# Defining & Understanding

Research

**Random Mouse Algorithm**

For this algorithm, the theoretical robot will traverse forwards until it faces a junction. Then, it will randomly choose one of the paths to traverse forwards until it faces a junction once more.

In terms of logic, this is the simplest approach to create a maze-solving algorithm. However, it is extremely inefficient especially in more complex mazes and relies greatly on time for processing. It should also be noted that theoretically, it has an infinitely approaching zero probability of failing to solve the maze even if given sufficient time.

Given its vastly uncertain run time, this algorithm will not be used in my approach of the assessment.

**Dijkstra's Algorithm**

In essence, Dijkstra’s algorithm works by finding the distance from one point in the maze to all other points and repeats this process for all points within the maze to find the shortest distance from the start to the end.

For this algorithm, every accessible point/junction within the maze is viewed as a node, if it is possible to traverse from one specified node to another, a path exists between the two nodes. Each node has a sequence of previous nodes recorded since the starting node (none if the node is the starting node). Two nodes may be the same point in the maze, but have different previous nodes, and therefore different paths to reach the node and possibly different lengths.

For initialisation, set the start node as the current node.

The following steps of the algorithm can be performed iteratively or recursively (though the latter is more common):

1. Calculate the distance from current node to all accessible nodes. Add these distances to the shortest found path from start to current node, to find the distances for each accessible node. Replace the distance with the newly calculated one and update previous nodes for the node if previous calculations yield greater distances.
2. Mark the current node as visited. The optimal path (of least length) is already used for the current node (as to be stated in step 4).
3. If the end node is marked visited, end the program. The current path is the shortest path from the start node to end node.
4. Otherwise, out of all unvisited nodes, select the node with the least distance from the start node as current node. If all unvisited nodes are inaccessible, end program; there will be no method of reaching the end node.

If the program is ended by the end node being marked visited, the shortest path is of distance that the end node is currently marked as, and the path is stored as previous nodes for the end node.

Displayed by the sheer amount of text under the heading of this algorithm, it is extremely complex, with the benefit of the ability to always find the shortest path to the exit of the maze if possible. In the case of this assessment, all accessible nodes will be equidistant, and it is not necessary to find the shortest path.

Given its complexity to implement, potentially lengthy runtime and unnecessary capabilities, this algorithm will not be used in my approach of the assessment.

**Dead-End Filling**

This algorithm works by filling in all dead-ends until only the correct path(s) remain.

In practice, the algorithm requires a loop to:

1. Search points in the maze that has only one entrance and one exit (i.e. a dead-end).
2. Replace the point with a wall.

The loop will end when all dead-ends – including those formed by walls filled in by the algorithm – are filled in, and only the correct path(s) from start to end remain.

This algorithm is simple and intuitive, while being easy to implement in code form. Despite its need for many iterations to check if all points are dead-ends, it is possible to optimise the later iterations by only checking for the formation of new dead-ends around previously filled-in dead-ends.

I believe that this is a simple algorithm with great potential, that also fits the requirements of this assessment well. Therefore, this is the algorithm I will use and modify in my approach.

# Planning & Designing

Problem Breakdown

**open\_maze(file)**

Opens the given file and returns its contents (the maze) as a 2D array.

**fill\_dead\_end(maze)**

Replaces all 0’s (spaces) in the maze surrounded by three 1’s (walls) or edges, with a wall.

This function should be looped until all possible dead ends are filled.

**create\_path(maze, filled\_maze)**

Finds all 0’s (spaces) in filled\_maze (maze with all dead ends filled), replaces their corresponding spaces with 5’s (paths) in maze.

This function should be called after all dead ends are filled.

Structure Chart

Algorithm Flowcharts

**Loading data from file and storing in 2D array**

**Finding the starting position within the maze**

**Finding empty square adjacent to current position**

# Testing & Evaluating

Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Maze No.** | **Expected** | **Outcome** | **Explanation** | **Fix** |
| 1 | For the correct path to be outputted. | The correct path was outputted. | As no dead-end filling had to take place, the program replaced all spaces with paths, and as there is only one possible path, it succeeded. |  |
| 2 | For the correct path to be outputted. | The correct path was outputted. | As no dead-end filling had to take place, the program replaced all spaces with paths, and as there is only one possible path, it succeeded. |  |
| 3 | For all dead-ends in *maze* to be replaced with walls. | Some dead-ends were filled, others not; noticeably those on the edges of *maze* were not filled successfully. | The approach of dead-end filling required counting the amount of edges and walls surrounding an empty space. For edges, the previous method relied on a *try, except* statement to detect whether an error would be produced when checking in a specific direction of a space. If an error was produced, there would be an edge in the given direction.  This approach is flawed, as Python supports negative indexes, and therefore will not produce IndexError: list *index* out of range when attempting to find *maze[row-1][column]* and *maze[row][column-1]*, where row and column are 0. | Instead of relying on errors, added an extra *if* statement to check whether the value of row or column is less than 0 or equal to the length of their respective lists.  If the condition holds true, there is an edge in the given direction. |
| Same as above | All dead-ends in *maze* were replaced with walls. | Previous fix was successful. |  |
| For *filled\_maze*, all dead-ends should be filled in with walls;  For *maze*, the correct path should be outputted. | *filled\_maze* was filled successfully;  all empty spaces from *maze* were marked as paths. | The function *create\_path(maze, filled\_maze)* takes two parameters, *maze* and *filled\_maze* respectively, and uses the coordinates of empty spaces in *filled\_maze* to generate valid paths in *maze* (as the size of the maze does not change in the process of dead-end filling).  The input order of the two positional parameters were swapped, leading to *filled\_maze* being outputted with all empty spaces from *maze* being filled as paths. | Provide the parameters in correct order. |
| Same as above | *filled\_maze* was filled successfully;  the correct path was outputted for *maze*. | Previous fix was successful. |  |
| The correct path should be outputted | In the output, all filled dead-ends were shown;  the correct path is (technically) outputted | As *maze* and *filled\_maze* were assigned to the same 2D list, they were modified together, and function *create\_path* essentially used empty spaces of *filled\_maze* as a reference to fill in *filled\_maze* itself…  Therefore, in the output all dead-ends were replaced with walls, and remaining empty spaces were replaced with paths. | Assign *maze* and *filled\_maze* to two separate calls of the function *open\_maze*. |

Evaluation

# Implementing

Logbook

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Progress** | **Screenshot (if necessary)** | **Sources (if available)** |
| 3/03/2023 | Assessment start  Create *Logbook* |  |  |
| 4/03/2023 | Complete *Research* section of documentation |  | [Random mouse algorithm](https://en.wikipedia.org/wiki/Maze-solving_algorithm#Random_mouse_algorithm)  [Dead-end filling](https://en.wikipedia.org/wiki/Maze-solving_algorithm#Dead-end_filling)      A conversation with *Zen Syahrizal*, who plans to use Dijkstra’s algorithm in his approach |
| 6/03/2023 | Complete *Problem Breakdown* section of documentation |  |  |
| 8/03/2023 | Edit *Problem Breakdown* section to reduce number of functions and increase efficiency   * Removed create\_wall and remove\_wall as they loop through every element one extra time   Implement all functions in code, see *Testing* section entries 1~4 for problems encountered | Works flawlessly |  |
| 11/03/2023 | Further optimisation of code   * Store coordinates of all empty spaces in a list to reduce iterations when searching for empty spaces * Create function check\_dead\_end(maze, row, column) to enhance readability * Added comments and removed redundant code to enhance readability   Drafting of a method to determine a single path |  |  |
|  |  |  |  |
|  |  |  |  |