. Both classes return a list of coordinates with the solution path when they are done running their respective search algorithms.

### SolutionDrawer

The SolutionDrawer is responsible for graphically rendering the solution path, as well as the solution's start and end point onto the maze in the GUI.

When initialized, the class gets a Solution-object and its start coordinates passed as parameters in the constructor and retrieves an instance of the current grid. When calling the public facing method draw_maze_solution the start and exit points are drawn as a red and green square on their respective cells in the maze. Then the passed in solution objects solve_maze method is called to calculate the optimal path, and the returned list of coordinates is used to draw the path in the GUI.

## 3.5 Root

The projects root folder contains Main, which is used for running the project and has the line of code overriding Python's default recursion limit, and the GUI-class which contains all the code for creating and running the GUI.

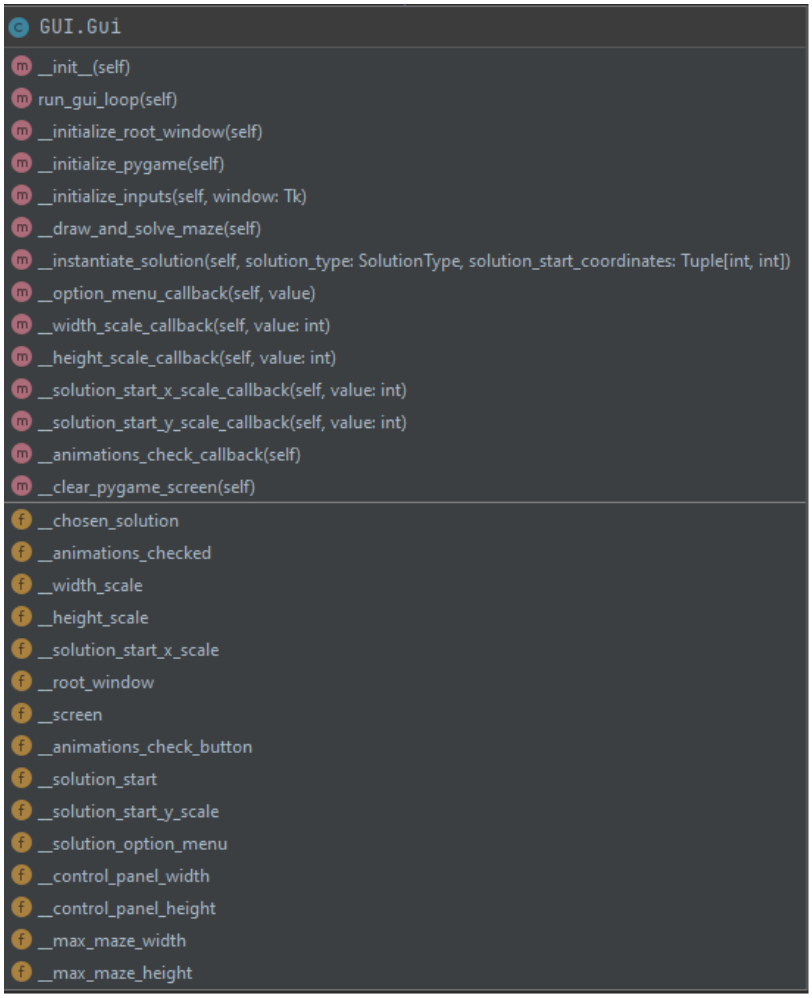


*Figure 13: Class diagram for GUI*

To create the GUI, a combination of the standard Python library tkinter has been used in combination with the third-party library pygame. Tkinter is excellent for creating simple user-interfaces and receive user-input, while pygame is dedicated to drawing graphics.

This class holds a lot of attributes and methods due to how one needs to keep track of the different input handles' return values, set call-back functions on most of them, and have attributes for the GUI state. Methods of note are:

- *initialize_inputs*
  - Initializes all the input handles and places them on tkinter's internal grid.

- *draw_and_solve_maze*
  - Creates MazeDrawer and SolutionDrawer objects, using return values from input handles where required, and calls their respective draw methods.

- run_gui_loop
  - Runs the loop which updates the GUI 30 times per second.

Most other functions are utility methods or call-back functions for the input handles.

# 4 Application Flow

Now that we a grasp of what the responsibilities of each class and their public facing methods are, we can go through the internal flow of the application to get a clear picture of how it is connected.

### Program Initialization

Upon running the program, an object of the GUI-class is initialized, and its run_gui_loop method is called. When creating the GUI-object, all of its elements are set to their default values defined in the class' attributes or in the Constants-file.

### User-input

When the user adjusts any of the input handles, call-back functions which performs the following actions are triggered:

Maze Width/Height

- GRID_WIDTH/HEIGHT and MAZE_WIDTH/HEIGHT in Constants are updated accordingly
- The handle for the corresponding dimension for solution start is updated to be capped at the current value of this handle.

Solution Algorithm

- The GUI-class attribute __chosen_solution is set to a SolutionType object with a string value corresponding to the solution algorithm chosen from the option menu.

Draw Animations

- Sets ANIMATIONS_ENABLED in Constants to true or false depending on whether the box is ticked (true) or not (false).

### Draw and Solve Maze

Most of the application's functionality is not triggered until the user presses the button labelled "Draw and Solve Maze", which triggers the GUI-class' draw_and_solve_maze method. Note that trying to change inputs while this function is executing will cause the GUI to freeze, requiring a restart of the application for it to function again.

Inside this method, a MazeDrawer object is initialized, which again initializes an instance of the Grid class and calls on it to generate a grid. The Grid class generates an *n* x *m* grid, where n = GRID_HEIGHT and *m* = GRID_WIDTH from Constants. MazeDrawers' draw-methods is then called, prompting the class to first draw a grid to the pygame window, and then the maze on top of the grid using the randomized depth-first algorithm.

Once the maze is finished drawing, an object corresponding to the class of the chosen solution algorithm is instantiated in the GUI class before being passed together with the solution start coordinates into the SolutionDrawer's constructor to initialize an object of its

type. When initializing, the SolutionDrawer retrieves the current instance of the Grid class and its current grid. SolutionDrawer's draw_maze_solution is then called, which first prompts it to mark the exit cell (coordinates found as ROOT_X and ROOT_Y in Constants) and the start cell (passed in solution start coordinates). It then calls the selected solution's solve_maze method, using it's returned list of coordinates to draw the red circles indicating the path from the start cell to the exit cell.

At this point the draw_and_solve_maze method in GUI is done executing, and the program is ready to take user-inputs again.

# 5 Conclusion

As a short self-evaluation, I will claim that I have completed the project in an orderly and thorough fashion, not only completing the written project tasks, but going above and beyond in some areas. I could have implemented even more different solutions for solving or generating mazes, but I chose to prioritize quality over quantity once I had implemented what is currently in the application. Writing proper unit or integration tests could also have been a great addition, but since I am not familiar with any testing frameworks in Python, this was not priority. The application is small enough that I feel my manual user tests checking whether all the implemented functionalities operates as intended should suffice.

As an ending note I would like to say that I am overall pleased with this project, it being my first large scale Python program which also incorporates a graphical user interface. I have been able to include a lot of what I already know from being heavily invested in Java and JavaScript, and at the same time learn about the inner workings of Python and popular search algorithms for graphs, of which I had no more experience other than the theoretical knowledge from the Algorithms and Data structures subject during my third semester as a bachelor's student.

The project has been exciting to work with and has opened my eyes for some of the flexibilities of Python as a programming language.

# References

Elton, D. C. (2016, February 8). *Recursion is slow.* Retrieved from moreisdifferent.com: http://www.moreisdifferent.com/2016/02/08/recursion-is-slow/

Python Central. (2016, October 10). *Resetting the Recursion Limit.* Retrieved from pythoncentral.io: https://www.pythoncentral.io/resetting-the-recursion-limit/

Wikipedia. (2020, November 30). *Depth-first search.* Retrieved from wikipedia.org: https://en.wikipedia.org/wiki/Depth-first_search

Wikipedia. (2020, December 12). *Kruskal's algorithm.* Retrieved from wikipedia.org: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Wikipedia. (2020, December 2). *Prim's algorithm.* Retrieved from wikipedia.org: https://en.wikipedia.org/wiki/Prim%27s_algorithm