Christopher Carr

Manchester metropolitian university

Artificial intelligence principles – 1cwk100

# Abstract

A common problem faced by mobile robots is the ability to find a path to an objective which it needs to reach. This is known as mobile robot path planning, and “the goal of mobile robot path planning is to find a path from the current position to the target position” (Yu et al. 2020). There are currently many different methods of searching for a solution to this problem. These solutions are known as search algorithms, or path finding algorithms.

Within this report, the following algorithms are introduced and assessed for their efficiency of solving the problem: Tree Depth First Search, Tree Breadth First Search, Tree Uniform Cost Search, Graph Depth First Search, Graph Breadth First Search, Graph Uniform Cost Search, Tree A\* Search, and Graph A\* Search.

A model environment is created with a start location and goal location. The former algorithms are then assessed as to their efficacy in solving the problem in the model environment.

# Search Algorithms

Each of the search algorithms presented below will play a role in helping solve the mobile path finding problem. They define a method through which we can search for a target location from a start location – via graphs and trees.

## Graph

“A graph is a mathematical structure for representing relationships.” (Peng, 2021). Graphs consist of nodes, which contain edges to neighbouring nodes. Only neighbouring nodes can be travelled to from any specific node. A brilliant example of a graph would be the UK train network, pictured below in Figure 1. The graph contains nodes (stations) and edges (train tracks).



*Figure 1, UK Train Network (Graph)*

## Tree

A tree is a graph with more rules: a tree can only have 1 root node; a circle cannot be discovered within the graph; each node can have at most a single parent; a node cannot be a parent of itself.

Figure 2 shows a binary tree. Each node has 2 children, non-cyclic dependencies, and is not disjoined.

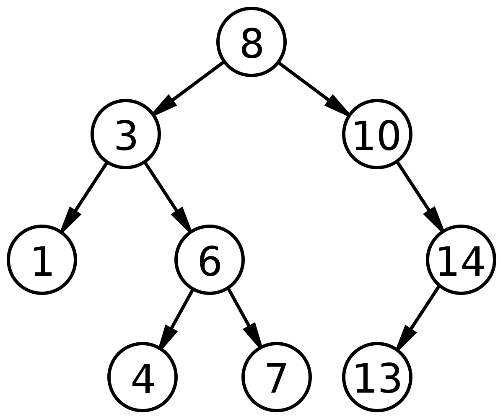
*Figure 2, Model Binary Tree (Tree)*

Figure 1, the UK train network, cannot be considered a tree. While it does not contain any disjoined nodes, it contains cyclic dependencies.

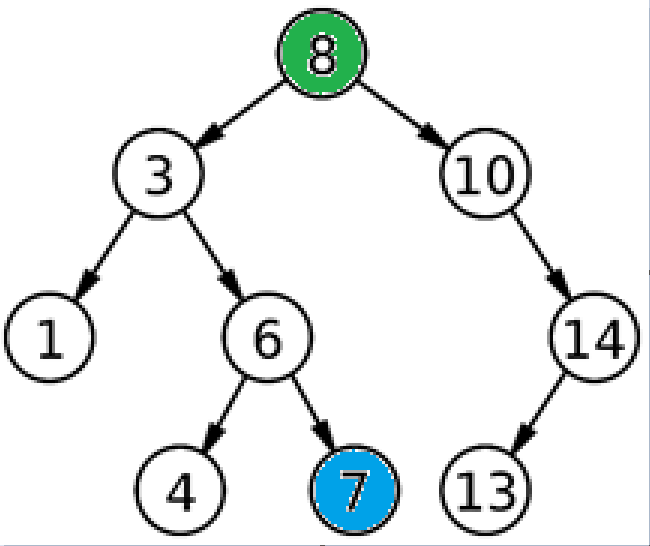
## Breadth First Search

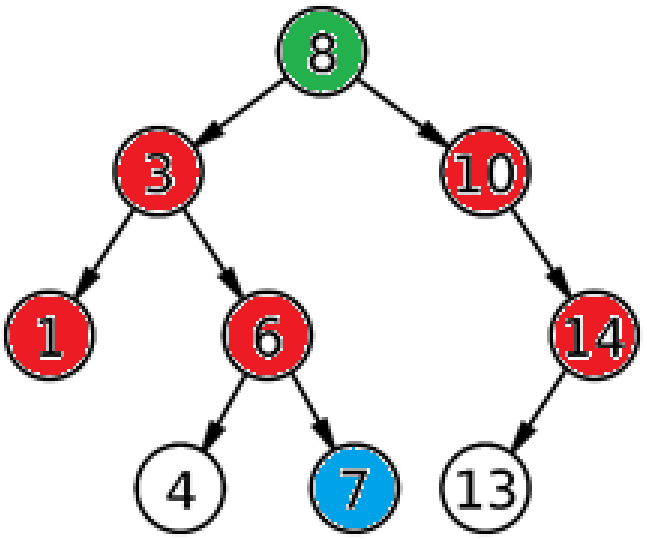
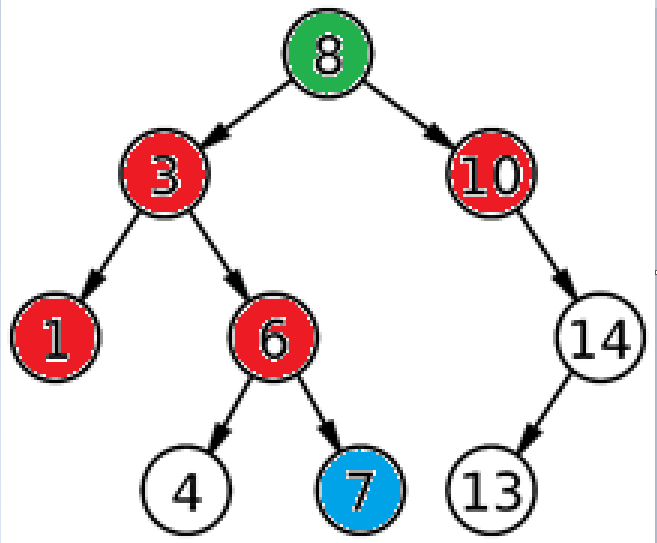
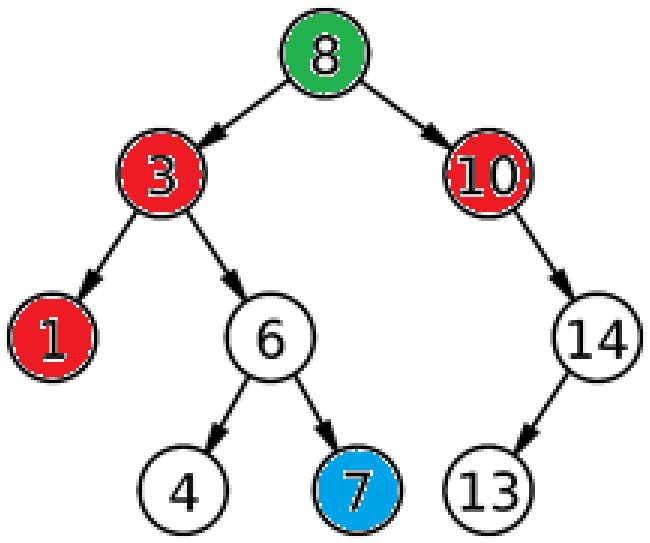
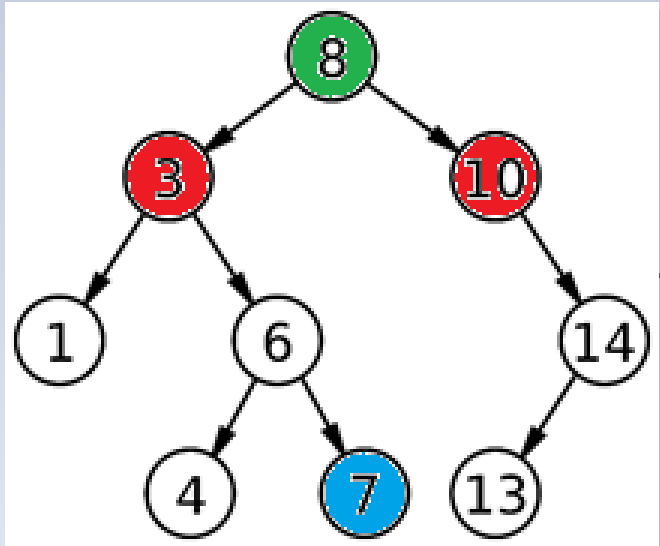
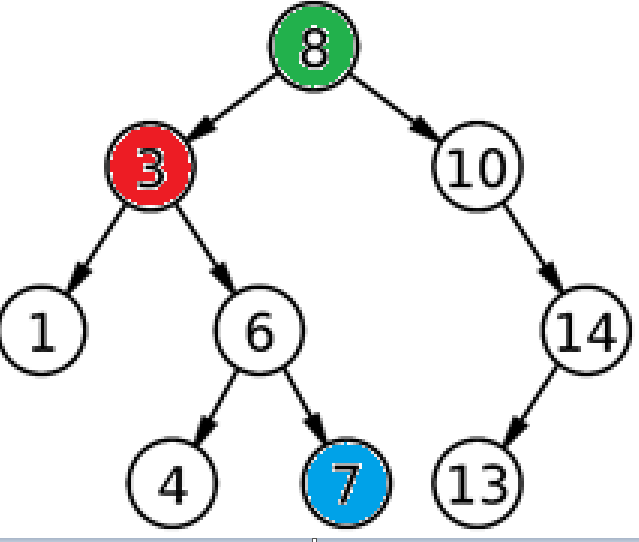
Breadth First Search is a method of searching either a tree or a graph from a route node to a target node. All children nodes are searched, then all children of those children, on repeat.

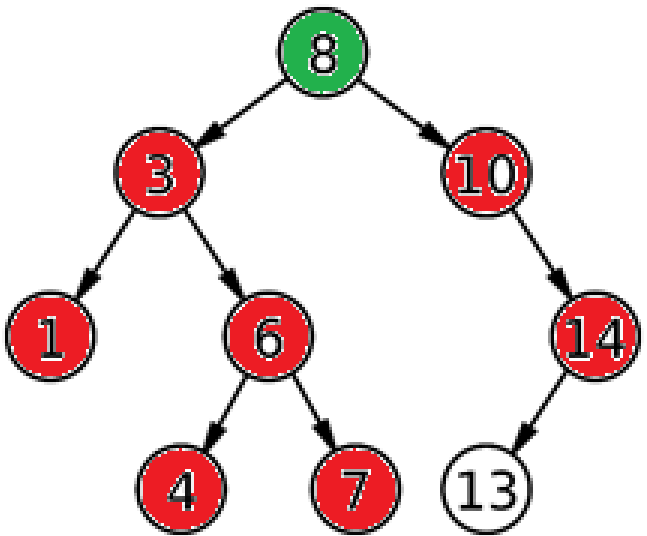
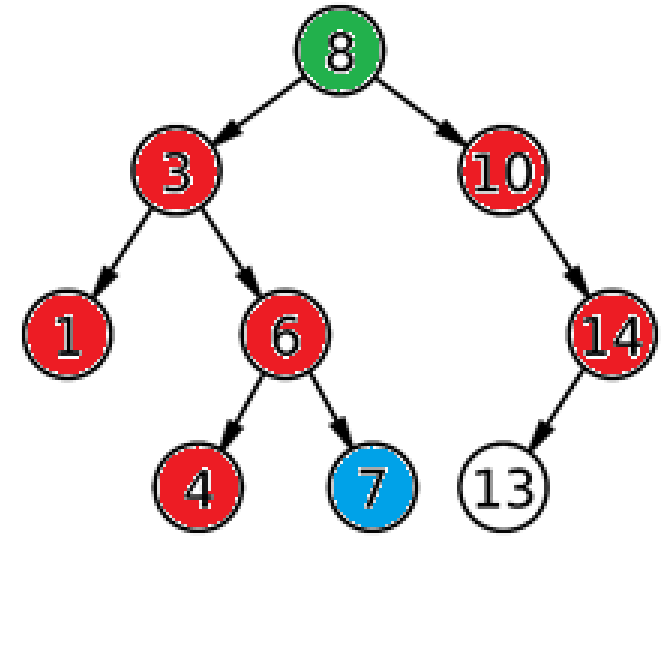
As steps, the search algorithm searches as follows:

1. Get all child nodes
2. Check if child nodes are target nodes
3. If not, get children of all child nodes
4. Search all the children of child nodes (n-2)
5. Search all the children of those nodes (n-3)
6. Search all the children of the graph/tree *width* wise, until target is found (n-k)

This can be displayed with the help of Figure 2. The following figures show how the Breadth First Search will search the tree until it finds its target node. The root node is green, and the target node is blue. Searched nodes are red.

 *Figure 3, problem description*





*Figures [4, 5, …, 10], breadth first search on a simple a graph. (Left to right, Top to bottom.)*

**Pros**: complete solution, finds relatively short path.

**Cons**: high cost, more memory required.

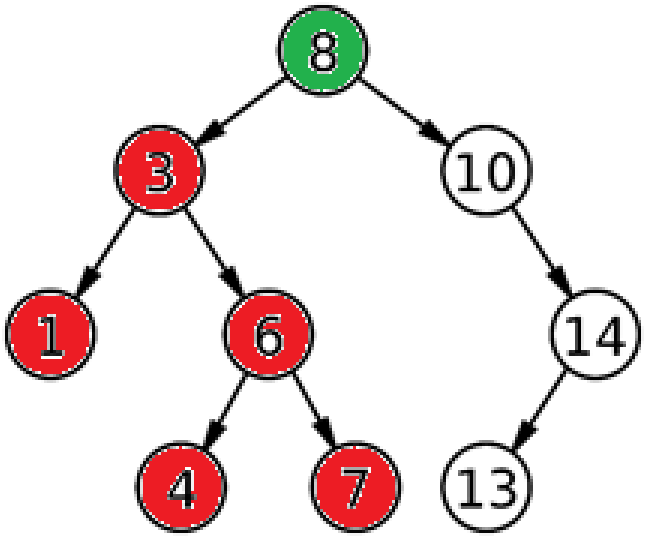
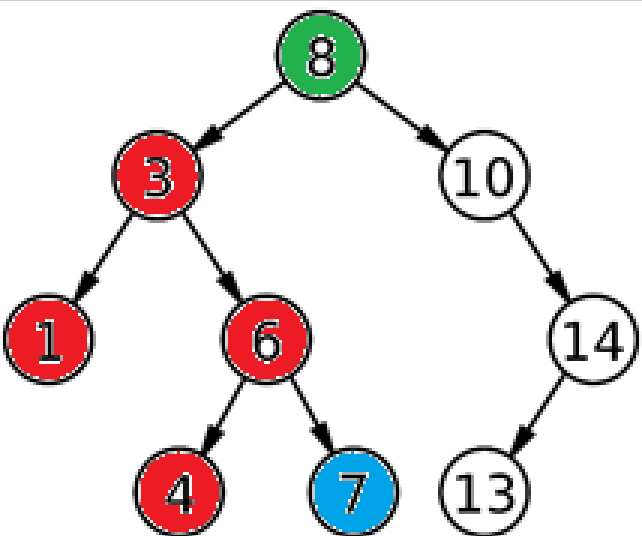
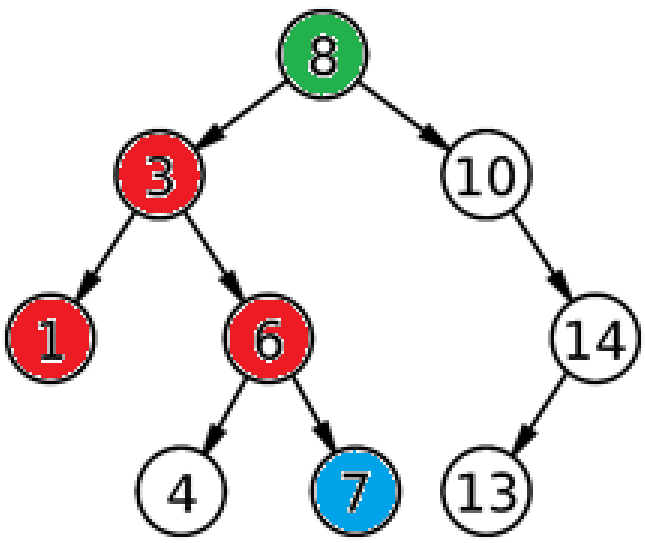
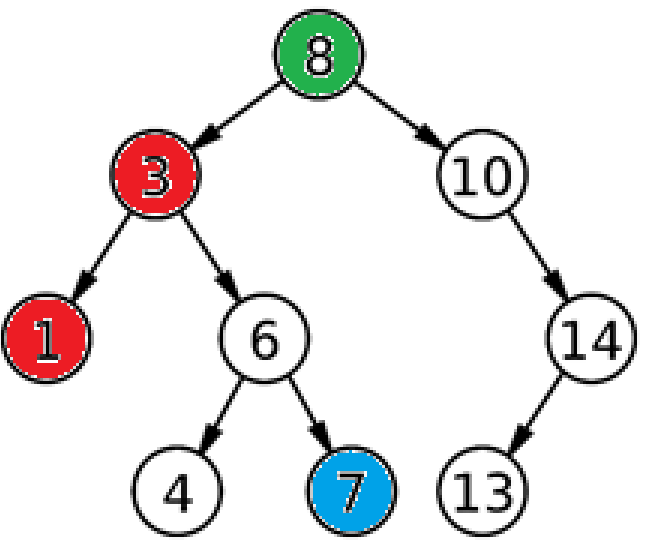
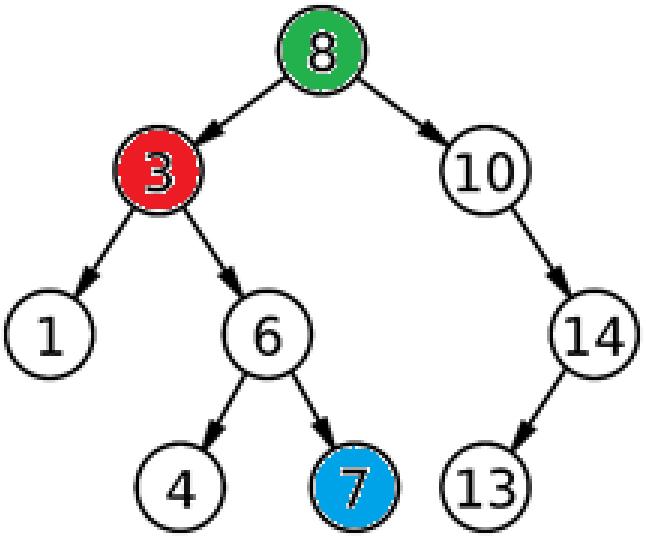
**Search complexity**:

## Depth First Search

Depth first search is like breadth first search in that it traverses a graph or tree from a root node in search of a target node.

Depth first search instead finds all children until it reaches the *nth* child (last child), and searches from the bottom upwards.

In images:



*Figures [11, 12, 13, 14, 15] depth first search on a simple graph (left to right)*

Like breadth first search, depth first search is also problem complete.

**Pros**: complete solution, not much memory used

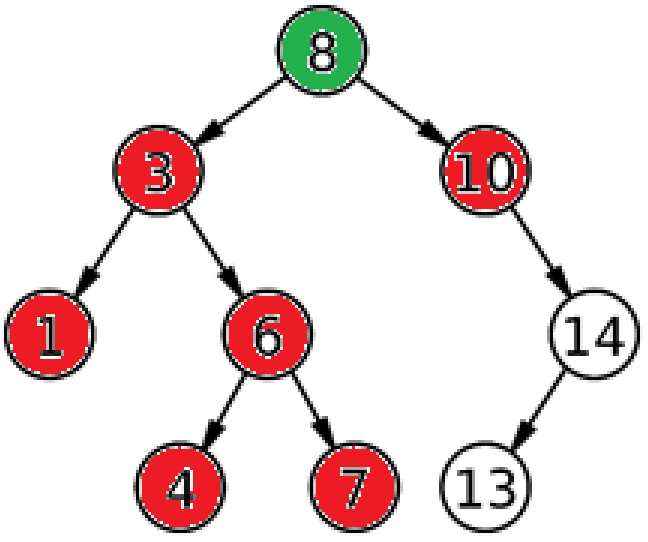
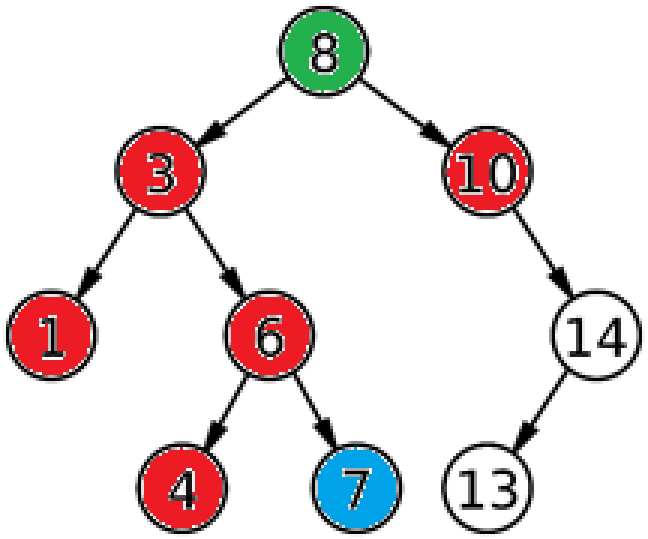
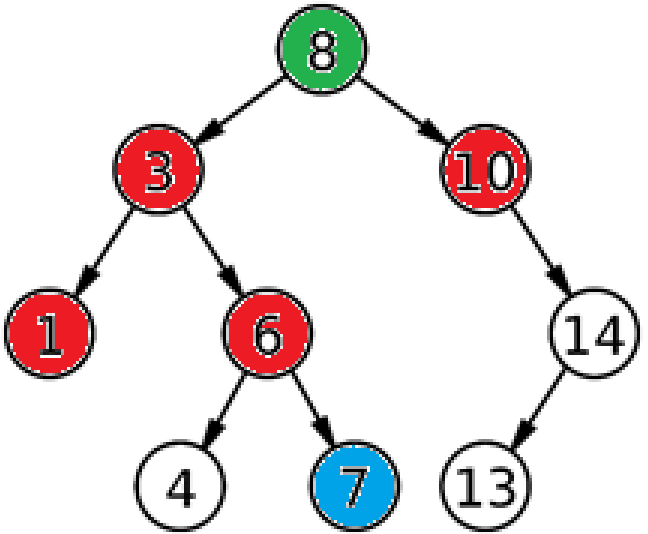
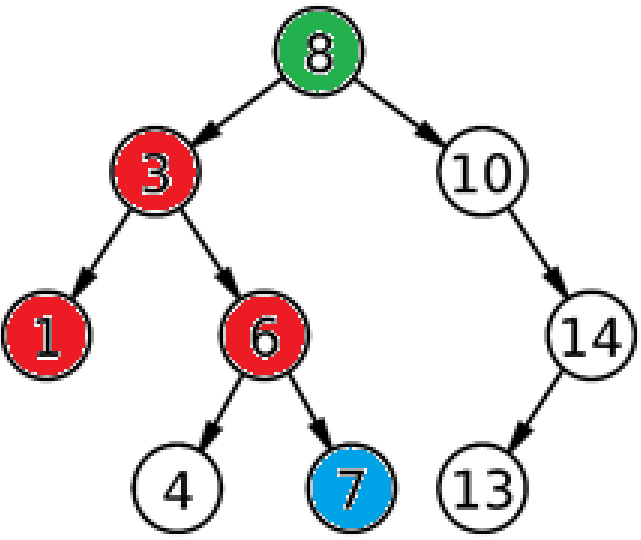
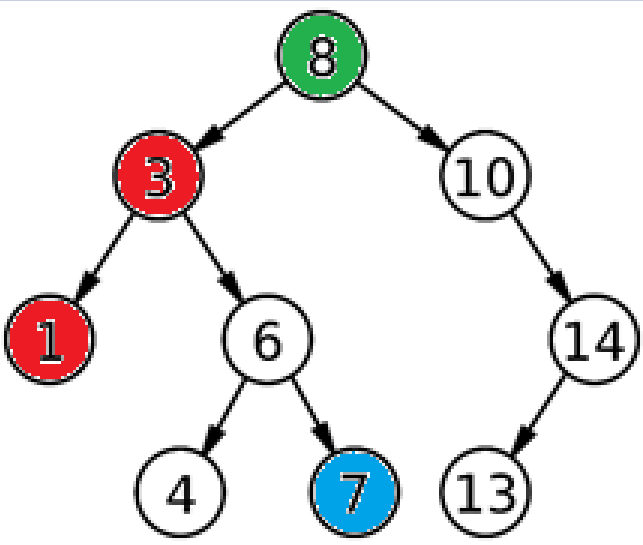
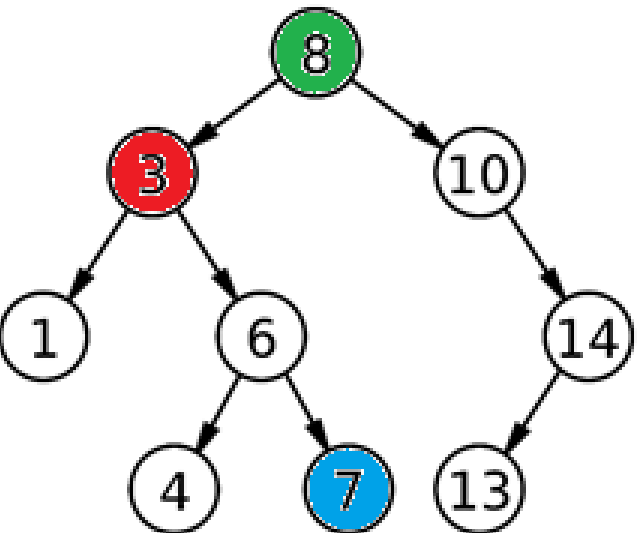
**Cons**: Can search forever (if there is no bottom - *nth* child is infinite), non-optimal solutions

**Search complexity:**

## Uniform Cost Search

Uniform cost search searches the graph in a completely different way to the previous methods. It prioritizes the next node to search based off a value known as the uniform cost. The uniform cost of a node is the cost of all previous nodes on the path to that node. The node with the lowest uniform cost will be searched.

In images:



*Figures [16, 17, …, 22] uniform cost search on a simple graph (left to right).*

**Pros**: will find the most optimal path, complete solution

**Cons**: large memory consumption, can take an exponential amount of time to complete

**Search complexity:**  where C\* is the count of nodes prior to objective.

## A\* Search

A\* search is a type of informed search. It is different to the previous methods in that it requires more information on the problem to solve. A\* is a heuristic method for finding the target node on a graph. It uses a value known as *h* to help it locate the target node and uses this value to influence its decision on which node to search next. The *h* value is usually either the *Manhattan Distance* to the node or *Euclidean Distance* to the node.

*Manhattan Distance* is simply the sum of the absolute differences of each axis of the nodes. This value, the *Manhattan Distance/Euclidean Distance* is used in addition with the total path cost to check which node will be searched next. The total path cost is added to the *h* value to prioritize the node. The resulting equation is usually represented as *g+h*.

**Pros**: complete, optimal, efficient

**Cons**: can become extremely complex

**Search complexity:**  where B is *branching factor*.

# Environment

The model environment is created with the help of Python and NumPy.

The environment is represented in 2 ways, mathematically and visually. For the mathematical representation of the environment, NumPy is used due to its ability to deal with arrays in a very fast manner.

## Mathematical Representation of the problem

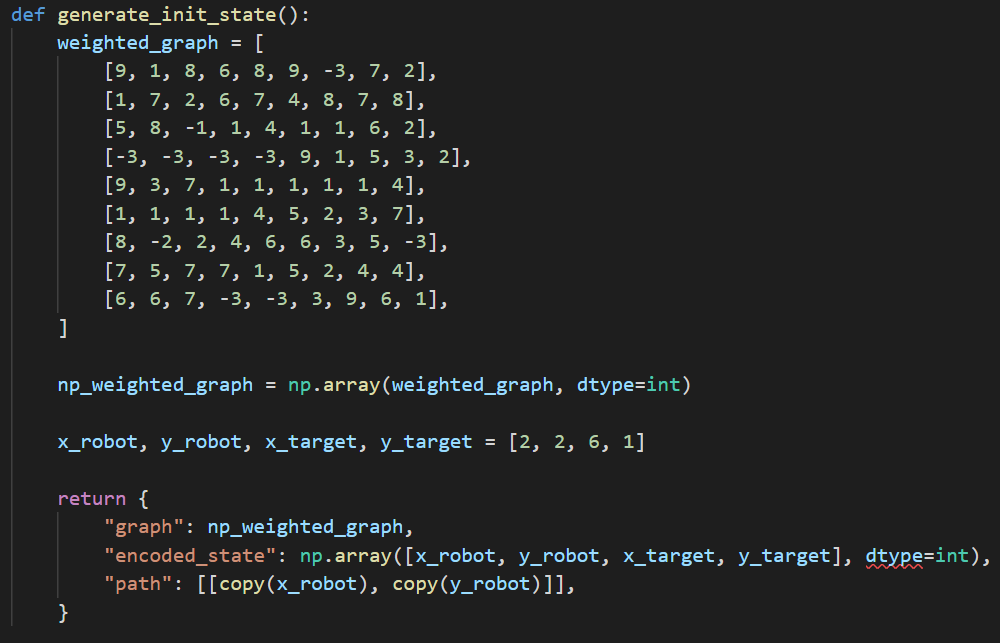
The problem is a vector matrix, with each node being a representation of the cost to reach that node from any neighbouring node. For instance, if there is a neighbouring value of 6, the cost to reach this node is 6. There are some special values within the matrix, identified with a negative value. Below, table 1, shows a full representation of each of the values and their meaning.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| -1 | Robot location |
| -2 | Goal location |
| -3 | Cannot travel to this node |
| 0-10 | Cost of travel to node |

*Table 1, values of the state matrix and their meaning*

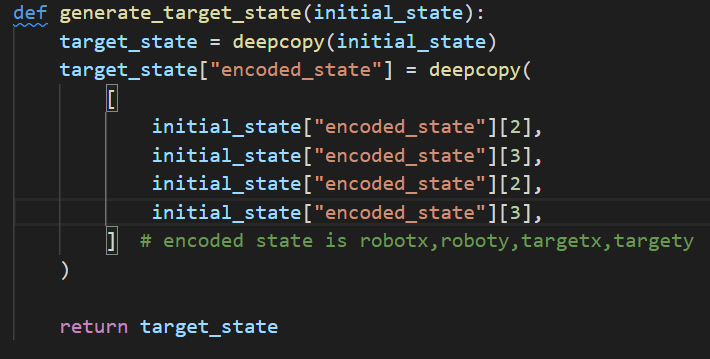
Additional values can be added to the state with ease, being a mathematical representation of a new meaning.

The weighted graph is created using Python lists converted into a NumPy array. The weighted graph becomes a representation of the problem. Figure 23 shows the function which creates the initial state in the program.



*Figure 23, using NumPy to generate an initial state, and problem*

With the initial state and problem created, a new state is created which is a representation of the target state. The target state for the aforementioned problem is when the robot has reached its objective, i.e. when robots x and y axis match the targets x and y axis. Figure 24 displays the method through which a target state is created.



*Figure 24, generating a target state with “deepcopy”*

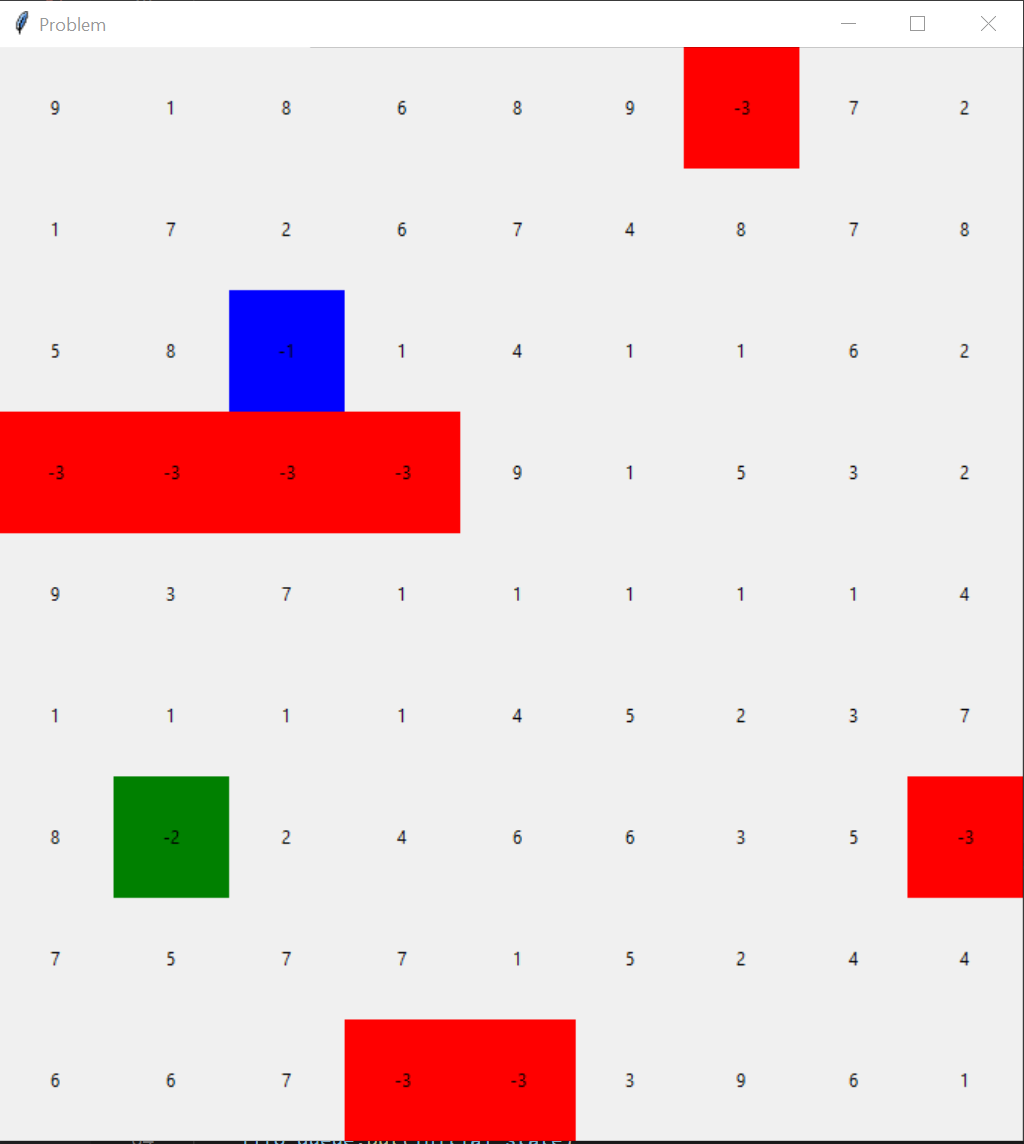
## Visual Representation of the problem

It is much easier to understand the information which is being displayed if we view the information visually, rather than just the mathematical form of it.

A library known as Tkinter is used to display the state matrix as a grid for easier visual consumption. The Tkinter library comes with inbuilt grid functionality, which helps substantially. Figures 25 and 26 show the method which draws the initial state of the problem, and the resulting output of the method.



*Figure 25, drawing a provided state with Tkinter*



*Figure 26, output of “draw\_state” function*

Figure 26 is an easy-to-understand view of the matrix, with colours representing special values within the state. It is worth to mention the meaning of the colours on the grid above. Table 2, ‘colours, and their meaning on the state grid’, is a key for the grid.

|  |  |  |
| --- | --- | --- |
| Value | Colour | Meaning |
| -1 | Blue | Robot location |
| -2 | Green | Target location |
| -3 | Red | Location which cannot be moved to |
| 0-10 | Grey | Cost of travel to node |

*Table 2, colours, and their meaning on the grid*

Each of the search algorithms will in turn draw a path from location “-1” to location “-2” on the grid.

## State encoding

State encoding is a way of representing the state. Rather than using the full graph to represent the state, instead it is possible to encode the state for memory efficiency and speed. In the environment, the initial generated graph never changes. Instead, the representation of the state is the only thing that changes. The encoded state is simply represented as:

[

*“robot x position”,*

*“robot y position”,*

*“target x position”,*

*“target y position”*

]

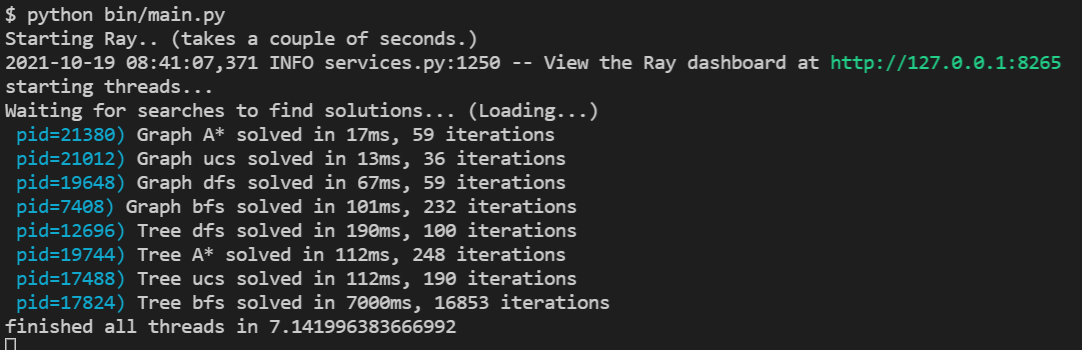
The encoded version of the state can then be used for quicker comparisons than using the full weighted graph as a representation of state for comparison.

## Hypothesis for Results

The estimation here is that the A\* algorithm will produce the lowest cost path, while still being relatively cheap in comparison to other search algorithms. The reason for this hypothesis is due to the extensive use of A\* within complex problems – it is a tried and tested algorithm.

# Results

The algorithms are all ran simultaneously with the help of multi-processing in Python. The output of the code when ran is displayed in Figure 26 – note separate pid’s (process id’s). The reason for multi-processing is due to the length of time each of the pathfinding algorithms takes to complete. Each of the problems are solved in tandem with each other, thus effectively minimizing the run time of them.



*Figure 26, output of code*

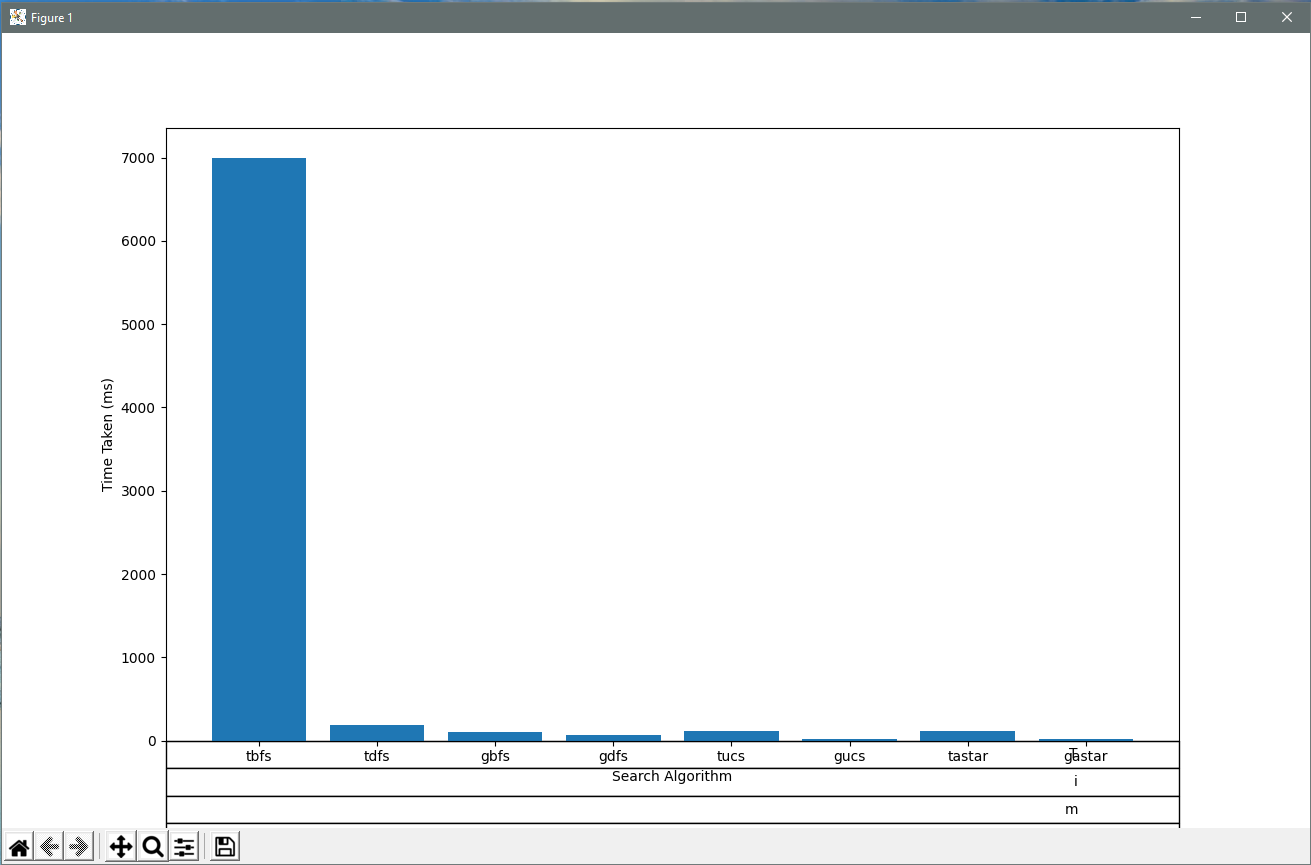
## Results summary

Table 3 is a collection of each of the states, paths, etc, used by each algorithm, for ease of reference. It may be easier to notice here the difference in path between Breadth First Search and Uniform Cost Search/A\* search. While breadth first search discovered the *shortest* path, it did not in fact discover the *lowest cost* path. *N.B tree depth first search removed from table.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Graph, with route drawn** | **Time taken** | **Iterations** |
| Tree Breadth First Search |  | 7000ms | 16853 |
| Graph Breadth First Search | ^ | 101ms | 232 |
| Graph Depth First Search |  | 67ms | 59 |
| Graph Uniform Cost Search |  | 13ms | 36 |
| Tree Uniform Cost Search | ^ | 112ms | 190 |
| Tree A\* Search |  | 112ms | 248 |
| Graph A\* Search | ^ | 17ms | 59 |

*Table 3, routes taken by each search algorithm*

The time taken for each graph to complete is also shown below in figure 27, presented as a bar chart using matplotlib.



*Figure 27, time taken for search algorithms to complete, in micro-seconds.*

Uniform cost search was the best performing algorithms on both a Tree and a Graph. The likely reason for this is that the route to the objective is one of the lowest uniform cost paths available. Through chance, uniform cost has outperformed A\* search. There are a lot of Nodes with a cost of 1 on the route to the objective, resulting in Uniform Cost Search finding the objective rapidly, with a very cheap cost.

## Breadth First Search

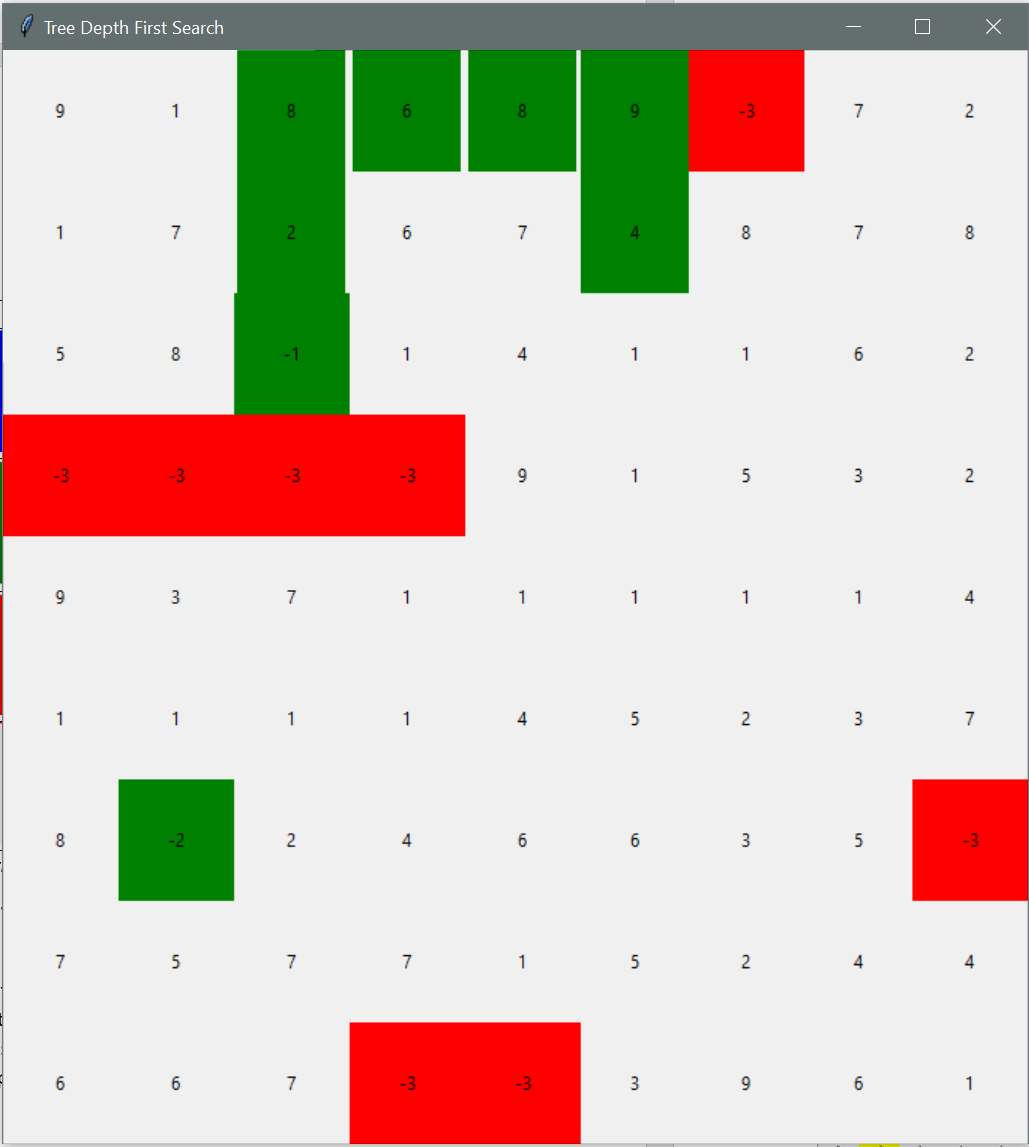
Note the effective route that the algorithm used to find its solution. The solution found by tree breadth first search here is extremely effective. Tree breadth first search is also the costliest though, with back-referencing on Table 3 showing over 7000ms to complete (7 seconds), and 17000 iterations!

Graph breadth first search produces similar output, while being among the cheaper to solve (130ms). The route taken by graph breadth first and tree breadth first are the same – thus no new image will be supplied.

**Breadth first search performed better on a graph. The reason for this is due to the relatively simplicity of the graph.**

## Depth First Search

Tree depth first search is among one the worst algorithms. The algorithm had to be limited to 100 iterations due to its inability to find a solution the problem. The tree would repetitively run to infinity, walking left and right, and never solve the problem. Figure 28 shows the route that tree breadth first search took. The resulting path is nowhere near the solution to the problem, and this is with over 100 iterations.

*Figure 28, Tree depth first search*

A much better result with depth first search was when using a graph to search the problem. While still among the worst, graph depth first search at least managed to solve the problem. There are poor results all round from depth first search within the experiment; both tree and graph depth first search performed awfully.

It should be noted that this does not make depth first search an ineffective algorithm overall. There are many other situations in which depth first search will be the most effective algorithm. It would appear though that depth first search is not a good algorithm for the model pathfinding problem presented in the paper.

**Depth First Search performed better on a graph. A tree-based model problem was too complex for Depth First Search to solve.**

## Uniform Cost Search

Uniform cost search was one of the better performing algorithms. It produced the most optimal solution to the problem. The reason for the high cost of Tree uniform cost search, is the ability of Tree uniform cost to “run in cheap circles”. The search method would iterate between low-cost nodes, moving left and right, until the cost of moving closer to the target node was more expensive. In other terms, tree uniform cost search was cut short due to its tendency to propagate in ***all*** the local optima (low-cost paths) before finding the global optima (lowest cost path to target).

Graph uniform cost search created a much better solution to the problem, discovering the global optima in a whopping 13ms! **Uniform cost search over a graph was in fact the quickest of any of the algorithms to find a solution.** The irony here is that the sorting, scoring, popping/queuing of the paths were all done manually here, rather than using the queue object. Thus, the fact that Graph Uniform Cost Search was the quickest may in fact be invalidated (unfair test – queue timing vs array timing).

A final note on uniform cost is the difference in path it took to reach the objective from breadth first search. While both algorithms resulted in a path of 8 movements, the cost of uniform cost is less.

**Uniform cost search was the best performing algorithm. It performed better on a graph.**

## A\* Search

A\* was easily among the best performing searches used. A\* search provided the 2nd quickest solution (26ms), whilst also being able to solve the problem in both a tree and a graph. A\* search was outperformed on this problem by uniform cost search. It is likely that the reason that A\* was outperformed is due to the relative simplicity of the cheapest path to the objective.

An interesting thing to note is that A\* did not find the cheapest solution, which is unexpected.

**Graph A\* searched performed better than Tree A\* search.**

# Bibliography

1. Yu, J., Su, Y. and Liao, Y. (2020) "The Path Planning of Mobile Robot by Neural Networks and Hierarchical Reinforcement Learning", *Frontiers in Neurorobotics*, 14. doi: 10.3389/fnbot.2020.00063.
2. Peng, W. (2021) “Artificial Intelligence Principles”, *Basics and Prerequisites of Search Algorithms, N/A*