

Artificial Intelligence Principles

A REPORT SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY FOR
THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING



2021

By
Christopher
Carr

School of Computing, Mathematics and Digital Technology

Contents

Acknowledgements	3
Abbreviations	4
Abstract	5
Chapter 1 – Search Algorithms	6
1.1 Graphs	6
1.2 Trees	7
1.3 Breadth First Search	7
1.4 Depth First Search	7
1.5 Uniform Cost Search	8
1.6 A* Search.....	8
Chapter 2 - Environment	4
2.1 Mathematical representation of the problem	4
2.2 Visual representation of the problem.....	4
2.3 State encoding	6
2.4 Robot Parameters	6
2.5 Hypothesis	7
2.6 Methods	7
Chapter 3 – Results.....	8
3.1 Results summary	8
3.2 Breadth First Search	11
3.3 Depth First Search	11
3.4 Uniform Cost Search	11
3.5 A* Search.....	12
3.6 Discussion.....	12
Appendix A.....	13
References.....	14

Acknowledgements

I would like to thank Peng Wang, my tutor for the subject, for an exceptional teaching of the subject.

Abbreviations

BFS	Breadth First Search
DFS	Depth First Search
UCS	Uniform Cost Search
A*	A Star Search
T-BFS	Breadth First Search on a Tree
G-BFS	Breadth First Search on a Graph
T-DFS	Depth First Search on a Tree
G-DFS	Depth First Search on a Graph
T-UCS	Uniform Cost Search on a Tree
G-UCS	Uniform Cost Search on a Graph
T-A*	A star search on a Tree
G-A*	A star search on a Graph
TASTAR (used in figures)	A Star Search on a Tree
GASTAR (used in figures)	A Star Search on a Graph

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: T-DFS, T-BFS, T-UCS, G-DFS, G-BFS, G-UCS, T-A*, and G-A*.

Results indicate that the A* algorithm is by far the most effective in the model environment.

A* search was the quickest, lowest cost, most memory efficient search algorithm. A* was the best performing algorithm on both a tree and a graph, substantiating its’ usage within real-world applications.

Chapter 1 – 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.

1.1 Graphs

“A graph is a mathematical structure for representing relationships.” (Peng, 2021). Graphs consist of nodes, which contain edges to neighboring nodes. Only neighboring 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)

1.2 Trees

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 disjoint. Figure 1, the UK Train Network, cannot be considered a tree. While it does not contain any disjointed nodes, it contains cyclic dependencies.

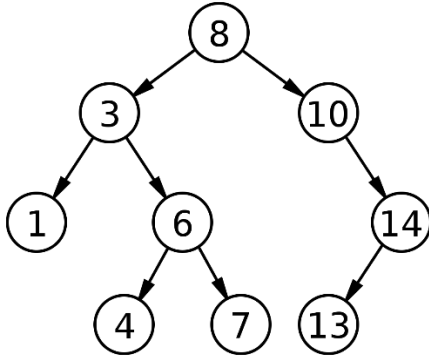


Figure 2, Binary Tree

1.3 Breadth First Search

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

This can be displayed with the help of Figure 3. The following figures show how the BFS 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.

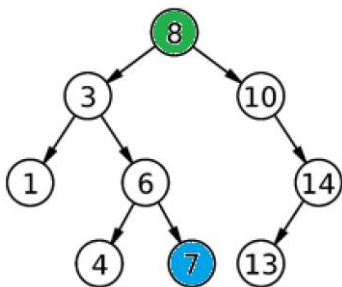
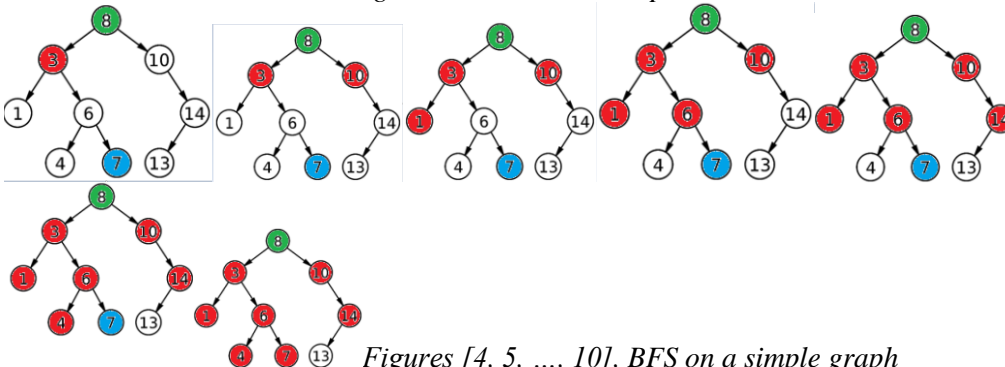


Figure 3, Problem Description



Figures [4, 5, ..., 10], BFS on a simple graph

Pros: complete solution (always finds solution), finds relatively short path.

Cons: high cost, more memory required.

Search complexity: $O(|N| + |E|)$ where N is nodes and E is edges

1.4 Depth First Search

DFS is like BFS in that it traverses a graph or tree from a root node in search of a target node.

DFS instead finds all children until it reaches the n^{th} child (last child), and searches from the bottom upwards.

In images:

Chapter 2 - Environment

It is possible to build an environment to test each of the algorithms. The environment will contain a robot, with a starting position and target position. The robot will simply be a point on the graph, with a target location represented as another point on the graph. Much like all the previous figures, there is then an initial state (robot start position), and end state (robot target position). The search algorithms can then use the robot to traverse the graph. An environment can be created to test the efficacy of each of the search algorithms.

The environment will be an integer matrix, much like the figures used to describe the algorithms. The model environment is created with the help of Python and NumPy. The environment is represented in 2 ways, mathematically and visually.

2.1 Mathematical representation of the problem

The problem is an integer matrix, with each node being a representation of the cost to reach that node from any neighboring node. For instance, if there is a neighboring 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 becomes a mathematical representation of the problem.

2.2 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.

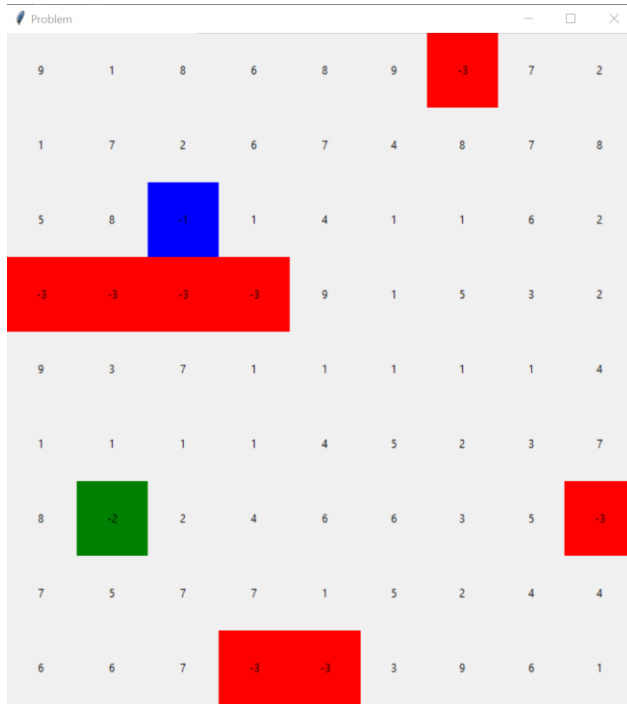


Figure 23, output of “draw_state” function

Figure 23 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 is a key for the grid.

Value	Colour	Meaning
-1	<div><div>-1</div>Blue</div>	Robot location
-2	<div><div>-2</div>Green</div>	Target location
-3	<div><div>-3</div>Red</div>	Location which cannot be moved to
0-10	<div><div>7</div>Grey</div>	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.

2.3 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 optimality. 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.

2.4 Robot Parameters

Simply, the robot is represented as a point on the graph. It would be possible to represent the robot as less than a single point on the graph, however, this massively increases the problem complexity. The action model presented below represents the movement of the robot on the graph, it can move: North, East, South, or West. Each of these movements are considered actions and usually come with a cost associated with them. To keep the robot simple, the cost of movement in any direction is uniform – 1. Figure 24 shows a visual representation of the action model.

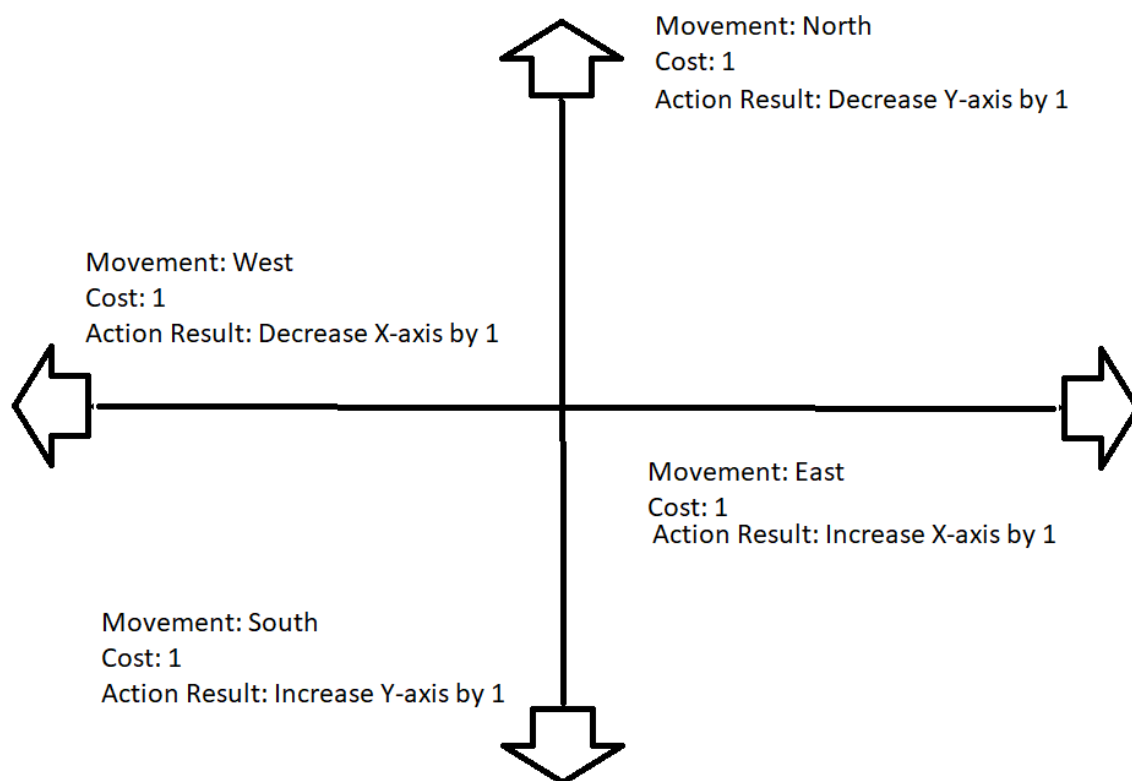


Figure 24, Robot Action Model

2.5 Hypothesis

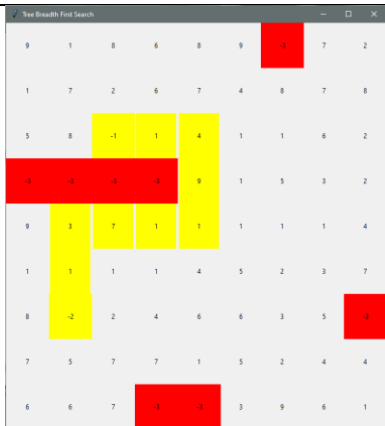
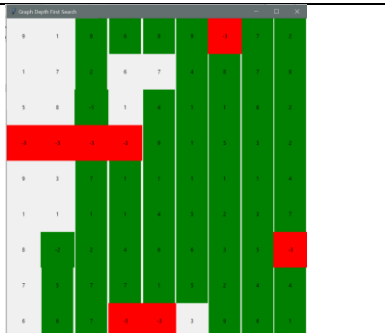
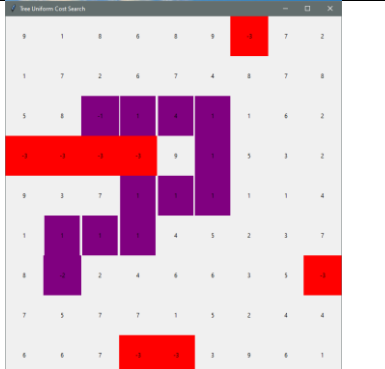
The estimation here is that the A* algorithm will produce the lowest cost path, while still being relatively cheap in comparison to other search algorithms. The reason for this hypothesis is due to the extensive use of A* within complex problems – it is a tried and tested algorithm.

2.6 Methods

The form used to analyze each of the search algorithms and gather results is presented in **Appendix A**.

3.1 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 BFS and UCS/A* search. While BFS discovered the *shortest* path, it did not in fact discover the *lowest cost* path. *N.B. T-DFS removed from table.*

Algorithm	Graph, with route drawn	Time taken	Iterations	Total Path Cost	Memory Usage
T-BFS		7000ms	16853	24	105107456B / 105MB
G-BFS	^	101ms	232	24	380928B / 0.380928MB
G-DFS		67ms	59	229	294912B / 0.294912MB
G-UCS		13ms	36	10	86016B / 0.086016MB
T-UCS	^	112ms	190	10	1441792B / 1.441792MB

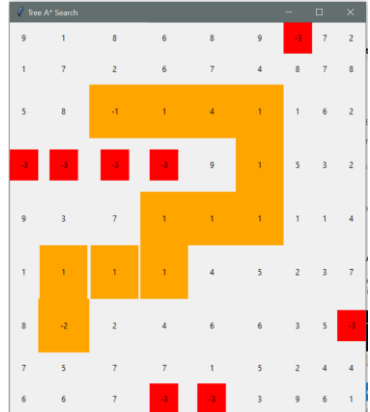
T-A*		4ms	14	10	65536B / 0.065536MB
G-A*	^	5ms	13	10	57344B / 0.057344MB

Table 3, routes taken by each search algorithm

The time taken for each graph to complete is also shown below in Figure 25, presented as a bar chart using matplotlib.

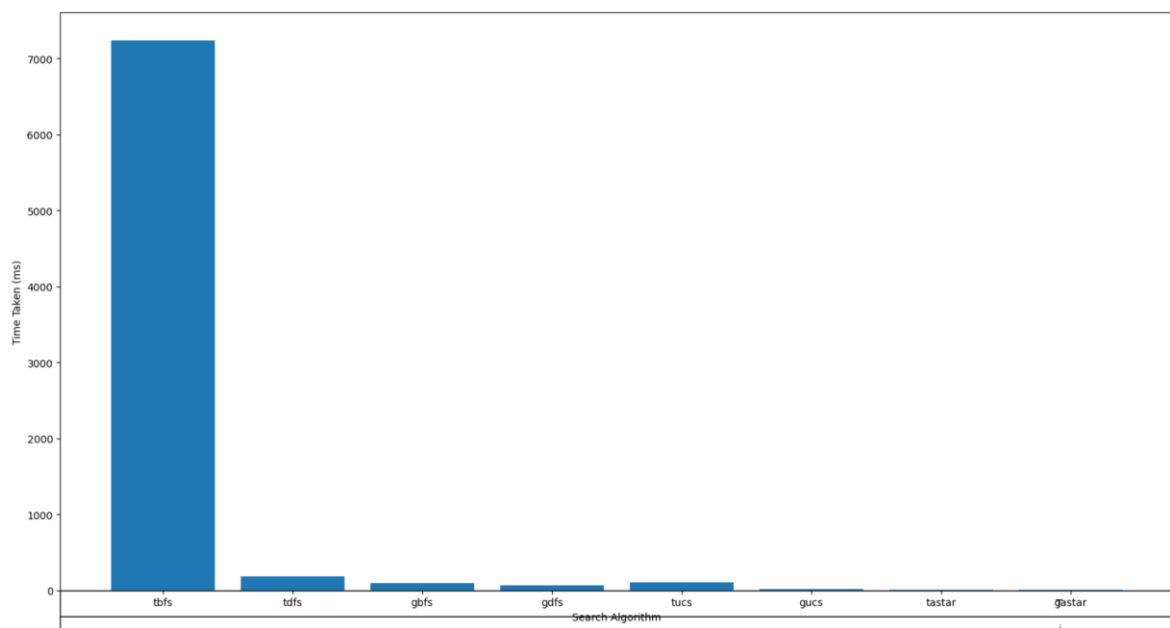


Figure 25, time taken for search algorithms to complete, in micro-seconds.

A* 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, along with the ability to use additional heuristic information by A*. Thus, A* found the solution in the least amount of time, with the lowest iterations of any of the algorithms.

Figure 26 shows a comparison of the total path cost of each algorithm. T-DFS is by far among the worst, with G-DFS a close second. Interestingly, without a requirement for cost, both T-BFS and G-BFS produced low-cost paths.

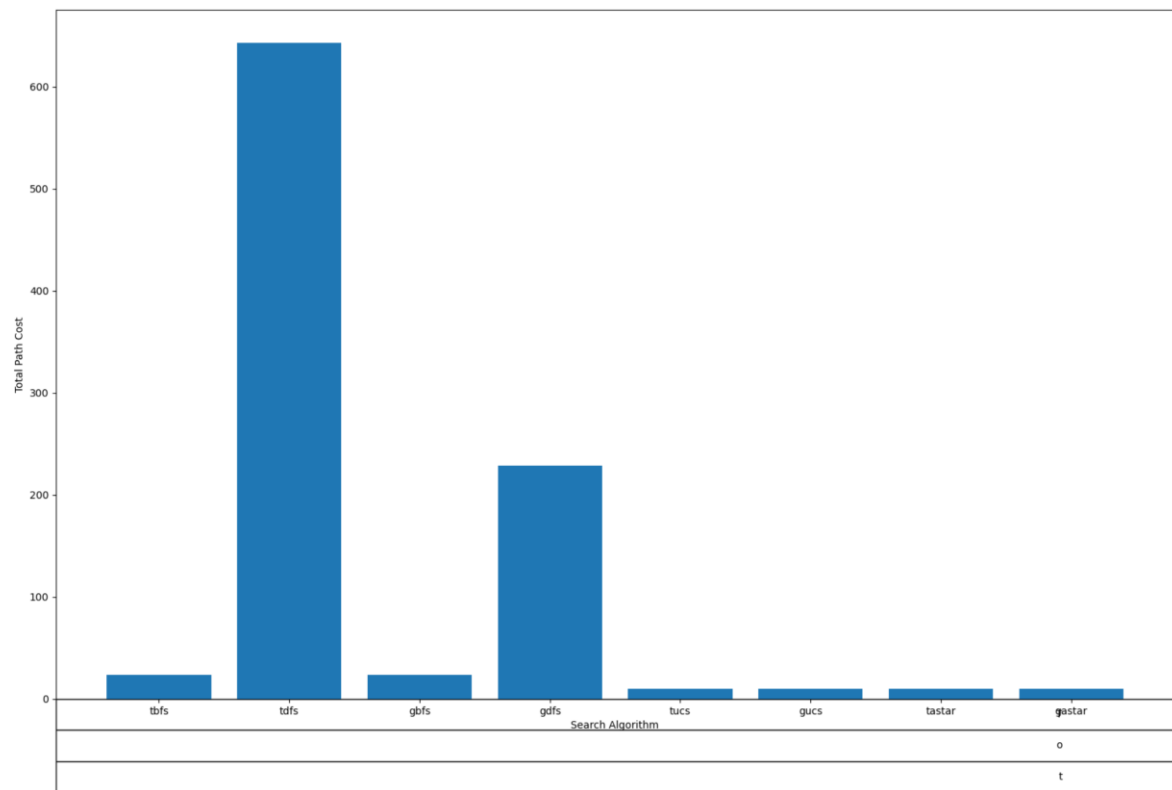


Figure 26, total path cost

Figure 27 shows the memory usage of each program. T-BFS used the highest amount of memory, while A* used the lowest. Consistently, A* is outperforming the other algorithms in a multitude of areas.

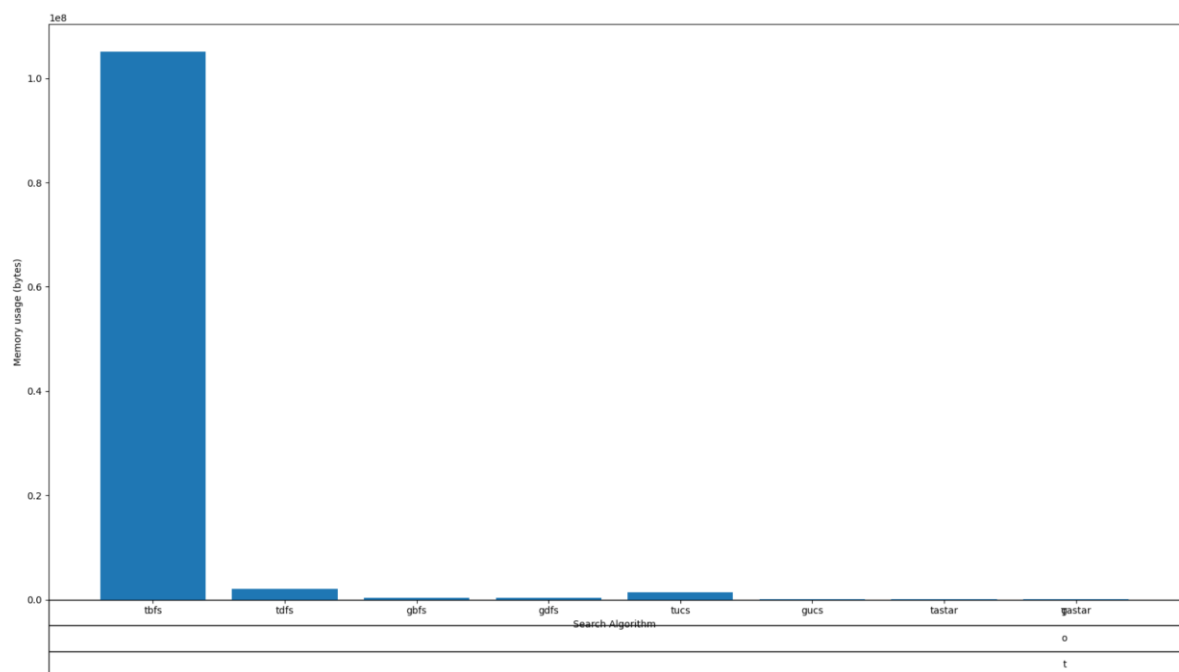


Figure 27, Memory usage of each algorithm

3.2 Breadth First Search

BFS performed better on a graph. The reason for this is due to the relatively simplicity of the graph in comparison to a tree. BFS produced a low-cost path *without the need for heuristics*. T-BFS required the most memory to solve the problem, with over 100MB requirement. BFS can thus be considered effective in model environments, when given a large amount of time to solve the problem. The results dictate that BFS remains an ineffective solution for real-time problems.

3.3 Depth First Search

T-DFS 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 T-BFS took. The resulting path is nowhere near the solution to the problem, and this is with over 100 iterations.

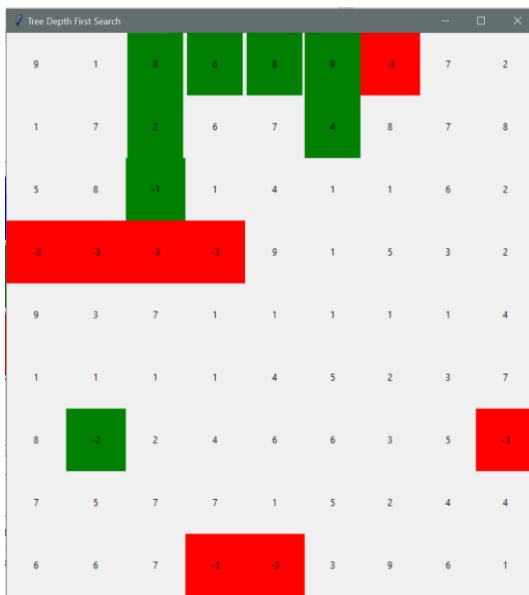


Figure 28, T-DFS

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

DFS performed better on a graph. A tree-based model problem was too complex for DFS to solve.

3.4 Uniform Cost Search

UCS was one of the better performing algorithms. It produced the most optimal solution to the problem. The reason for the high cost of T-UCS is the ability of T-UCS to “run in cheap circles”. High cost here, means the comparison of 190 iteration by T-UCS to the 36 iterations by G-UCS. 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, T-UCS tends to propagate in *all* the local optima (low-cost paths) before finding the global optima (lowest cost path to target).

G-UCS created a more robust solution to the problem, discovering the global optima in a whopping 13ms! A final note on UCS is the difference in path it took to reach the objective from BFS. While both algorithms resulted in a path of 8 movements, the cost of UCS is less.

3.5 A* Search

A* search was the quickest, lowest cost, most memory efficient search algorithm. A* was the best performing algorithm on both a tree and a graph, substantiating its' usage within real-world applications. The reason that A* is the best performing algorithm is due to its ability to permanently be moving closer to the objective, heuristically.

3.6 Discussion

There are some aspects of the search which can be taken further to test different ideas on the pathfinding solution. For example, from this experiment the clear solution is the A* algorithm, completing the problem in a minuscule time in comparison to any other algorithm.

All previous problems presented in the paper are solved on a relatively simple matrix. The matrix is a 9x9 matrix, due to the complexity of some of the algorithms.

The A* algorithm was tested further, to see if it could complete a massively complex problem. The A* algorithm was tested on a 1000x1000 matrix, to see how fast it could solve a complex 2d problem. It took A* 23 seconds to solve the problem on a 1000x1000 matrix. While still the best algorithm for solving complex problems presented in the paper, A* is still ineffective for real-time applications, such as self-driving cars. In real-world applications, problems are much more complex than a 1000x1000 matrix. When presented a problem, it should not take over 23 seconds to find a solution – regardless of problem complexity. It is possible the A* solution used in this paper is lacking performance improvements required for real-world problems. In autonomous flight, a 3d plane (1000x1000x1000), may prove too difficult for the algorithm to solve in anywhere near appropriate time.

Appendix A

Results Form

[illegible]

References

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*,