A comparison of search algorithms, using an intelligent environment, on pathfinding problems; mazes.

Matthew Laws

The Question and the Background

The Question

Path finding and search algorithms have been a mainstay of problem solving within computer science, and beyond, and not least of these applications are mazes. The goal of this project, as a non-computer science background student, is to design from the ground-up a series of pathfinding algorithms in order to solve mazes, and compare the effectiveness of their implementations. The project endeavours to create an intelligent environment to establish its mazes with a node map to substantially improve the efficiency of the path finding algorithms being deployed. The project set out to build the staple breadth-first and depth-first searching algorithms, Dijkstra's and A* searches and finally the use of a genetic algorithm. The primary objective is to compare these five methods to see which is the most efficient using both perfect mazes, those with one solution, and braid mazes, those with multiple. Each algorithm will be run on a selection of mazes 25 times and measured as a product of total time to complete (efficiency) and the number of nodes used out of the total built in the environment (optimality).

Background

Solving a maze is, for a computer, a simplified version of many of the obstacles and problems that many computer systems may be faced with. From plotting a course on a map, to creating self-navigating robots, to search-and-rescue assistance, maze-solving concepts form the basis of many of these ideas and many algorithms have been developed to optimise these processes in a variety of ways. The algorithms A*, Dijkstra, and Breadth-first are often considered to be the 'best' pathfinding algorithms (Permana *et al.*, 2018) although, with their own strengths and weaknesses, they are not the sole viable or effective algorithms available.

Breadth-first searches widely through the neighbours of the starting point, continuing along a path until it reaches a new fork, searching all of the neighbours and so on. It was one of the earliest techniques created in the field and at its core is one of the most basic, being widely used due to these factors (Jamal & Teahan, 2017). However, it can be heavy on processing power and, while it may find a route through the maze, this route is not guaranteed to be the shortest possible (Jamal & Teahan, 2017). In contrast, depth-first algorithms will explore as far along a path as it is able before it will attempt another. This can be preferable in many situations involving robots, where turning around repeatedly for a breadth-first search can be impractical,

however the benefits to a depth-first algorithm are often dependent on the specific shape of the maze itself and also carries the same issue of not finding an optimised solution to the maze (Mahmud *et al*., 2012).

Dijkstra's algorithm attempts to solve mazes using the minimum node value at each juncture in an attempt to solve the mazes with a higher level of optimisation than can be seen in either breadth or depth-first, but sacrifices computational time to do so (Murata *et al.*, 2011; Permana *et al.*, 2018). The A* algorithm is based on the Dijkstra algorithm that attempts to reduce the search area at each node and produce a more highly optimised route through the maze. Murata *et al*. (2011) found A* to outperform Dijkstra on every maze tested, however AbuSalim *et al.* (2020) found that A* no longer outperformed Dijkstra when the mazes were large.

Finally, genetic algorithms attempt to mimic the natural selection patterns that we see in the wild. Commonly, they are used in search problems to find highly-optimised solutions as the reproductive aspect of the algorithm allows the search to be continued and refined through the generations until a defined end-point is reached, such as the number of generations, length of path, or arriving at the destination (Pasquier & Erdogan, 2011). The value of a genetic algorithm in a maze-solving system varies widely depending on many factors such as the maze type and size, the 'intelligence' of the agents undergoing 'evolution', and the desired outcome of the algorithm. The process is also computation-heavy, often requiring more time and/or more processing power than the other algorithms mentioned (Pasquier & Erdogan, 2011; Kim & Kim, 2019).

Designing the Environment,

In order to be able to assess the efficiencies of the search algorithms, first the environment needed to be made. First, the environment was designed as a program which interprets mazes from an image. The simple design, which requires the entrance to be on x.min and the exit on x.max, is that walls have a value of 0 and paths have a value of 1; or more specifically >0.

The environment we have built by this point is enough to start working on algorithms but lacks any real efficiency, especially for larger mazes, and so the idea to build nodes in the environment was conceptualised. The premise of these nodes is built upon the idea that the pathfinding algorithm being deployed does not need to make decisions on every horizontal or vertical path but rather only on the exact pixel where continuing in a straight line isn't an, or the only, option. The ethos here being that all pixels that are not nodes can be completely disregarded having a significant impact on saving the amount of computational power required to find a solution. For context, on the "small" 15x15 map without nodes there are 100 pixels under consideration in the solution and with the node structure there are only 37 (**Figure 1)** and if we directly scale this up to a 4001x4001 map, which is ~71,147 times bigger, that gives us approximately 7,114,700 tiles to consider without nodes and a *more manageable* 2,632,427 pixels to consider with nodes.

The first iteration of the node map builders, which was deemed inefficient, added nodes only at junctions and not at corners. The idea behind this was to increase efficiency by creating the minimum number of nodes necessary and ignoring corners and dead ends. Furthermore, it would also record the path between the nodes however by recording the path between the nodes it was effectively building a flood fill between each node, making the nodes relatively redundant and the program sluggish. The second iteration, and the one used in the final program, added nodes at corners and, whilst it added a small amount of extra computational overhead to building the node map, it substantially increased the efficiency of the search algorithms in finding their solutions. This final iteration also stored the nodes as a simple point in memory for the search algorithms to jump to if necessary rather than recording a path from node to node.

For each node the algorithm examines the surrounding nodes in each direction. For simplicity these are named North, South, East and West. The pixel colour of the neighbour is used to determine if the adjacent pixel is a path or a wall, as Black is 0, which is logically the same as False (also 0) each cardinal direction was set to a bool indicating if a given node had a neighbour in that direction:

```
if (data[y][x] > 0):
  isEast = (x < len(data[0]) - 1) and data[y][x + 1] > 0
  isWest = (x > 0) and data[y][x - 1] > 0
  isNorth = (y > 0) and data[y - 1][x] > 0
  isSouth = (y < len(data) - 1) and data[y + 1][x] > 0

  if self._isNode(isEast, isWest, isNorth, isSouth):
     node = Node(y, x)
     #Holding in memory a node if it is connected to another node on it's
west-most side, else don't.
     if isWest:
        node.addNeighbour(leftMost, 1)
        leftMost.addNeighbour(node, 3)
     leftMost = node if isEast else None
     #Same as for isWest, but for southern nodes.
     if isNorth:
        node.addNeighbour(topMost[x], 0)
        topMost[x].addNeighbour(node, 2)
     topMost[x] = node if isSouth else None

     graph.append(node)
```

Once it is known if a node has a neighbour in a given direction, the algorithm searches for its Northern and Western neighbours, if they are present, it also sets itself as discoverable later for

nodes to the South and East. By only searching for two of the four neighbours it means that connecting each node to its neighbour isn't creating duplicate work:

```
def _findNeighbours(self, y: int, x: int, data: [[bool]]):
  neighbours = []
  # West neighbour (x - 1)
  if ((x > 0) and data[y][x - 1]):
     neighbours.append(Coord(y, x-1))
  # North neighbour (y - 1)
  if ((y > 0) and data[y - 1][x]):
```

Methods

The methods aren't designed with any specificity to the question, rather they were made from the ground up to be as efficient as I was capable of doing. The mazes used in the solution were taken from the repository from Computerphile's series on mazes featuring Assistant Prof. Mike Pound of the University of Nottingham; here is a link to the repository.

Breadth-First and Depth-First

A breadth-first search starts on our node map at the root of the tree and will expand and search the neighbours of all its children before descending down a level into their children. This method works through the maze using a combination of the SearchItem class along with the explorationQ and exploredQ found within the BreadthFirst class. Once a node is explored it is removed from this list and added to the exploredQ. Because this method relies on expanding the nodes in the order they are added from the tree, explorationQ will always explore the next node at position [0] in the explorationQ list:

```
while True:
     node = self._explorationQ[0].node
     path = self._explorationQ[0].path
```

Depth-first, by contrast, will always expand the first node in the queue and continue expanding it down to it's terminal point; either the end of the maze or a deadend. Depth-first works in the same way but will always explore the item added to the explorationQ most recently, at position [-1], to force the algorithm to explore all the way to a terminal point of that branch of the node graph:

```
while True:
      node = self._explorationQ[-1].node
      path = self._explorationQ[-1].path
```

Both of these methods' execute functions rely on the final node being in the list of nodes and so the final node is initialised as self._finalNode = nodes[-1] as we can always safely assume the final node will be the last in the list.

Dijkstra's and A*

Dijkstra's search algorithm is primarily used to find the shortest paths between nodes on our graph. This is achieved by adding a value between two nodes that can be considered it's complexity to use. For our purposes, the value that is added between the nodes is the distance between them; i.e. the number of pixels travelled from one node to another. The program determines this by using a length variable which is updated throughout the search using currentLength, the total length of this route currently, added to the distance from the currentNode to its neighbour:

```
while (not self._queue.isFinished(current)):
  # Update the neighbours
  for i in range(4):
     if current.node().branches[i] != None:
        length = current.length() + current.node().distanceToNeighbour(i)
        self.queueItemPathLength(current.node().branches[i], current, length)
  # Go again for the next node
  current.setVisited()
  current = self._queue.getNext()
```

Typically this is used in satellite navigation and explains why typically motorways are prioritised as they provide the least complexity to traverse. The algorithm is designed to always work it's way to the end of the maze using the lowest cumulative Dijkstra's value.

A* search algorithm works using the same weights established in Dijksra's model whilst adding an extra heuristic. This added heuristic is the distance from the next given node to the end of the maze as a value which is then added to the Dijkstra's value. The program determines the distance from the end, the aStarFactor, at the target node's location in the following way:

```
def setAStarFactor(self):
  for queueItem in self._queueItems:
     currentNode = queueItem.node()
     aStarFactor = abs(self._finish.node().xcoord + currentNode.xcoord) +
abs(self._finish.node().ycoord + currentNode.ycoord)
     queueItem.setAStarFactor(aStarFactor)
```

The value is not cumulatively calculated and is instead specific to the next node which means this value always gets lower as the algorithm works towards the end of the maze; generally prioritizing a route that always works towards the destination.

Genetic Algorithm

The metaheuristic approach of genetic algorithms is a paradigm that provides functions to mimic the process of natural selection and evolution to solve a problem to find a solution considered *good enough*. Our program assigns a fixed number of a genomes to a generation, each genome consisting of a random selection of numbers, their genes, up to 3 (0-indexed) which reflect NESW movement; 0 being North, 1 East, 2 South and 3 West:

```
for i in range(populationSize):
   genome = random.randint(4, size=genomeLength).tolist()
   self._population[i] = Genome(genome)
```

The length of a genome is equal to the maximum number of moves it could possibly take to solve the maze. Generation 0 is always a random sequence however their children and subsequent generations are created using a crossover function and a mutation function, which has a fixed chance to alter a random gene in a genome at a random position. This goes a long way to replicating what is seen in nature and only leaves creating the fitness of each genome. The fitness of a genome is considered it's distance to the end, ergo the lower fitness score the more fit the genome is. Importantly, a genome is punished under certain conditions and increases its fitness score, lowering its overall fitness by a degree of +=1. These conditions are if a move causes it to hit a wall or if the move made by the genome is not a novel move; i.e. it has visited the current node previously:

```
for genome in self._population:
   currentNode: Node = nodes[0]
   for direction in genome.getGenome():
       nextNode = currentNode.branches[direction]
       if (nextNode == None):
           genome.punish()
       else:
           if (visitedNodes.__contains__(nextNode)):
               genome.punish()
           else:
               visitedNodes.append(nextNode)
           currentNode = nextNode
```

An example of a genome looks can be seen in **Figure 2**, where you can see its fitness, it's genome array, current status of completion and finally its total acquired penalty.

Creating new generations is carried out in a few ways. First of all, we have elitism. Elitism will keep a fixed number of the top few genomes and will not alter them in any way; this makes sure

our current best solution is not destroyed. In a similar way, a fixed number of our few worst genomes are destroyed due to a lack of fitness. Those that are left are subject to both crossover and mutation and thus the next generation is created; this will loop until a solution is found:

```
def createNextGen(self):
   #sortedPopulation to allow for selected first and last with highest and lowest
fitness respectively
   sortedPopulation = sorted(self._population, key=lambda g: g.fitness())
   breedingPopulation = sortedPopulation[0:(-1*( self._fitChildrenNum))]
   newPopulation = [None]*self._popSize

   for i in range(self._fitChildrenNum):
       newPopulation[i] = Genome(sortedPopulation[i].getGenome())
   for i in range( self._fitChildrenNum, self._popSize):
       newPopulation[i] = self.crossBreed(breedingPopulation)

   self._population = newPopulation
   self.mutate()
```

```
def crossBreed(self, breedingPopulation: [Genome]):
   genome1 = breedingPopulation[random.randint(self._popSize -
self._fitChildrenNum)].getGenome()
   genome2 = breedingPopulation[random.randint(self._popSize -
self._fitChildrenNum)].getGenome()

   split = random.randint(len(genome1))

   part1 = genome1[:split]
   part2 = genome2[split:]
   addedArray = part1 + part2
   return Genome(addedArray)

def mutate(self):
   for x in range(self._mutationNum, self._popSize):
       genome = self._population[x]
       for i in range(self._genomeLength):
           mutationChance = random.randint(100)
           if (mutationChance > 50):
               genome.mutateGene(i, random.randint(4))
```

Running, Results and Analysis

Running the Program

In order to run the experiments the program is designed to provide a detailed breakdown in the console displaying the time to build the node map and how many nodes there are, along with how many of these nodes were used, and the time it took to find the solution; the time for the solution is independent of the time to build the node graph (**Figure 3**). In order to ensure a legitimate solution was found, and to compare the way the methods find their solutions, once one is found it is printed and saved as solution.png in the maps folder, an example can be seen in **Figure 4**. To aid in ensuring the program is running as intended, both for the solution and the environment, the nodes are coloured in a light grey colour and the path taken between them from the start to the destination is a darker grey.

Please refer to README.pdf for instructions on running the program.

Results

Averages of 25 runs were calculated per algorithm per maze. The genetic algorithm was only used for the smallMaze as it is not efficient enough for larger mazes. The mazes selected for analysis were smallMaze, normalMaze, braid200 and combo400. Tiny maze did offer any significant insight as all algorithms solved it in a similarly instantaneous speed. Larger mazes such as braid2k and perfect4k were not used because the time taken for any method other than depth-first were too exhausting; the 25 results for braid200 using A* took 7.5 hours alone. Small and Normal maze only have one solution, briad200 has multiple solutions and combo400 has multiple solutions with some replaced as deadends for a more complex required solution.

Maze	DepthFirst	BreadthFirst	Dijkstra	A*	Genetic
Small (Time(s))	0.0*	0.00049	0.0009	0.0024	1.5
Nodes Used	23	17	17	16	24

Table 1: smallMaze solution time (seconds) and number of nodes used out of a maximum 37 nodes. *time faster than python can compute, , typically < 0.0004s.

Maze	DepthFirst	BreadthFirst	Dijkstra	A*
Normal (Time(s))	0.0*	0.00049	0.0009	0.0024
Nodes Used	119	119	118	119

Table 2: normalMaze solution time (seconds) and the number of nodes used out of a maximum 325 nodes. *time faster than python can compute, typically < 0.0004s.

Maze	DepthFirst	BreadthFirst	Dijkstra	A*
braid200 (Time(s))	0.14	82	33	63
Nodes Used	1082	193	196	193

Table 3: braid200 solution time (seconds) and the number of nodes used out of a maximum 6309.

Maze	DepthFirst	BreadthFirst	Dijkstra	A*
combo400 (Time(s))	6.41	DNF*	588	1090
Nodes Used	1478	N/A	323	319

Table 4: combo400 solution time (seconds) and the number of nodes used out of a maximum 27,296. *Breadth-first took >6 hours and was not completed.

Analysis and Discussion

The results show us depth-first was the quickest at solving all the mazes by a substantial margin. However, it is interesting to look at the amount of nodes it used to find its solutions, in comparison to other methods, to reveal its poor optimization in the found solution. Dijkstra's and A* have the added heuristics of finding *optimal* solutions. This proved to add considerable runtime yet, however, it had the desired outcome in terms of optimization; both Dijkstra and A* were more optimal than depth-first in every case.

For comparison of Dijkstra and A*, adding the heuristic of distance to the end for A* proved to increase how optimal its solutions were over Dijkstra's, however it took longer to complete. The

results show that it was almost always finding more optimal solutions, those using fewer nodes, than the Dijkstra solutions. It is then safe to surmise that A* and Dijkstra's are much better at path-finding optimisation tasks, such as SATNAV, whereas depth-first may often prove superior at fast search based tasks. Additionally, breadth-first was seen to be efficient but only up to a point. Once the mazes got much larger, such as in the combo400 maze, its intrinsic solving method sees it run through the majority, if not all, of the 27,296 nodes. This method would likely show better results if the destination was not at the x.max location in our maze but at a random point within the maze, however when given this specific type of maze solving problem, it performs poorly.

The outcome for genetic algorithms was, although entirely functional, disappointing. Their main difficulty is the efficiency of the fitness function. Within the purpose of maze solving, the distance to the end as a score of fitness is the standard usage, however this does cause problems. If a genome were to reach a deadend near the end of the maze, requiring a very large loopback further up the maze, this decision will never be made and the genes will likely be stuck here indefinitely - unless a perfect mutation at the right gene should occur. Optimising this fitness score led to the greatest challenge and ultimately was the biggest limitation of this paradigm. Further significant limitations are found in designing the parameters of the algorithm, such as the number of genomes in a generation, the mutation chance or the length of the genome. There are no general fixed rates or numbers for these values and tweaking them to find the optimum algorithm for a solution is necessary - however this could have a detrimental effect for the algorithm on other maze types. Finally, their randomness proved to be one of the most detrimental factors for any real determinable outcome. To apply context, five consecutive executions of the genetic algorithm on smallMaze solved it in 0, 27, 58, 154, and 454 generations. Genetic algorithms are exemplary at optimization tasks, and to an extent, search tasks. Problems such as the "knapsack problem" are where it is shown off at its best. When the knapsack problem has 77 items to choose from it could take ~5 billion years to solve without a genetic algorithm and with one, only ~2 seconds.

Conclusion and Further Work

The project was a success in what it set out to do: design an intelligent environment and compare pathfinding algorithms designed from the ground up and, in addition to this, all the results reflect the findings in the *Background* section of the report. Depth-first searches proved to be vastly superior to other implemented methods on all mazes in terms of its completion time but was significantly worse in it's optimization of the solution, always using far greater numbers of nodes. Breadth-first performed similarly but, as to be expected with the way it works, proved to take significantly longer as the mazes grew in size. Dijkstra's and A* search algorithms performed relatively similarly on smaller mazes however, as the mazes grew larger, these methods took much longer than depth-first. The added heuristic of A* proved to improve efficiency over Dijkstra's resulting in a more optimal solution in terms of nodes used, but did take longer to calculate.

Upon reflection, a significant limitation to this project is the small sample size of mazes used. Variations of same sized mazes to improve sample size would be principal in achieving empirical evidence of efficiency between the algorithms. Furthermore, whilst the algorithms weren't designed with these specific mazes in mind, their efficiencies were tweaked based on the performance within them. Therefore, it does not give us an accurate representation of an algorithm's general efficacy on maze solving but rather its maze solving efficiency for these maps specifically. Again, the solution is either a greater sample size of mazes, or the use of a random maze generator. Additionally, redesigning the destination position for the mazes could prove to yield more robust and significant differences between the methods. Having a random start point in the maze and a random destination, one that isn't on x.max, could really help enhance the analysis.

I would also implement a live-drawing/animated solver. This would enable us to see exactly what the algorithm, such as the genetic algorithm for example, is doing as it attempts to solve the maze and, specifically for genetic algorithms, could help in optimising the algorithm to work on larger mazes. It would also provide interesting illustrations to the dynamic differences between breadth and depth-first and how the added heuristic of A* improve on Dijkstra's paradigm. Additionally, I would also implement the use of colour to the solution.png, this turned out to be quite a significant challenge.

Figures

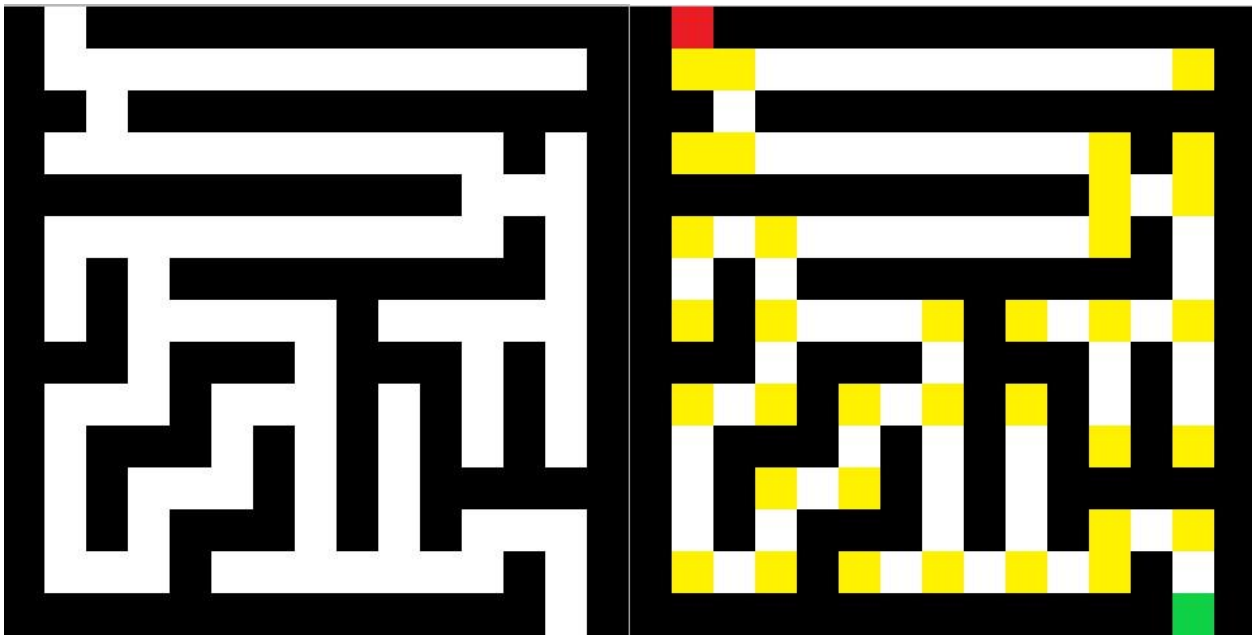

Figure 1: A comparison of the environment without nodes (L) and with nodes (R)

```
_fitness = {int} 41
_genomeArray = {list: 37} [2, 2, 0, 2, 3, 1, 3, 1, 1, 0, 2, 1, 1, 0, 2, 3, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 2, 2, 2, 3, 1, 0, 2, 2, 2, 0, 2]
_isComplete = {bool} False
_penalty = {int} 16
```

Figure 2: An example genome from the Genetic Algorithm at a random point in an ongoing solution attempt.

```
----------------------------------------
Building node graph for chosen Maze
Time taken equals 0.032263994216918945 seconds
There are 325 nodes in the Maze
 ----------------------------------------
Running the solving method
Time taken equals 0.0004961490631103516 seconds to solve using Depth First search algorithm
Using 119 number of nodes in the solution
 ----------------------------------------
```

Figure 3: An example successful console readout for the purpose of running the experiment.

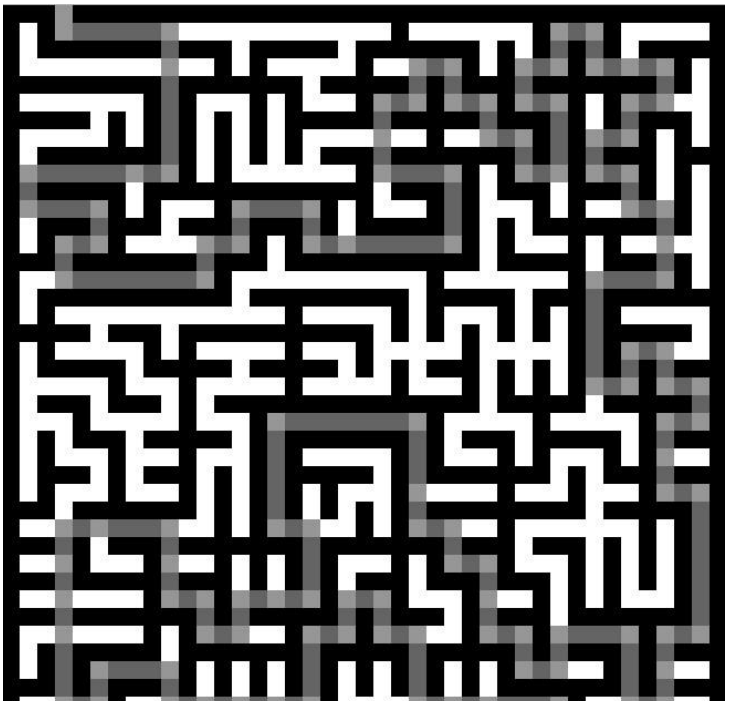

Figure 4: An example solution found - this used depth-first on normalMaze.

Acknowledgements

All of the coding done for this project was designed from the ground up by myself however the mazes were taken from Mike Pound's repository and the way the program reads mazes is based off of the work done by Mike in this project. A big thanks to Mike for making the work he did with Computerphile open source and unrestricted; it was a great help.

Bibliography

AbuSalim S. W. G., Ibrahim R., Saringat M. Z., Jamel S. and Wahab J. A. (2020), "Comparative Analysis between Dijkstra and Bellman-Ford Algorithms in Shortest Path Optimization", *IOP Conference Series: Materials Science and Engineering*, 917, pp. 1-11.

Jamal A. A. & Teahan W. (2017). "Alpha Multipliers Breadth-First Search Technique for Resource Discovery in Unstructured Peer-to-Peer Networks", *International Journal on Advanced Science, Engineering and Information Technology*, 7, pp.1403-1412, doi: 10.18517/ijaseit.7.4.1451.

Kim J., Kim S.K. (2019). "Genetic Algorithms for Solving Shortest Path Problem in Maze-Type Network with Precedence Constraints", *Wireless Pers Commun* 105, pp. 427–442 . https://doi.org/10.1007/s11277-018-5740-3

Mahmud M. S., Sarker U., Islam M. M. and Sarwar H. (2012), "A Greedy Approach in Path Selection for DFS Based Maze-map Discovery Algorithm for an autonomous robot," *2012 15th International Conference on Computer and Information Technology (ICCIT)*, pp. 546-550, doi: 10.1109/ICCITechn.2012.6509798.

Murata Y. and Mitani Y. (2011), "A study of shortest path algorithms in maze images," *SICE Annual Conference 2011*, pp. 32-33.

Pasquier, T., & Erdogan, J. (2011). Genetic Algorithm Optimization in Maze Solving Problem.

Permana S.D.H., Bintoro K. B. Y., Arifitama B. and Syahputra A. (2018), "Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game", *International Journal of Information System and Technology*, 1(2), pp.1-8.