|  |
| --- |
| COS30019 |
| Assignment 1 Report – Benedict Zeng 6450555 |
| Option B: Robot Navigation Problem |

|  |
| --- |
| Benedict Zeng  4/26/2019 |

Contents

[Introduction 2](#_Toc7186624)

[Search Algorithms 3](#_Toc7186625)

[Uninformed Search 3](#_Toc7186626)

[Depth-First Search 3](#_Toc7186627)

[Breadth-First Search 4](#_Toc7186628)

[Informed 5](#_Toc7186629)

[Greedy Best First Search 5](#_Toc7186630)

[A\* Search 6](#_Toc7186631)

[Implementation 7](#_Toc7186632)

[Uninformed 7](#_Toc7186633)

[Depth-First Search 7](#_Toc7186634)

[Breadth-First Search 8](#_Toc7186635)

[Informed 8](#_Toc7186636)

[Greedy Best First Search 8](#_Toc7186637)

[A\* 9](#_Toc7186638)

[Custom 2 9](#_Toc7186639)

[Features/Bugs/Missing 10](#_Toc7186640)

[Research 11](#_Toc7186641)

[Visualiser 11](#_Toc7186642)

[Diagonal Movement and Diagonal Distance 12](#_Toc7186643)

[Conclusion 14](#_Toc7186644)

[References 15](#_Toc7186645)

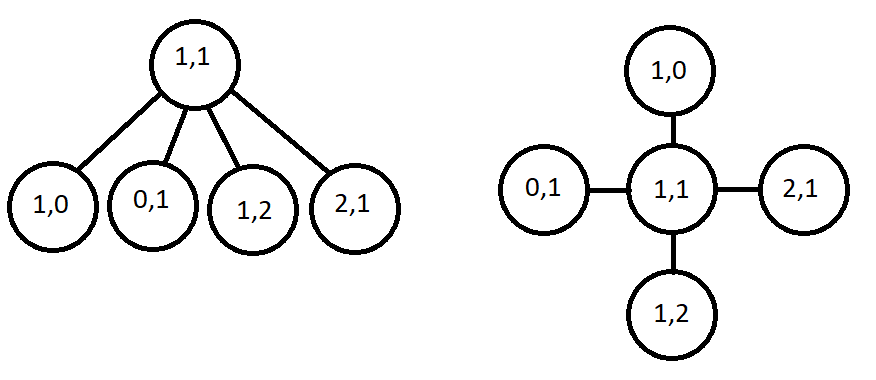
# Introduction

The Robot Navigation Problem is a common maze problem that tests different search algorithms on search trees. A start point and end point is usually provided, and the search algorithms are tested on how effectively they find the solution compared to each other.

For this assignment, a 2D array was provided along with a start point, goal point and various obstacles. The task was to reassemble this data into a world that can be traversed by search algorithms, and then create four separate search algorithms, as well as an optional two custom search algorithms. The required algorithms were Depth-First Search, Breadth-First Search, Greedy Best First Search and A\* Search. The output required me to return the filename, the method used, the number of nodes searched and the path taken.

To solve this problem, I had to first understand graph and tree concepts. The fundamental idea is that a node can have multiple child nodes, and expanding all possible nodes will form a tree graph that would represent the entire data set. For example, the very top parent node could be (1,1) to represent the starting position, and if it had four child nodes around it they would be “up” (1,0) “left” (0,1) “down” (1,2) and “right” (2,1) . Each of those in turn might have their own child nodes around them. Branching the entire tree out would yield the full data set, as well as (hopefully) the goal state to reach.

*Fig 1: A single-step tree showing the starting point of 1,1 and its branching options, as well as the same node structure represented in grid-form.*



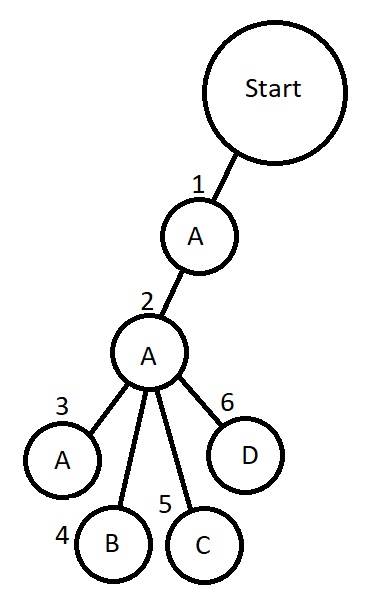
# Search Algorithms

## Uninformed Search

Uninformed search algorithms do not inherently know anything about the data set other than the start and goal positions. It will step through every single possible node until it reaches the goal position. Typically, a program created for this search method would also keep a memory of the nodes it had previously searched, in order to avoid infinite loops.

### Depth-First Search

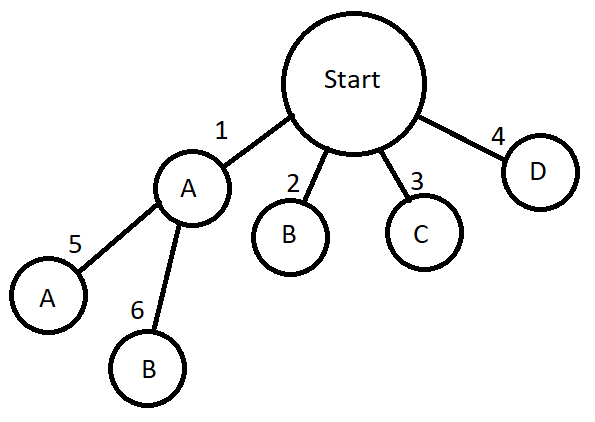
Depth-First Search (DFS) is designed to travel in a single direction all the way to the end, before backtracking and travelling in a single direction on a separate branch. Given a tree diagram of 3 steps of nodes, each of which containing the letters A B C and D, its search path would be ‘A, AA, AAA, AAB, AAC, AAD, ABA, ABB, ABC, ABD…’ and would continue this pattern until it reached the end. This is visually represented by a tree diagram where one side of the tree is searched first, before moving to the side.



*Fig 2: A tree diagram showing the search pattern for DFS. The numbers indicate the order in which the DFS method would expand child nodes.*

### Breadth-First Search

Breadth-First Search (BFS) is very similar to DFS, however instead of searching the first child node from each node, it will search all child nodes. This would be visually represented by a tree expanding sideways before expanding down. Given the same naming structure from DFS, the search path would be ‘A, B, C, D, AA, AB…’ and so on until the end is reached.



*Fig 3: A tree diagram showing the search pattern for BFS. The numbers indicate the order in which the BFS method would search through nodes.*

Comparing DFS to BFS, it can be easily deduced that these two are actually the same search patterns, albeit with the logic swapped on when to search another existing node and when to expand a node. DFS expands first, whereas BFS expands second.

## Informed

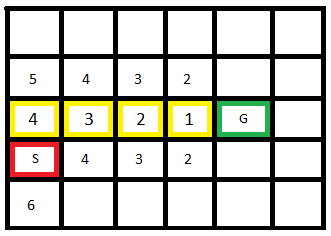
Informed search algorithms are the more commonly used search algorithms due to its more useful nature. This is because the search algorithms actually take heuristic data into account when searching through nodes. The types of data required depend on the data set’s search requirements. In this assignment specifically, a 2D maze structure would at a bare minimum require basic distance estimates between a node and the goal, allowing the search algorithm to make ‘smarter’ decisions.

This estimate is typically found using one of three methods: Manhattan, Diagonal and Euclidean. Manhattan is designed specifically for four directions of movement, and provides the heuristic values based on horizontal and vertical distances being added together. Diagonal is designed for eight directions and provides a heuristic value based only on the highest value between the horizontal and vertical distances. Euclidean is designed for every angle of movement on a 2D plane, providing a heuristic value based on Pythagoras’ theorem using the horizontal and vertical distances.

The Manhattan method was the best approach to use for this assignment, as the movements were limited to only horizontal and vertical.

### Greedy Best First Search

Greedy Best First Search (GBFS) is the most rudimentary form of Informed Search, using only the estimated distance between the node and the goal to decide which node to search next. This makes it very effective at finding straight line paths between the start and goal nodes. Given the starting position of (0,3) and goal position of (4,2), the GBFS method would move upwards, then all the way to the left, each node finding the new heuristic value driving it towards the goal.

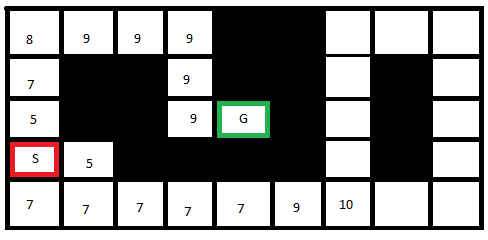


*Fig 4: A grid showing the starting ‘S’ position of (0,3) and the goal ‘G’ position of (4,2). Each number is the heuristic distance found using the Manhattan method to determine which node to expand next. The GBFS method moves up first, then left towards the goal, denoted by the yellow square outlines.*

Unfortunately, obstacles in its path would increase the steps taken, and GBFS cannot deal with this well because it does not take the amount of steps it has already taken into account.

### A\* Search

A\* Search (A\*) is the more advanced counterpart to GBFS. The only difference between the two is that A\* also takes its already taken steps into account, adding it to the heuristic value. However, this simple addition can actually be the difference between taking an efficient route and an inefficient route. By adding the amount of steps taken to the existing Manhattan distance value, A\* can essentially understand when a path is no longer effective anymore, and will select a new path that has a better heuristic value.

  
*Fig 5: A grid showing a start ‘S’ position of (0,3), a goal ‘G’ position of (4,2) and a number of obstacles in between. The numbers show the heuristics calculated from combining both the Manhattan distance with the steps taken to reach that node.*

As seen in Fig 5, the bottom row shows a number of ‘7’ values, and as a result A\* will traverse this path first, believing it to be the shortest path. However, as it reaches beyond the goal position, the heuristic value quickly climbs, and A\* is able to change paths and select the top node instead. If this same maze was searched using the GBFS method, it would continue searching the bottom row and then the entire right hand loop, before returning to the other path, resulting in extra nodes being searched. This effectively shows when A\* is more efficient than GBFS.

# Implementation

The first step in my implementation was to parse the given text file providing the data set. I generate the ‘world’ data using the grid size, start and goal positions and obstacle positions. This is expressed using a multi-dimension array of strings, where the start and goal positions are denoted as ‘S’ and ‘G’. Obstacles are denoted using a custom ascii character that is just a complete block, and an empty space is denoted as a space character ‘ ‘.

Every search method is called using the function ‘BeginSearch’. This essentially acted as the main method for them, neatly containing all the function calls required for the search method to work. ‘BeginSearch’ is given the world array so that it can search through it. It creates an ‘outcome’ list that will be returned as a string at the end of the search method.

I also find the start and goal positions, just to check that they actually exist. If either of those two cannot be found, then no solution can be found, and the program returns the appropriate message.

## Uninformed

### Depth-First Search

For my DFS, while my notVisited list still contained items and the goal hasn’t been reached, I loop through these functions:

1. Increment node count
2. Visit the first node in the notVisited list
   1. currentPosition becomes the first node
   2. The first node in notVisited is popped to the visited list
3. Find valid adjacent positions around the currentPosition
   1. Clear the adjacents list
   2. Check all four directions to ensure they are valid positions
   3. Check each direction for the value within its position
      1. If the value is ‘G’, then the goal has been found
   4. Add valid directions to the adjacents list
4. Add adjacents to the notVisited list
   1. Check that the position hasn’t already been visited before
   2. Check that the position isn’t already added to the notVisited list to avoid infinite loops

It is important to note that for DFS, the values are added to the beginning of the notVisited list. Therefore, the order of directions to be checked must be in reverse of the assignment’s required order to counter this.

### Breadth-First Search

My BFS implementation is very similar to my DFS, with the only difference being that it places valid adjacents to the back of the notVisited list instead of the front.

1. Increment node count
2. Visit the first node in the notVisited list
   1. currentPosition becomes the first node
   2. The first node in notVisited is popped to the visited list
3. Find valid adjacents around the currentPosition
   1. Clear adjacents list
   2. Check all four directions to ensure they are valid positions
   3. Check each direction for the value within its position
      1. If the value is ‘G’, then the goal has been reached
   4. Add all valid directions to the adjacents list
4. Add adjacents to the notVisited list
   1. Check that the adjacent nodes don’t already exist in the visited list
   2. Check that the position doesn’t already exist in the notVisited list to avoid infinite loops

## Informed

### Greedy Best First Search

For my GBFS, the addition of the Manhattan distance and the lack of ordered lists is what differentiate this from the previous Uninformed methods.

1. Increment the node count
2. Visit the node that currently has the lowest estimated distance from the goal
   1. currentPosition becomes the node from the openSet with the lowest distance
   2. Remove said node from openSet
3. Find valid neighbours around the currentPosition
   1. Clear the neighbours list
   2. Check all four directions to ensure they are valid positions
   3. Check each direction for the value currently in that node’s position
   4. If they are valid, add the direction to the neighbours list
4. Generate the new estimated distance value of each neighbour node
   1. Calculate the Manhattan distance from the neighbour node to each goal
      1. Only take the shortest distance
   2. Check that the neighbour node doesn’t already exist in either openSet or closedSet
   3. Add valid nodes to the openSet

### A\*

A\* is very similar to GBFS, except that it also incorporates the steps already taken in its heuristic value calculations.

1. Increment the node count
2. Visit the node that currently has the lowest heuristic value
   1. currentPosition becomes the current node from the openSet
   2. Said node is then removed from the openSet
3. Find valid neighbours around the currentPosition
   1. Clear the neighbours list
   2. Check all four directions to ensure that it is a valid position
   3. Check each direction for the value currently in that node’s position
   4. If they are valid, add the direction to the neighbours list
4. Generate the new heuristic value of each neighbour
   1. Calculate the Manhattan Distance from the neighbour to the goal
   2. Add value to the steps already taken to reach the neighbour
   3. Check that the neighbour doesn’t already exist in either the openSet or the closedSet with a lower heuristic value due to lesser steps
   4. Add valid nodes to the openSet

Checking the existing openSet for an identical position with a lower heuristic value ensures that the shortest path is retained, even if a loop ends up in the same position later.

### Custom 2

Custom 2 is essentially A\* but with diagonal movement added. Throughout my research for A\*, it became apparent to me that most implementations on a 2D plane showing A\* actually incorporates diagonal movement, so I thought it would be a good experiment to take, and to see just how big of a difference it was. In essence, Custom 2 is actually an identical implementation of A\*, with the only difference being that it also checks diagonal movement and instead of using Manhattan Distance, it uses Diagonal Distance to calculate its heuristics. This will be discussed in the Research section.

# Features/Bugs/Missing

I am currently missing the Custom1 search algorithm. It is only partially coded out, and is not functional. It was to be a Random Mouse algorithm, for the sole purpose of exploring the ineffectiveness of randomised search. This was an interesting idea to me, given that in most mazes, an agent actually wouldn’t know where the goal position is. This would effectively make the Informed search algorithms useless, since they rely on that kind of data to create heuristics properly. Random Mouse would have been a very entertaining Uninformed search algorithm to watch for the sake of drawing parallels to how real living creatures might try to navigate a maze.

I am not aware of any existing bugs for the four required search algorithms, or the custom algorithm.

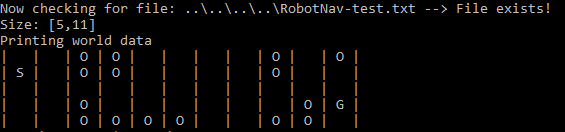
Throughout the development of the algorithms, I implemented various console outputs that allowed me to quickly and easily debug problems with calculations. These were all removed from the code eventually, however a part of me believes they should have remained, simply for the sake of clarity. This is because it became clear to me that sometimes values aren’t easily disseminated from just viewing the path that the algorithm decides to take.

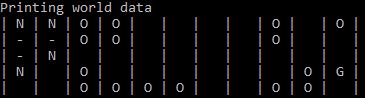
I did find that I had created a startPosition variable that ended up not being all that useful. It just ended up eating more CPU cycles without any real benefit. As a result, I have commented/removed it from the code. In theory, this would have been retained so that it could be referenced later on in development should I want to expand on the functionality, but given the limited amount of time left, it was not feasible.

# Research

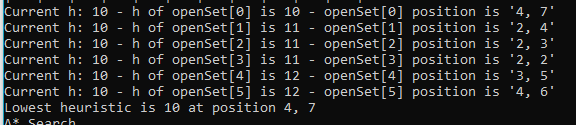
## Visualiser

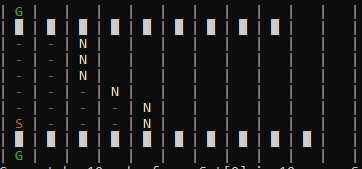
The visualiser for the world was the single most effective way to debug problems with my search algorithms. Throughout the debugging process, seeing error codes about values being out of index or seeing a search algorithm return a value and path was nowhere near an effective solution to finding out what aspect of your code was not working.

In the beginning, I had all the outputs written to console, which would give values and world positioning in its entirety.   
*Fig 6: The initial check to see if a file exists at all, and then a printout of the world space. The ‘S’ denotes the Start position, the ‘G’ denotes the Goal position, and the ‘O’ denotes an Obstacle.*

Then, as the search algorithm begins navigating the world, the values were updated to show this. *  
Fig 7: The search algorithm has navigated through 3 nodes, represented by a ‘-‘. These nodes have also revealed 4 child nodes, represented by an ‘N’.*

As each node and child node was visited, data for heuristics were printed out to show how the calculations were doing.  
*Fig 8: Current h was the currently lowest found heuristic, h of openSet[i] was the heuristic of the openset’s node, and the position was where in the world grid it occupied.*



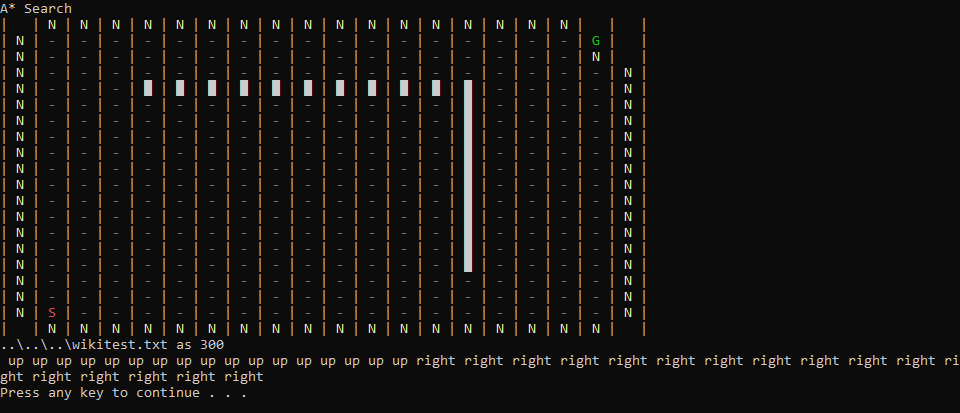
But then, towards the end of development, all of these outputs were culled out to only show the world grid updating. I also implemented some new colours and changed the ‘O’ Obstacles to a custom ascii block character.  
  
*Fig 9: the ‘S’ was now red, the ‘G’was green, the ‘N’ unsearched nodes were yellow and the ‘-‘ searched nodes were grey. The cusom ascii block characters were a much clearer representation of an obstacle.*

In addition to this visual update, I also added a tiny delay in every single world draw call, so that it could be clearly seen by the human eye before being redrawn.

## Diagonal Movement and Diagonal Distance

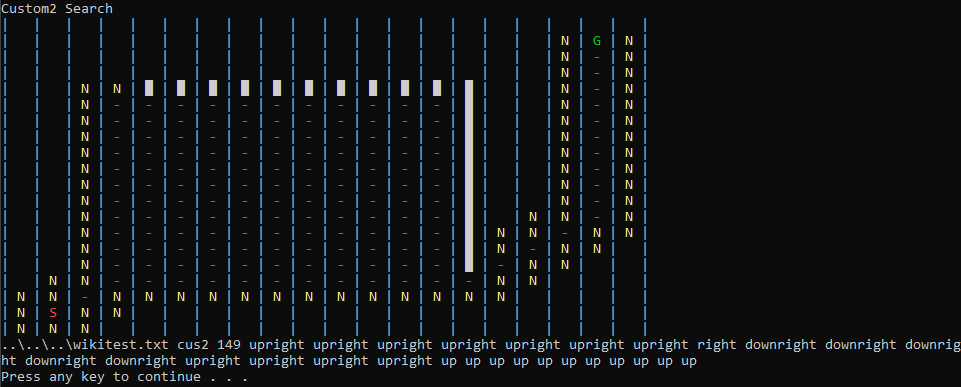
The second aspect of research came in the form of implementing my Custom2 search algorithm. This was essentially an identical version of the A\* search algorithm, except that it had eight directions instead of four to choose from. This would inevitably lead to lesser nodes being viewed. In addition to this, the Manhattan Distance was changed to a Diagonal Distance, which is theoretically what it was designed for. However, the results were somewhat peculiar.

For reference, I recreated the Wikipedia entry’s version of their A\* world and ran the assignment’s default A\* search on it.

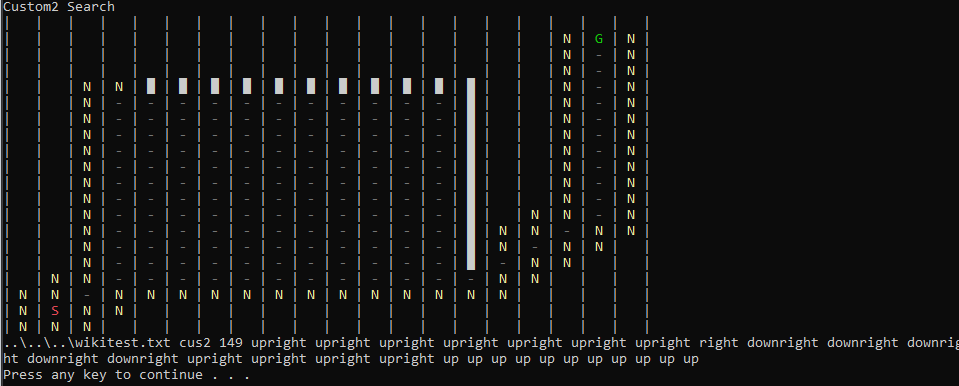


This method searched 300 nodes and almost visited the entire world.

By contrast, this is the same map searched by my Custom2 method, which can move in 8 directions.



This was very impressive at only 149 nodes being searched. In high spirits, I changed the Manhattan Distance to Diagonal Distance, hoping it would give an even smaller search node.



Unfortunately, it seems that using the Diagonal Distance heuristic calculation instead of the Manhattan Distance was actually worse off. This was particularly strange to me, as it had been proposed that Diagonal Distance was better suited to diagonal movements. In future, I would like to revisit this and find out why, and perhaps even try to find a more optimal way to measure heuristics. Currently, my Custom 2 search algorithm uses the Diagonal Distance heuristic measurement.

# Conclusion

Conceptually, A\* is clearly the better search method out of all the given required methods. While it is true that certain grid layouts can be solved faster using other methods, knowing which method would be best suited would require prior knowledge, which can in itself be a big computational task. A\*, while slightly slower, will almost always find the optimal path.

The biggest hindrance is the limitation of using only four directions. It is clear that having diagonal movements could potentially lower the searched nodes by more than half, which is a huge optimisation, and is definitely an avenue for me to pursue.

In future, I would like to streamline how heuristics are performed, as currently I suspect I am wasting a lot of CPU cycles calculating those during the search. I would also like to implement a better way to show the path. I have an idea about using more custom ascii characters to do this.

# References

1. Belwariar, R 2016, *A\* Search Algorithm*, Geeks For Geeks, viewed 20/04/19, <<https://www.geeksforgeeks.org/a-search-algorithm>>
2. Hayden, D 2016, *Breadth-First Search And Shortest Path In C# And .NET Core*, Koder Dojo, viewed 20/04/19, <<http://www.koderdojo.com/blog/breadth-first-search-and-shortest-path-in-csharp-and-net-core>>
3. Hayden, D 2016, *Depth-First Algorithm In C# And .NET Core*, Koder Dojo, viewed 20/04/19, <<http://www.koderdojo.com/blog/depth-first-search-algorithm-in-csharp-and-net-core>>
4. Microsoft 2018, *Tuple Class*, Microsoft Developer Network, viewed 22/04/19, <<https://msdn.microsoft.com/en-us/library/system.tuple(v=vs.110).aspx>>
5. Microsoft 2018, *SortedList Class*, Microsoft Developer Network, viewed 23/04/19, <<https://msdn.microsoft.com/en-us/library/system.collections.sortedlist(v=vs.110).aspx>>
6. Wikipedia 2019, *A\* Search Algorithm*, Wikipedia, viewed 24/04/19, <[https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)>
7. Maze Solving Algorithms 2018, *Maze-Solving Algorithms*, viewed 24/04/19, <[https://github.com/theJollySin/mazelib/blob/master/docs/MAZE\_SOLVE\_ALGOS.md#random-mouse](https://github.com/theJollySin/mazelib/blob/master/docs/MAZE_SOLVE_ALGOS.md%23random-mouse)>