Documentation Tony Wang 11SDD.02

Defining and Understanding the Problem

Random Mouse Algorithm

Random mouse algorithm is a simple algorithm involving trial and error, where the path would be constructed through the algorithm going in a straight direction until it is met with an intersection of junctions where it makes a random or preset sequential decision from left right or forward but never backwards. The only instance where the algorithm would move backwards would be if confronted with a dead end where the main primary directions of left right and forward are invalid, the algorithm would then proceed backwards. Random mouse algorithm is extremely inefficient and requires a long time for the algorithm to solve a maze due to the heavy reliance upon trial and error hence because of this, it will not always be able to quickly find the shortest path possible.

DFS Algorithm - Depth First Search

Depth first search also commonly written as DFS is an algorithm used to investigate and construct paths/adjacent connections with each node. DFS is an accurate algorithm, it consists of a recursive function/loop that is reliant on a stack. DFS functions through having a function inside the algorithm which is capable of returning valid adjacent nodes. Finding the valid adjacent nodes will consist of a sequential set of connections/relations and conditions, such that in the case of a 2d list maze, rows can be seen as the y and columns can be seen as the x, applying vector directions such as row-1 unit, row+1 unit, column-1 unit, column+1 unit would be able to shift it to the nearest adjacent node up, down, left and right and such conditions may be that certain nodes can not be accepted such in the case of a maze the domain and range of the plain as well as the walls. After being able to retrieve the valid adjacent nodes, It will append the first adjacent node it discovers (depends on the sequence of vector directions that are applied) to a stack, a stack may be an array such as a list. The stack would hold the each node and with depth first search it delves into one specific branch of nodes fully to construct a path unless the branch is subjected to a dead end it will apply the LIFO structure that a stack possesses meaning the last in first out, hence this structure would in a sense backtrack to the previous valid node and apply a different direction vector with the added condition of that the dead end node will not be visited again by checking if the node has already been visited. Hence the recursive DFS would apply LIFO and backtrack from the depth of the branch of nodes with removing the visited nodes until it finds another valid path on the branch. Therefore the invalid or dead end nodes will be removed/popped from the stack and only the valid path through the branches of nodes will be left within the stack. The main flaw of depth first search for a maze searching algorithm would be that it would not necessarily discover the shortest path as that would be up to the sequence of movement vectors set, it would not be as time efficient either as it would go deep searching one branch of nodes fully at a time hence it would have to backtrack from every dead end branch to output the final path/solution.

BFS Algorithm - Breadth First Search

Breadth First Search algorithm commonly written as BFS, is an algorithm to construct a path between adjacent nodes. BFS functions through iterating with a queue. A queue utilises a FIFO data processing structure being that first in first out. BFS checks if a path exists between nodes, finding all possible adjacent paths/nodes at once and appending it to a queue. Finding the valid adjacent nodes will consist of a sequential set of connections/relations and conditions, such that in the case of a 2d list maze, rows can be seen as the y and columns can be seen as the x, applying vector directions such as row-1 unit, row+1 unit, column-1 unit, column+1 unit would be able to shift it to the nearest adjacent node up, down, left and right and such conditions may be that certain nodes can not be accepted such in the case of a maze the domain and range of the plain as well as the walls. Recursively with each path appended to the queue, the queue will pop the 0th which is the first path in the queue (FIFO) and then add another valid node to the end of the original path which has been popped.

The queue will consist of paths in the form of lists. Each new path is constructed with each new found valid adjacent node which is appended to its respective original path. Hence it would cater for the case of multiple diverging paths from one node.

In the case of a dead end, the new path will not be appended and hence that path towards the dead end will be dropped/removed as a valid path. Therefore recursively this will leave only the valid paths. The advantage of BFS is that as soon as the first possible valid path is created, the recursive loop/function would be terminated to return the path before the longer path is able to be constructed on the next iteration of the queue within the recursive function. Hence it would be able to output the shortest path with the least hops/1 adjacent node connections.

Therefore, BFS would be the most efficient algorithm to solve the path of a maze with dead ends, retrieve the shortest path and the fastest something which the previous random mouse and DFS algorithms are incapable of doing. The advantage with BFS is that since it simultaneously advances through the paths/maze, it is able to discover dead ends and drop/remove those paths as it proceeds through the maze, save on speed as DFS only focuses on one branch at a time which through performing BFS it is checking multiple layers/branches hence is able to discover shortest path. But potentially it would not be able to pick up a path within a maze that has high complexities of deep branches/paths from each node as fast as a DFS algorithm.

Planning and Designing

Problem breakdown

Maze path finder algorithm

Load maze

- Create directory for the python file to import the maze text file
- Import the text file as read()
    - Create rows within the imported list through constructing the elements of the list by splitting at every new line character
    - Create a maze = [] list
    - For every row in imported list, split the row and append that to the maze list
        - Creating a 2d list, list of rows which contain a list of elements (columns)

Map out the maze - Return and define values that would affect the conditions of the maze

- Define length of maze to return the amount of rows
    - Stating the range
- Define the length of the maze's row to return the amount of columns
    - Stating the domain
- Enumerate each row and column into display the indexes
    - Constructing the coordinate of the start = 5 with the row and column indexes

Implementing Breadth First Search Algorithm

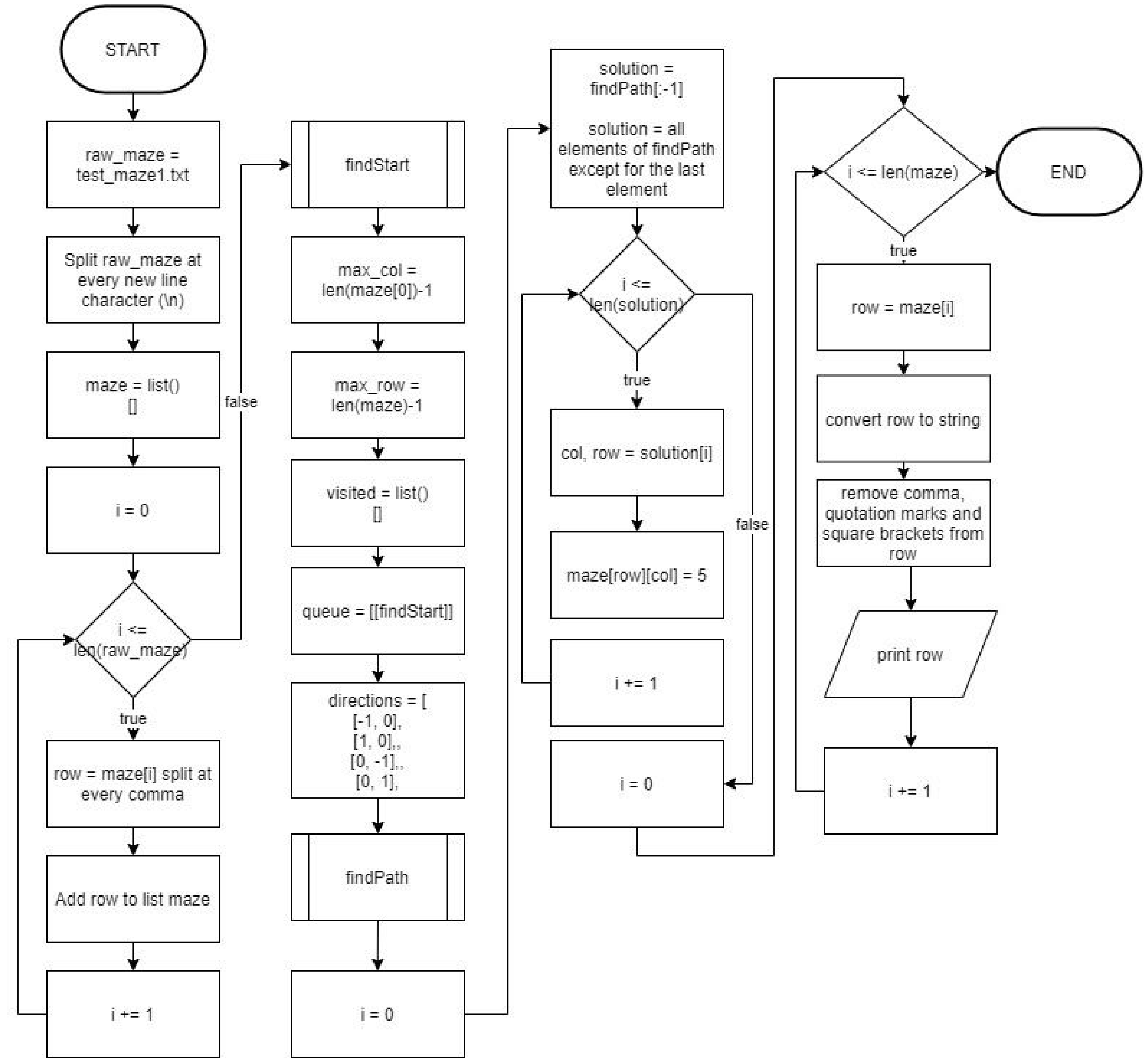
- Define a queue as a list
    - Place the start coordinate in a list within the queue so it will be processed first in the FIFO structure
- Define visited as an empty list
- Define movement vectors
    - Up : row index - 1, column index + 0
    - Down : row index + 1, column index + 0
    - Left : row index + 0, column index - 1
    - Right : row index + 0, column index + 1
- Construct a recursive function/loop
    - Whilst the queue has more than 1 path stored it in it
        - Path is the first element (list) of the queue (FIFO) and it is popped from the queue. Path is represented as a list
        - Node is the last appended element within the path (the starting point)
        - Prevent visiting the same node multiple times, hence that could lead to infinite loops and getting stuck in a dead end.
            - Verify that the node input is not the result of a node that has been already visited. Only pass if it is not in the list of visited nodes

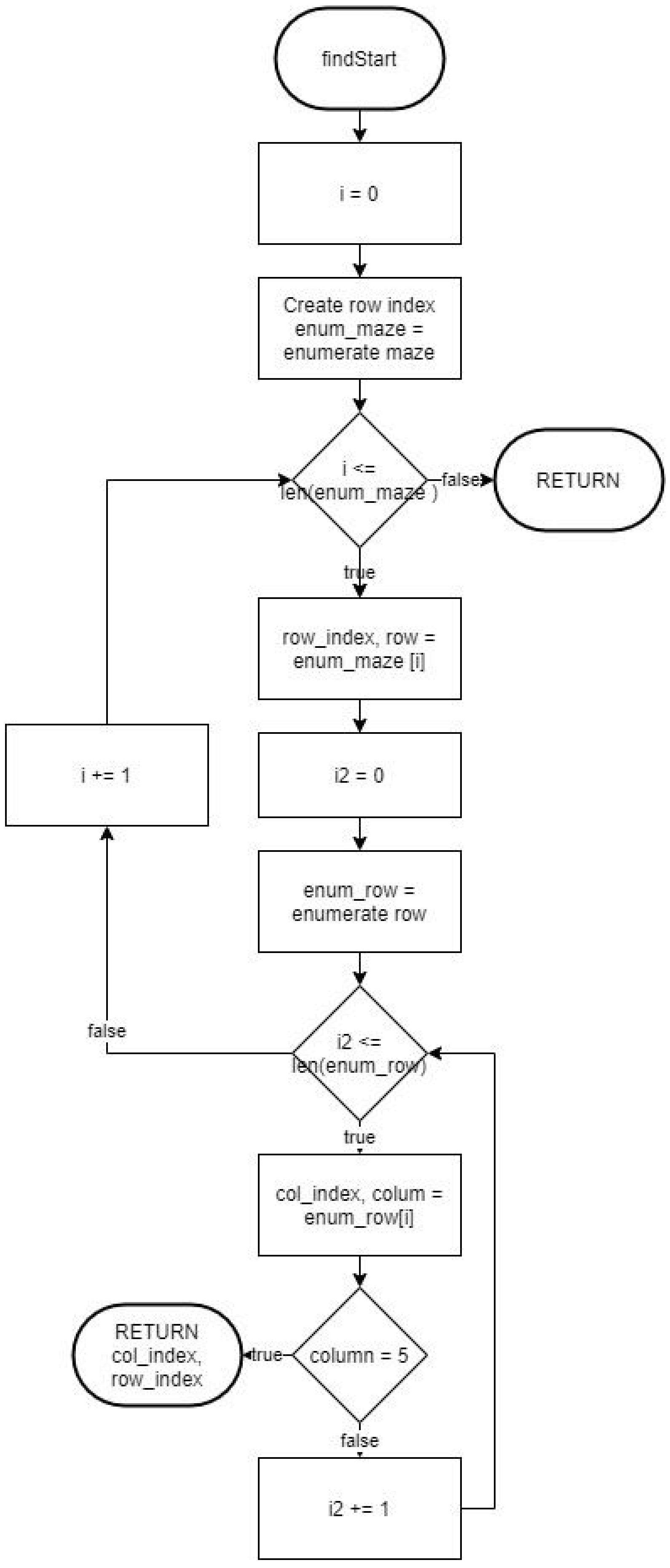
- With the current path, use the last appended node to find all the possible adjacent nodes
    - Find all valid adjacent nodes through applying the movement vectors
        - Validate the result of the vectors with the conditions such as the bounds of Domain and Range as well as the Wall = 1
- For every adjacent node, attach the adjacent node to the end of the path to construct a new path.
    - Check if the adjacent node is the end of the maze
        - If it is the end of the maze, return the new path and break the recursive loop/function
- Attach the new path to the queue
- Store/Append the node to the visited list

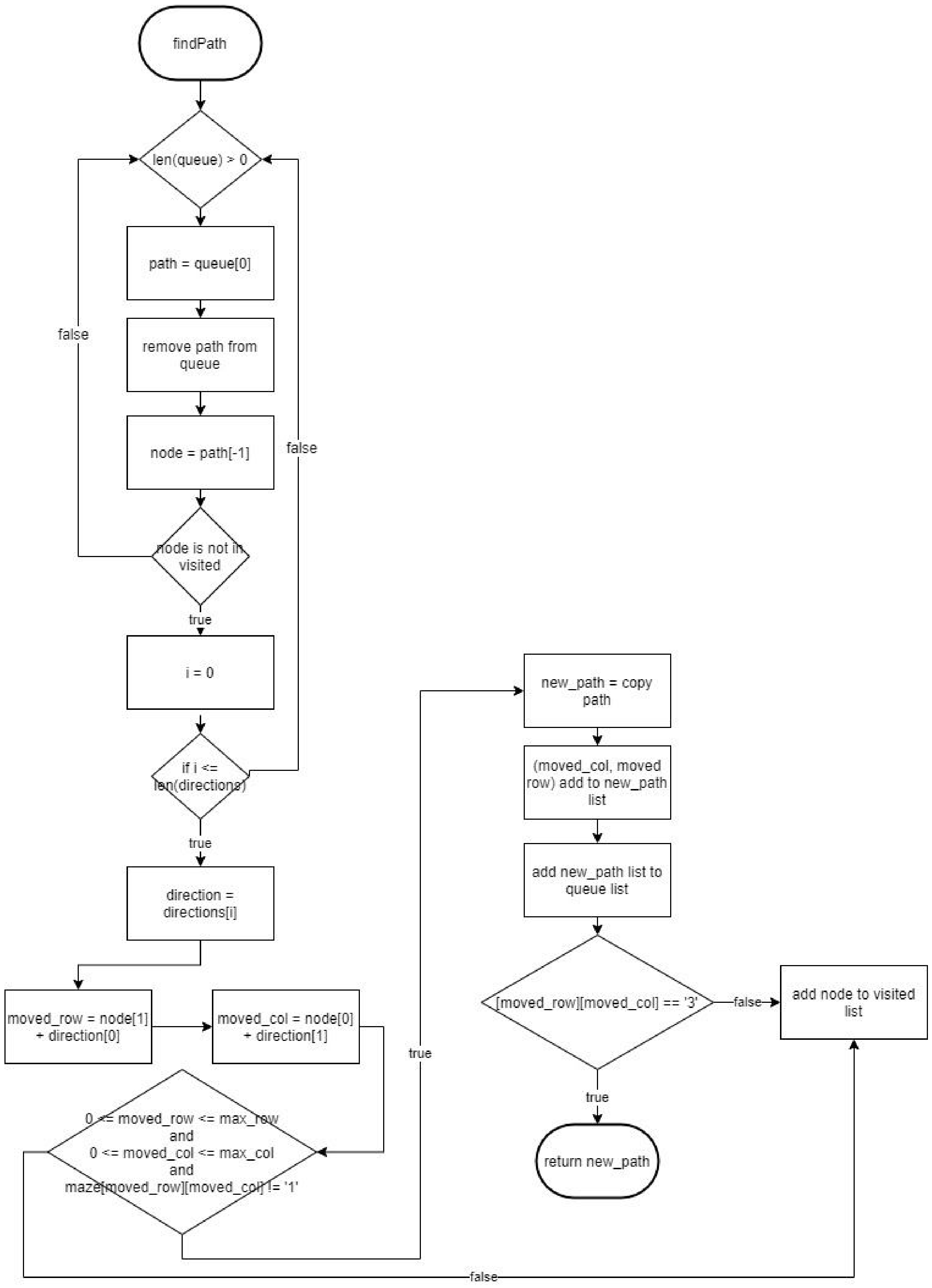
Displaying the solution output

- Output the maze path within the maze whilst keeping the finish node as ‘3’
    - For every coordinate in the solution returned from the BFS recursive algorithm except the last one appended which is the finish.
        - Extract the column and row index from the tuple (coordinate)
            - In the original maze replace the coordinate with 5 to create a path of fives
                - Find the node in the path through applying which list the coordinate of the node is in maze[row index][column index]
    - Print each row of the list in a grid like format
        - Transform each row of the list into a string and format it into a grid like row by removing the quotation marks, commas and square brackets
        - Print the formatted row

Algorithm flowchart

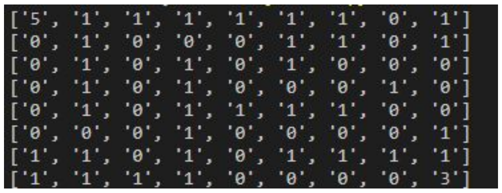



findStart
i = 0
Create row index enum_maze = enumerate maze
i <= len(enum_maze)
false
RETURN
true
row_index, row = enum_maze [i]
i2 = 0
enum_row = enumerate row
i += 1
i2 <= len(enum_row)
false
true
col_index, colum = enum_row[i]
column = 5
true
RETURN col_index, row_index
false
i2 += 1


findPath
len(queue) > 0
path = queue[0]
remove path from queue
node = path[-1]
node is not in visited
false
true
i = 0
if i <= len(directions)
false
true
direction = directions[i]
moved_row = node[1] + direction[0]
moved_col = node[0] + direction[1]
0 <= moved_row <= max_row and 0 <= moved_col <= max_col and maze[moved_row][moved_col] != '1'
true
false
new_path = copy path
(moved_col, moved row) add to new_path list
add new_path list to queue list
[moved_row][moved_col] == '3'
false
add node to visited list
true
return new_path

Implementing

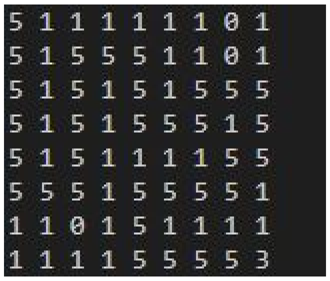
Date	Description Log	Media

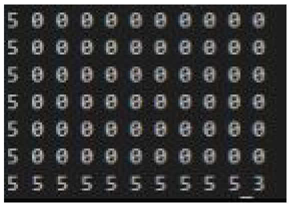
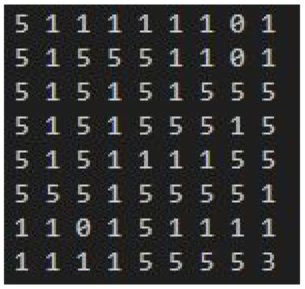
03/04/20	Constructed maze text file importing system.
	
	Formatter, constructed a 2d list which is used as a maze
		
		`['5', '1', '1', '1', '1', '1', '1', '0', '1']`
		`['0', '1', '0', '0', '0', '1', '1', '0', '1']`
		`['0', '1', '0', '1', '0', '1', '0', '0', '0']`
		`['0', '1', '0', '1', '0', '0', '0', '1', '0']`
		`['0', '1', '0', '1', '1', '1', '1', '0', '0']`
		`['0', '0', '0', '1', '0', '0', '0', '0', '1']`
		`['1', '1', '0', '1', '0', '1', '1', '1', '1']`
		`['1', '1', '1', '1', '0', '0', '0', '0', '3']`

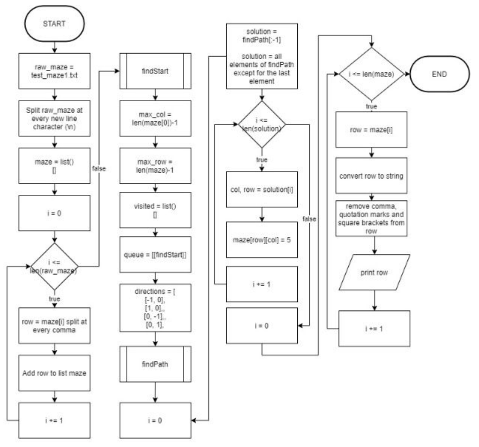
03/04/20	Researching algorithms Random mouse movements Sources: http://roboscience.org/book/html/Navigation/Mazes.html Adv - Simple recursive based algorithm - Trial and error - Easy to implement Flaws - Highly inefficient, slow, unsystematic DFS Sources: https://www.youtube.com/watch?v=W9F8fDQj7Ok&t=453s Adv - More accurate than the random mouse algo, more systematic Flaws - Inefficient for maze solving, doesn't always construct the shortest/fastest path BFS Sources: https://www.youtube.com/watch?v=KiCBXu4P-2Y&t=269s Adv - Systematic, always outputs the fastest/shortest path - Efficient for maze solving, searches based off of layers instead of depth which may cause it to waste time in complex rabbit	

	hole situations Flaws: - More difficult to implement	
03/04/20	Evaluating algorithms Sources: https://www.youtube.com/watch?v=TIbUeeksXcI&t=130s BFS, clearly most efficient and shortest path Video explanation of BFS and how to apply it to maze.	
03/04/20	Construction of Recursive BFS maze solving algorithm Update log: 1. Remodelled vectors - Old: 4 If statements including conditions - New: A list of direction vectors as it is the only variable that is changed when performing each movement. - For loop taking in each direction vector from the direction vector list - Improved readability and compaction of code. 2. Path is returning to original start point when applied the following maze: - 5 0 0 0 0 0 0 - 0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 - 0 0 0 0 0 0 3 - Testing the functionality of my BFS algorithm's ability to construct the shortest path. - Output is shown in Media cell - Problem: - Node was being appended to the visited list was being applied to the first direction vector - Solution:	3. Output - Path: [(0,0),(0,1),(0,0)]

	- Node was appended to the visited list after it has been applied to all 4 directional vectors. Outside the for loop that utilises the directional vector.	
03/04/20	Completion of BFS maze path finder with returning coordinates in the format of a list of tuples	[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 5), (2, 5), (2, 4), (2, 3), (2, 2), (2, 1), (3, 1), (4, 1), (4, 2), (4, 3), (5, 3), (6, 3), (6, 2), (7, 2), (8, 2), (8, 3), (8, 4), (7, 4), (7, 5), (6, 5), (5, 5), (4, 5), (4, 6), (4, 7), (5, 7), (6, 7), (7, 7), (8, 7)]

03/04/20	Constructed output formatter.
	
	Going to each coordinate on the original maze and replacing each node as a 5 to construct a path of 5s
		
		`5 1 1 1 1 1 1 0 1`
		`5 1 5 5 5 1 1 0 1`
		`5 1 5 1 5 1 5 5 5`
		`5 1 5 1 5 5 5 1 5`
		`5 1 5 1 1 1 1 5 5`
		`5 5 5 1 5 5 5 5 1`
		`1 1 0 1 5 1 1 1 1`
		`1 1 1 1 5 5 5 5 3`

03/04/20	Testing for shortest path and dead ends
	
	Applied 1. Discover shortest path
	- 5 0 0 0 0 0 0
	- 0 0 0 0 0 0 0
	- 0 0 0 0 0 0 0
	- 0 0 0 0 0 0 0
	- 0 0 0 0 0 0 0
	- 0 0 0 0 0 0 3
	
	Applied 2. Handle dead ends
	- 0 1 1 1 1 1 1 0 1
	- 0 1 0 0 0 1 1 0 1
	- 0 1 0 1 0 1 0 0 0
	- 0 1 0 1 0 0 0 1 0
	- 0 1 0 1 1 1 1 0 0
	- 0 0 0 1 0 0 0 0 1
	- 1 1 0 1 0 1 1 1 1
	- 1 1 1 1 0 0 0 0 3
		Display of valid solution to
		
		Maze 1.
		
		
		`5 0 0 0 0 0 0 0 0 0 0`
		`5 0 0 0 0 0 0 0 0 0 0`
		`5 0 0 0 0 0 0 0 0 0 0`
		`5 0 0 0 0 0 0 0 0 0 0`
		`5 0 0 0 0 0 0 0 0 0 0`
		`5 0 0 0 0 0 0 0 0 0 0`
		`5 5 5 5 5 5 5 5 5 5 3`
		
		Maze 2.
		
		
		`5 1 1 1 1 1 1 0 1`
		`5 1 5 5 5 1 1 0 1`
		`5 1 5 1 5 1 5 5 5`
		`5 1 5 1 5 5 5 1 5`
		`5 1 5 1 1 1 1 5 5`
		`5 5 5 1 5 5 5 5 1`
		`1 1 0 1 5 1 1 1 1`
		`1 1 1 1 5 5 5 5 3`

03/04/20	Constructed algorithm flowchart Sources: Draw.io https://app.diagrams.net/ Main Subprocesses/Functions - findStart - findPath	 START raw_maze = test_maze1.txt Split raw_maze at every new line character (\n) maze = list() [] i = 0 i <= len(raw_maze) false true row = maze[i] split at every comma Add row to list maze i += 1 findStart max_col = len(maze[0])-1 max_row = len(maze)-1 visited = list() [] queue = [[findStart]] directions = [[-1, 0], [1, 0],, [0, -1],, [0, 1], findPath i = 0 solution = findPath[:-1] solution = all elements of findPath except for the last element i <= len(solution) true false col, row = solution[i] maze[row][col] = 5 i += 1 i = 0 i <= len(maze) END true row = maze[i] convert row to string remove comma, quotation marks and square brackets from row print row i += 1

Testing and Evaluating

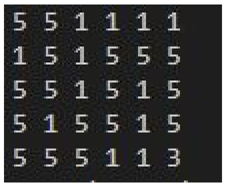
Note:

Through testing, I've directly dropped in the list within the code, thus having to change the string representation of numbers(nodes) into integers. Whilst the submission copy will only check against string representation as it is important a comma'd list of lines.

```
maze1 =     [[5, 0, 1, 1, 1, 1],
             [1, 0, 1, 0, 0, 0],
             [0, 0, 1, 0, 1, 0],
             [0, 1, 0, 0, 1, 0],
             [0, 0, 0, 1, 1, 3]]
```

- Expected Case:
  - BFS maze path finder
    - Locate start point via nested for loop and enumerating row and column indexes
      - Output = (0,0)
    - Only one possible path, one path needed in BFS with only adding 1 direction vector at a time, 1 new path created at a time. No paths are dropped due to no dead ends being present.

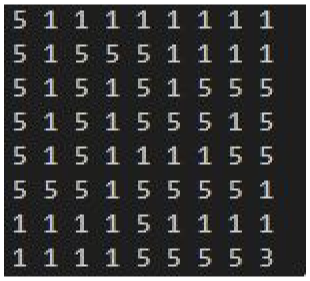
- Experimental Case:
  - Output



```
5 5 1 1 1 1
1 5 1 5 5 5
5 5 1 5 1 5
5 1 5 5 1 5
5 5 5 1 1 3
```

```
maze2 =     [[5, 1, 1, 1, 1, 1, 1, 1, 1],
             [0, 1, 0, 0, 0, 1, 1, 1, 1],
             [0, 1, 0, 1, 0, 1, 0, 0, 0],
             [0, 1, 0, 1, 0, 0, 0, 1, 0],
             [0, 1, 0, 1, 1, 1, 1, 0, 0],
             [0, 0, 0, 1, 0, 0, 0, 0, 1],
             [1, 1, 1, 1, 0, 1, 1, 1, 1],
             [1, 1, 1, 1, 0, 0, 0, 0, 3]]
```

- Expected Case:
  - BFS maze path finder
    - Locate start point via nested for loop and enumerating row and column indexes
      - Output = (0,0)
    - Only one possible path, one path needed in BFS with only adding 1 direction vector at a time, 1 new path created at a time. No paths are dropped due to no dead ends being present.
- Experimental Case:
  - Output



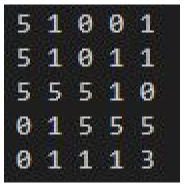
```
5 1 1 1 1 1 1 1 1
5 1 5 5 5 1 1 1 1
5 1 5 1 5 1 5 5 5
5 1 5 1 5 5 5 1 5
5 1 5 1 1 1 1 5 5
5 5 5 1 5 5 5 5 1
1 1 1 1 5 1 1 1 1
1 1 1 1 5 5 5 5 3
```

```
maze3 =     [[5, 1, 0, 0, 1],
             [0, 1, 0, 1, 1],
             [0, 0, 0, 1, 0],
             [0, 1, 0, 0, 0],
             [0, 1, 1, 1, 3]]
```

- Expected Case:
  - BFS maze path finder
    - Locate start point via nested for loop and enumerating row and column indexes
      - Output = (0,0)
    - Multiple paths will be created to reach the end due to 3 dead ends, through BFS, the dead end paths will be discarded when it is run through the conditional statement of

    -
      ```
      if 0 <= moved_row <= max_row and 0 <= moved_col <=
      max_col and maze[moved_row][moved_col] != 1:
      ```
    - As well as it will not have any adjacent nodes that match the end hence it will not be re appended back into the queue
    - The shortest/fastest path will be returned after the dead end paths are dropped and before the longer path is able to be constructed on the next iteration of the queue.

- Experimental Case:
  - Output



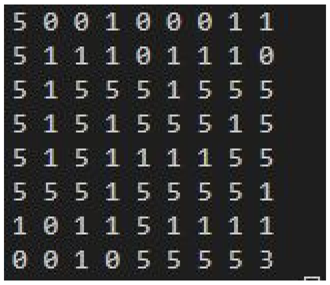
```
5 1 0 0 1
5 1 0 1 1
5 5 5 1 0
0 1 5 5 5
0 1 1 1 3
```

```
maze4 =   [[5, 0, 0, 1, 0, 0, 0, 1, 1],
           [0, 1, 1, 1, 0, 1, 1, 1, 0],
           [0, 1, 0, 0, 0, 1, 0, 0, 0],
           [0, 1, 0, 1, 0, 0, 0, 1, 0],
           [0, 1, 0, 1, 1, 1, 1, 0, 0],
           [0, 0, 0, 1, 0, 0, 0, 0, 1],
           [1, 0, 1, 1, 0, 1, 1, 1, 1],
           [0, 0, 1, 0, 0, 0, 0, 0, 3]]
```

- Expected Case:
  - BFS maze path finder
    - Locate start point via nested for loop and enumerating row and column indexes
      - Output = (0,0)
    - Multiple paths will be created to reach the end due to 3 dead ends, through BFS, the dead end paths will be discarded when it is run through the conditional statement of

    -
      ```
      if 0 <= moved_row <= max_row and 0 <= moved_col <=
      max_col and maze[moved_row][moved_col] != 1:
      ```
    - As well as it will not have any adjacent nodes that match the end hence it will not be re appended back into the queue
    - The shortest/fastest path will be returned after the dead end paths are dropped and before the longer route is able to be constructed on the next iteration of the queue.

- Experimental Case:
  - Output



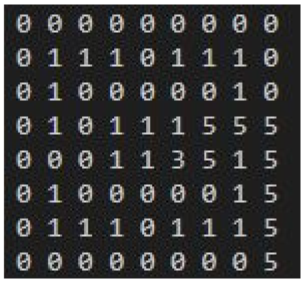
```
5 0 0 1 0 0 0 1 1
5 1 1 1 0 1 1 1 0
5 1 5 5 5 1 5 5 5
5 1 5 1 5 5 5 1 5
5 1 5 1 1 1 1 5 5
5 5 5 1 5 5 5 5 1
1 0 1 1 5 1 1 1 1
0 0 1 0 5 5 5 5 3
```

```
maze5 =         [[0, 0, 0, 0, 0, 0, 0, 0, 0],
                  [0, 1, 1, 1, 0, 1, 1, 1, 0],
                  [0, 1, 0, 0, 0, 0, 0, 1, 0],
                  [0, 1, 0, 1, 1, 1, 0, 0, 0],
                  [0, 0, 0, 1, 1, 3, 0, 1, 0],
                  [0, 1, 0, 0, 0, 0, 0, 1, 0],
                  [0, 1, 1, 1, 0, 1, 1, 1, 0],
                  [0, 0, 0, 0, 0, 0, 0, 0, 5]]
```

- Expected Case:
  - BFS maze path finder
    - Locate start point via nested for loop and enumerating row and column indexes
      - Output = (8,7) Indexes begin with 0
    - Multiple paths will be created to reach the end due to multiple dead ends, through BFS, the dead end paths will be discarded when it is run through the conditional statement of

    -
      ```
      if 0 <= moved_row <= max_row and 0 <= moved_col <=
      max_col and maze[moved_row][moved_col] != 1:
      ```
    - As well as it will not have any adjacent nodes that match the end hence it will not be re appended back into the queue
    - The shortest/fastest path will be returned after the dead end paths are dropped and before the longer route is able to be constructed on the next iteration of the queue.

- Experimental Case:
  - Output



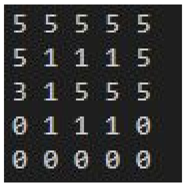
```
0 0 0 0 0 0 0 0 0
0 1 1 1 0 1 1 1 0
0 1 0 0 0 0 0 1 0
0 1 0 1 1 1 5 5 5
0 0 0 1 1 3 5 1 5
0 1 0 0 0 0 0 1 5
0 1 1 1 0 1 1 1 5
0 0 0 0 0 0 0 0 5
```

```
maze6 =     [[0, 0, 0, 0, 0],
             [0, 1, 1, 1, 0],
             [3, 1, 5, 0, 0],
             [0, 1, 1, 1, 0],
             [0, 0, 0, 0, 0]]
```

- Expected Case:
  - BFS maze path finder
    - Locate start point via nested for loop and enumerating row and column indexes
      - Output = (2,2) Indexes begin with 0
    - Multiple paths will be created to reach the end due to multiple dead ends, through BFS, the dead end paths will be discarded when it is run through the conditional statement of

    -
      ```
      if 0 <= moved_row <= max_row and 0 <= moved_col <=
      max_col and maze[moved_row][moved_col] != 1:
      ```
    - As well as it will not have any adjacent nodes that match the end hence it will not be re appended back into the queue
    - The shortest/fastest path will be returned after the dead end paths are dropped and before the longer route is able to be constructed on the next iteration of the queue.

- Experimental Case:
  - Output



```
5 5 5 5 5
5 1 1 1 5
3 1 5 5 5
0 1 1 1 0
0 0 0 0 0
```

Evaluation

The maze pathfinder written in python utilises BFS Breadth First Search algorithm to parse through the nodes within each maze. It is extremely efficient due its nature of searching in layers, meaning it utilises a FIFO queue which is able to handle new updates of new adjacent nodes to each path. Since the queue handles multiple paths at a time and may discard any invalid paths that do not reach the goal and are able to return the immediate shortest path, it is optimal for mazes that contain shallow dead ends or shorter branches/ varying paths. Hence after investigating the sample mazes, they entail shallow complexities of dead ends and overall the maze domain and range is not very long either hence BFS is the most viable option. The most outstanding development issue was translating the concept into code, with implementing a

FIFO queue and recursive loop. During the development multiple points of assignment and change were printed out to help display and understand the ordinates of where the path was traversing within the maze. Differences that would be done if I were to redo the program would be to possibly consider more conditions as BFS does have an apparent flaw of not potentially being as efficient as DFS in the case where the maze has high complexities with dead ends, meaning that there is a high occurrence of deep paths. Possibly implementing threading to run multiple maze finder algorithms at once and returning the fastest one.