1 Introduction

This project aims to develop a tree-search algorithm using Python. Initially, the general concept and structure of a search tree will be outlined, along with an explanation of the various search types implemented in the code. The focus will then shift to the Python implementation, including details on code structure and data representation. Finally, the algorithm will be tested on some example problems to evaluate its performance and compare the different search strategies.

2 What is a Tree Search algorithm?

If we consider problem in which the possible states can be represented as nodes of a tree, and the actions can be represented by edges between nodes, we can define a search strategy that builds a tree structure in which the first node is the starting status, and the children are the states that can be reached with a particular action. This can be continued to build a tree-like structure. The goal of the algorithm is, starting from the problem definition, build a tree that represents the problem and find a path that connects the starting node to the goal node through a list of actions.
There exists multiple search algorithms and strategies, but they can be subdivided in one of these two classes:

- Uninformed search algorithm
- Informed search algorithm

To explain this concept, let's consider an example in which a search algorithm is applied to find a path between two points in a map. In this problem each intersection is represented as a state and the edges are the action to move from an intersection to another. We know intuitively that starting from a certain point, to find a possible solution is better to "go" in the general direction of our goal.

An uninformed search algorithm doesn't know if a particular path is leading to a solution state or not. This means that these algorithms can start searching in a direction that will never lead to a solution.

However, some search algorithms have a way to tell if a path or direction is good or not.This algorithm are called *Informed search algorithms*. In the previous example, an informed search algorithm can use the distance between the current position and the goal to evaluate if it's moving in a good direction, toward the target, or not. Using this information the search is more goal-driven and usually more efficient.

2.1 General Structure

The general structure of a search algorithm is the following, represented in python:

```
def tree-search(goal):
  #Fringe list, initialized with the starting node
    fringe = [initialNode]

    #infinite loop, exits when finds a solution or if there is no
    #solution
    while True:
        #if the fringe is empty, there is no solution
        if len(fringe)<1:
            return []

        #choose the next node to be expanded
        NextNode = chooseNextNode()

        #check if the selected node is our goal node
        if NextNode == goal:
            return pathTo(NextNode)

        #expand the selected node and add the childs to the fringe
        childs = expandNode(NextNode)
        fringe.append(childs)
```

This is of course a very simplified representation, where some implementation details have been left out.
The algorithm keeps a list of nodes that are yet to be expanded, called fringe. At each step we choose one node from the list and we expand it. These steps differs depending on the type of search we are performing and the type of problem we are solving. The nodes obtained from the expansion are called childs and are added to the fringe.
If we reach a state in which the fringe is empty, this means that we cannot expand the tree and so we haven't found a solution.
The check of the goal state is performed when a node is chosen for expansion, this is done to guarantee the optimality of certain algorithms.

2.2 Search types

There exists a lot of different search algorithm, this is because the choice is often a compromise. In fact some algorithms are simpler to implement, while other ones perform better in terms of time or used memory.
These algorithms share the same general structure and differs only on the way the next node is chosen for expansion.
Here's a list of the implemented search algorithms with a brief description on their characteristics:

- Breadth-first search (BFS)
- Depth-first search (DFS)

- Greedy best-first search
- A*

2.2.1 Breadth-first search

This type of search is easy to implement and can be useful when all actions have the same cost. This algorithm expands first the nodes with the lowest depth. When all actions have the same cost, this is guaranteed to find the best solution. It's implemented by putting the new nodes at the end of the fringe and expanding always the first node in the list.

2.2.2 Depth-first search

DFS always expands the deepest node in the fringe and "backs up" when it finds a dead end. DFS is not cost-optimal as it returns the first solution if finds, even if it not the cheapest.
The advantage of DFS is the memory usage, in fact when a dead end is found it can be removed from the memory as we are sure that there is no solution there. DFS it's implemented by putting the new nodes at the start of the fringe and expanding always the first node in the list.

2.2.3 Greedy best-first search

Greedy best-first search is an informed algorithm that expands the nodes that are more promising, using the heuristic function to evaluate them.
In its implementation, this algorithm doesn't rely on the order of the nodes in the fringe, but it chooses the nodes with the lowest heuristic.

2.2.4 A*

Greedy best-first search doesn't consider the depth of the nodes, therefore the solution found is not always the best one. A* on the other hand, consider the heuristic and the depth of a node. For this reason, if the heuristic is admissible[1] it will find always the cost-optimal solution.

[1] An heuristic is admissible if it never overestimate the true cost of reaching the goal state

3 Implementation

The implementation is written in Python. Python was chosen because is an Object-Oriented programming language with a large repository of libraries. These characteristics allow to implement algorithm in an high-level way without worrying too much on the low level data representation and processing.
However, the principal disadvantage is the low speed and high memory usage. In fact, in general high level implementations doesn't allow to make low level optimization.

3.1 Code structure

The implementation is based around two principal classes: **Node** and **Tree**.
Node is a very simple class that represent a single node in the tree. It contains some attributes:

- *data*, this represent the state associated with the node;
- *parent*, a reference to the parent node;
- *childs*, a list of the childs of the node;
- *heuristic*, used only with informed search strategies, contains the value of the heuristic function evaluated on the state of the node;
- *actionCost*, contains the sum of the costs of all actions needed to reach this node starting from the initial node;
- *ID*, is an unique identifier of this node. This is needed because more than one node can have the same state associated;
- *hash*, contains the result of an hash function on the data attribute. This is useful in some cases as the comparison between hashes can be much faster than comparing directly the states.

Tree contains the core functionalities of the searching algorithm and tree management. The "core" method is called "find", that implements the search algorithm previously described. Moreover, it contains methods useful to manage a tree and to support the search algorithm.

The algorithm is generalized following the general structure previously described. However some functions are problem-dependant, for example: **expandNode()**, **getHeuristic()** and **getActionCost()**. Their specific implementation will be explained later. Moreover, the heuristic is not intrinsic of the problem, but different heuristic can be applied to the same problem.

4 Problems definition

In this section, the implementation will be tested by applying it to two different problems, evaluating the results. In the first part, the problems will be defined, mentioning the problem-specific details, aftet some tests will be executed and the results analyzed.
Following the previously defined code structure, for each problem is defined a new object that inherits the Tree class. In this way the fundamental code is used in each variation, while the custom problem-specific code is defined on top of that.

4.0.1 Pathfinding

In this problem, I'm considering the case in which we need to find the path between two points in a map. For the map representation, a image can be used in which white pixels are considered walkable and black pixel are considered obstacles. As I have discussed previously, there are some methods in the code that are problem-specific and thus needs to be defined. They are:

- *expandNode()*: Checks for possible paths around the expanded node;
- *getHeuristic()*: The available heuristics in this case are: LineOfSight and Manhattan distances.
- *getActionCost()*: Each movement is of one pixel, thus the cost is always the same and fixet at 1.

4.0.2 NPuzzle Solver

N Puzzle is a sliding blocks game that takes place on a $k*k$ grid with $(k*k)-1$ tiles each numbered from 1 to N. The goal of the puzzle is to reorder the tiles in a numerical order.
The best know versions of this puzzle are the 15 and 8 versions. The first contains 15 tiles in a 4x4 grid while the second have 8 tiles in a 3x3 grid.
In this case, each node saves the board status as a 3x3 or 4x4 matrix.
The abstract methods are defined as:

- *expandNode()*: Checks for all possible moves starting from the current status, of which there can be up to four;
- *getHeuristic()*: For this problem a possible heuristic can be the sum of the *Manhattan distances* from each tile to its goal position.
- *getActionCost()*: Each possible movement is equal, so the action cost is fixed to 1.

The test setup contains an interesting aspect, how do we generate the puzzle to be solved? In fact, not every combination of blocks is solvable and thus they are not very useful for the evaluations. There exists some mathematical theories

and some derived tests to understand if a puzzle is solvable or not, however a simpler method is to start from a puzzle in the solved state and then do a number of random moves. Using this technique we can generate puzzles that are for sure solvable and with the added bonus that we can choose the maximum depth of the solution[2]. In fact, if we make, for example, 10 random moves we know that the solution must be at maximum at depth 10.

[2]It can be noted that the maximum depth of the best solution is also limited by the puzzle size

5 Testing Results

5.1 Pathfinding

The pathfinding algorithm was tested in two problems, firstly is tested on a real map and lastly is used to solve a maze. In both cases, four different algorithms are evaluated and compared: A*, BFS, DFS and GreedyBFS.
To setup this test, I extracted the map of my house from my vacuum robot. The robot has a Lidar sensor that maps the walkable floor each time it cleans. This map is stored as an image in the robot memory, and through the app it can be used to direct the robot in a specific room or spot. To emulate this behaviour, I processed the saved map to have only white and black pixels[3], selected a starting point which is near the base of the robot and an end point in one of the rooms.
The Table 1 shows the results of the tests, while the Figures 2, 3, 4 and 5 show a screenshot of the tests during the run. In the screenshots, the white parts represents the allowed part in which the robot can move, while the black pixels represents walls or otherwise obstacles. During the solving process the progress is shown by coloring red the pixels that has been explored, while the current explored path is shown as a grey line. After a solution is found, its highlighted with a green line.

	Solving time [s]	Iterations [/]	Solution depth [/]
A*	5.49	16974	473
BFS	17.74	32255	473
DFS	21.50	31663	5425
GreedyBFS	0.50	4374	515

Table 1: Results of the pathfinding tests on the map

From the result's table and the screenshots we can see that the fastest algorithm is GreedyBFS, that found a path in half of a second and in 4374 steps. However, it hasn't found the best solution[4]. That was found by A* and BFS, but we can clearly see on the screenshots that A* has explored almost half of the space in comparison with BFS. This is because BFS is not aware of where the goal is, it searchs in every direction without discrimination, exploring all the space in the lower-right rooms and entering in all the other rooms. On the other end we can see that A* has a more direct approach, it doesn't explore every part of the lower-right rooms, focusing the search in the direction of the goal. The worst algorithm in this situation is clearly DFS as its starts to search in a snake-like pattern disregarding the goal direction. Its worth noting that the search pattern of DFS depends on the order in which the new nodes are generated. In this case

[3] It's worth noting that in this experiment I didn't account for the width of the robot, which will be represented as a single pixel. This however can be done by expanding the black pixels by a $robot_{width}/2$ amount

[4] In this case, the best solution is defined as the shortest path between the start and end points

the nodes are generated in the order: up-right-down-left. So, during the search the upper-right direction is preferred, which is very unfortunate in this case, as the solution is in the lower-left corner.

Figure 1: Map extracted from the robot with the selected staring and ending points

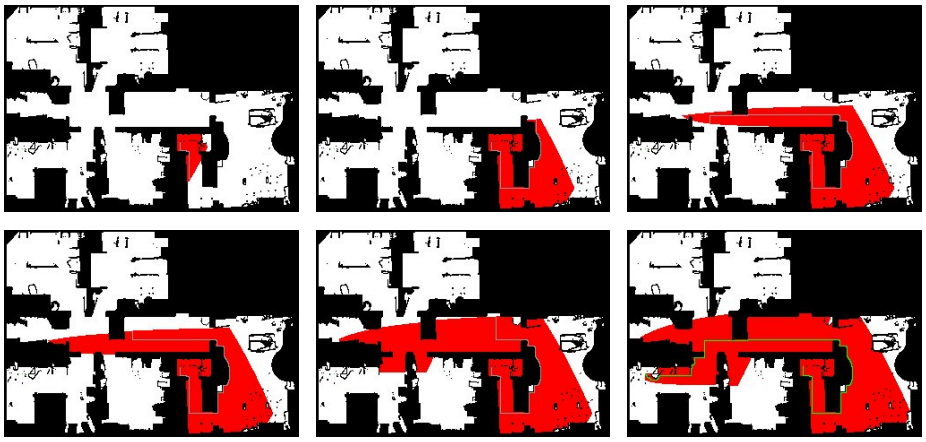


Figure 2: A*

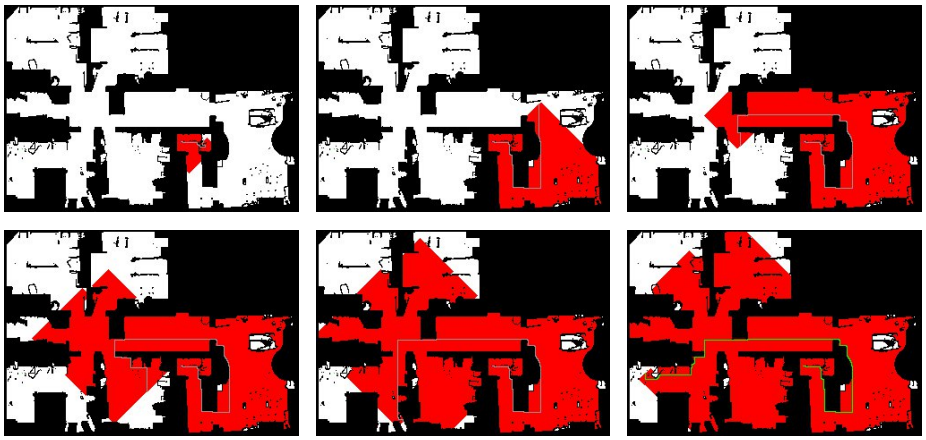


Figure 3: BFS

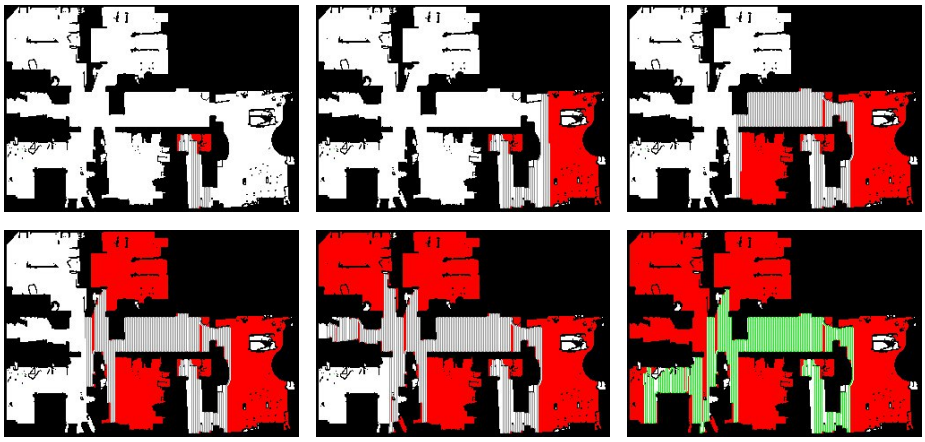


Figure 4: DFS

Figure 5: GreedyBFS

5.1.1 Maze solving

This test is setup is identical with the previous one. Instead of loading a map, with load a maze with the same characteristic. Moreover I have set the starting point in the upper-left corner and the goal in the lower-right corner. Two mazes were choosen to test the algorithm a small one with dimensions 201 x 151 and one big with dimensions 699 x 1079.
The results relative to the small maze are shown in the table 2, while the figures 6, 7, 8 and 9 shows some steps of the solving process for the different algorithms. In this case, the maze generation algorithm ensures that there is only one solution, so the way in which evaluate the different algorithms is based only on the steps taken and the solving time. From the table and the screenshots we can clearly see that the best algorithm is DFS, as it explores only a small

part around the solution. The second best algorithm is GreedyBFS, from the screenshots we can see that its similar to DFS but with more area explored. The worst algorithms in this particular case are A* and BFS, both perform in a very similar way, exploring the majority of the puzzle before finding a solution. To explore if the small scale maze has introduced some biases, I did run also the algorithm on multiple bigger maps, the results of one of these are presented in the table 3. The results are very similar to the ones of the small puzzle, confirming the conclusions expressed before.

	Solving time [s]	Iterations [/]	Solution depth [/]
A*	2.45	12766	5417
BFS	2.61	13160	5417
DFS	0.82	6300	5417
GreedyBFS	1.21	8021	5417

Table 2: 201 x 151

	Solving time [s]	Iterations [/]	Solution depth [/]
A*	282.97	124980	42151
BFS	173.13	126352	42151
DFS	46.76	63010	42151
GreedyBFS	82.39	85834	42151

Table 3: 699 x 1079

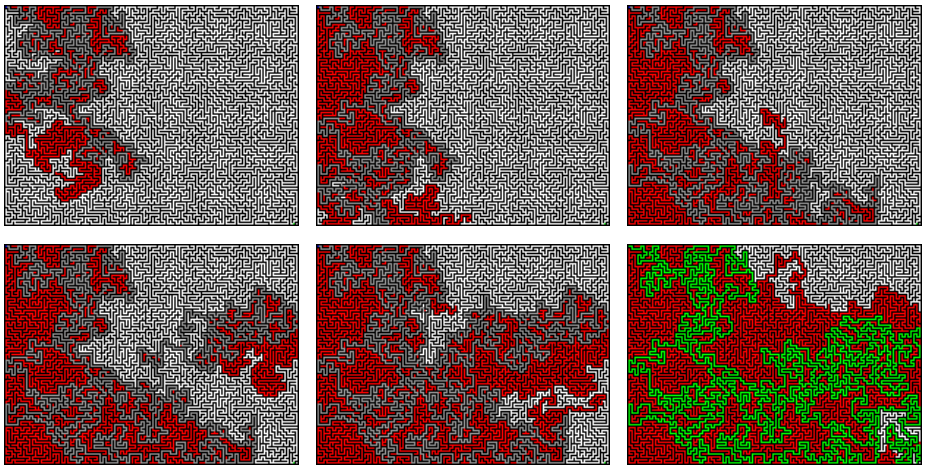


Figure 6: A*

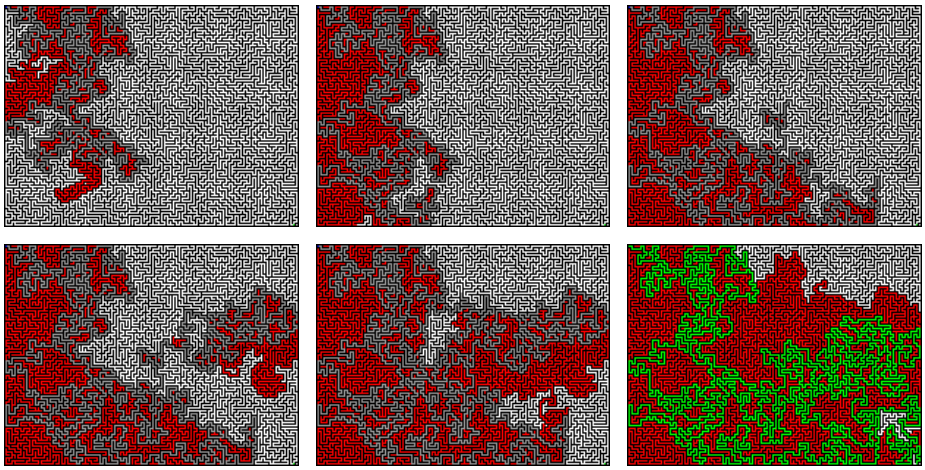


Figure 7: BFS

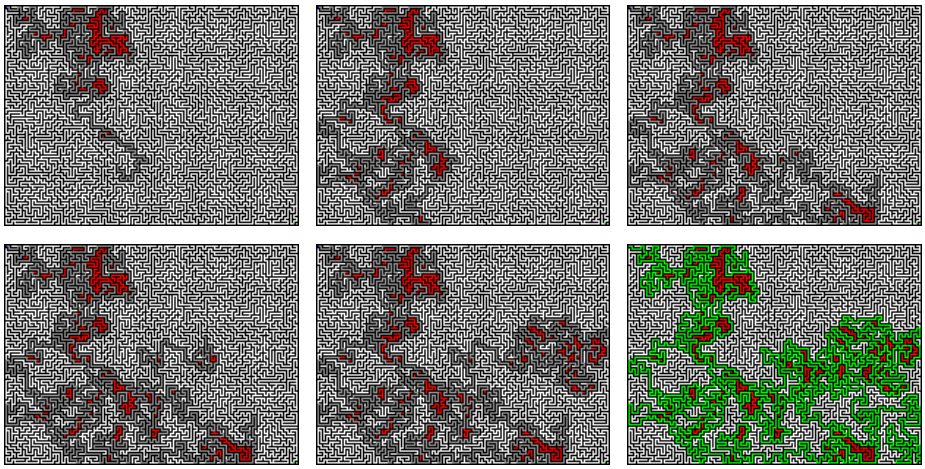


Figure 8: DFS

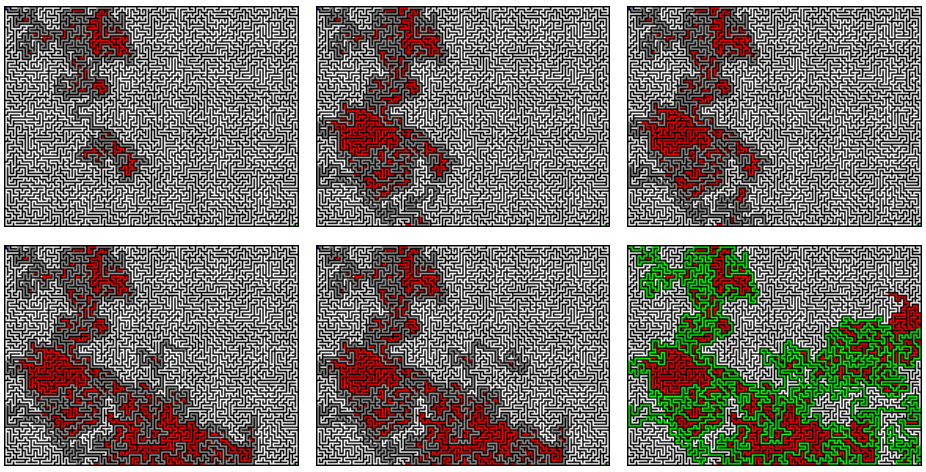
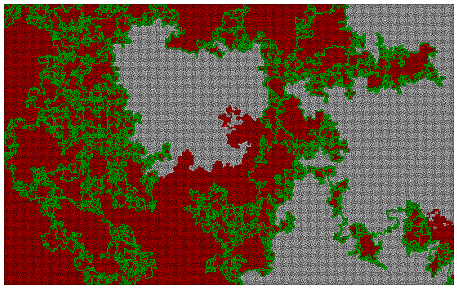
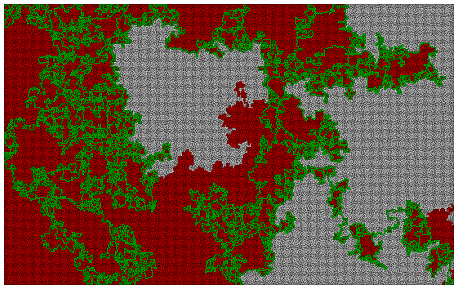


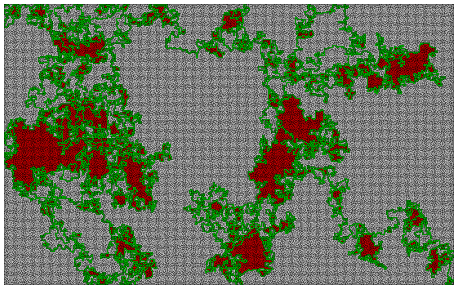
Figure 9: GreedyBFS



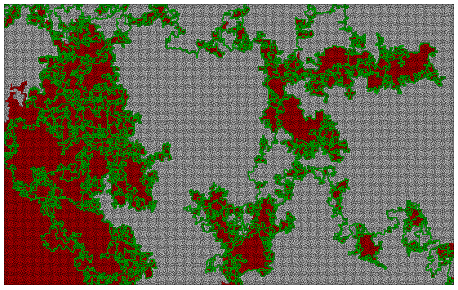
(a) A*



(b) BFS



(c) DFS



(d) GreedyBFS

Figure 10: Puzzle 699 x 1079

5.2 NPuzzle Solver

In this test is evaluated the performance of four algorithm: BFS, DFS[5], A* and GreedyBFS. For the last two algorithms, the heuristic function was defined as the sum of the Manhattan distances from each tile to its goal position.
The test is divided in multiple phases. The first test was to solve a 3x3 puzzle and then a 4x4 puzzle is tackled. This was done as the 3x3 puzzle is much easier than the 4x4 version. In fact, the 3x3 version has a maximum solution depth of 31, meanwhile the 4x4 version has a maximum solution depth of 80.

5.2.1 3x3 Puzzle

This test consists of 400 puzzle, each one tested with the four different algorithms for a total of 1600 runs. The test code was parallelized to run on 16 threads and finished in around 420 seconds[6]. For time constrains, each algorithm was limited on a maximum iteration number of 5000. With this conditions, the solving rate is as follows:

- A*: 98.5%
- BFS: 59%
- DFS: 37.25%
- GreedyBFS 100%

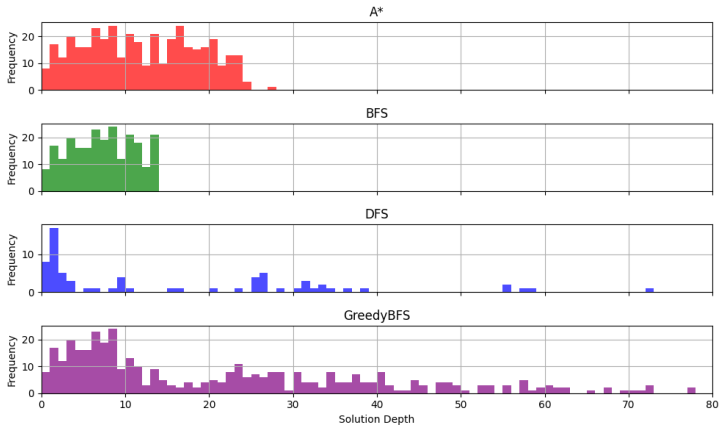
As previously suggested, the uninformed search algorithms have a low success rate in this test condition, meanwhile the informed search algorithms have an almost perfect solving rate.
The figure 11c shows, for each successful run the found solution Depth vs the number of iteration that it took, both in a logaritmic scale. In this picture we can see that the DFS follows a linear trend and thus the solution depth increases exponentially with the number of iterations[7]. We can also see that A* is the best as it finds consistently the best solution in the least number of iterations. In comparison, BFS, even if it finds also always the best solution, the iterations that it take is consistently higher. GreedyBFS is slightly worse than A* as it doesn't find the best solution, but is the fastest one.
The figure 11a shows an histogram of the solutions depth found by the different algorithms. We know that BFS and A* finds always the best solution in this problem, this can be also observed in the graph as both graphs are initially the same, but the BFS's stops at around depth 15. This is because of the limit in iterations, BFS couldn't explore past this depth. On the other hand, DFS solution depth is not consistent as the algorithm does random moves unitl it finds a solution. In the end, GreedyBFS, altrough is the fastest one, the solution

[5] It's worth noting that the Deep First Search algorithm used in this test was not depth-limited

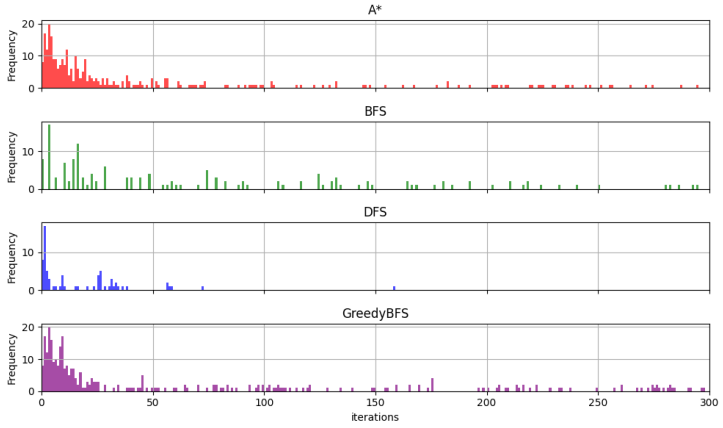
[6] The run time is just an indication as the test conditions were not equal for each test, with different programs running and different power configuration of the computer

[7] This is because the algorithm is not depth-limited and thus it never backtracks

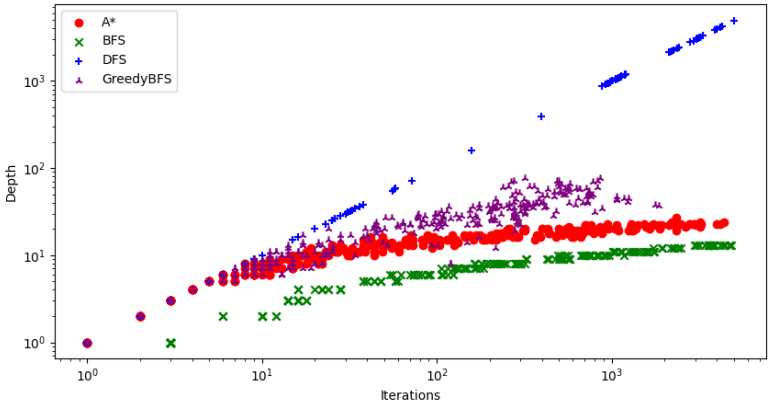
depth is not very consistent, it starts matching A*/BFS but after that it has a slow decay.



(a) Depth



(b) Iterations



(c) Depth vs Iterations

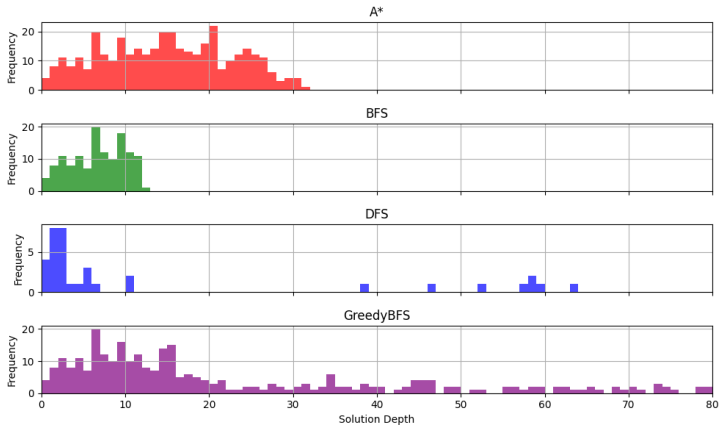
Figure 11: 3x3 Grid

5.2.2 4x4 Puzzle

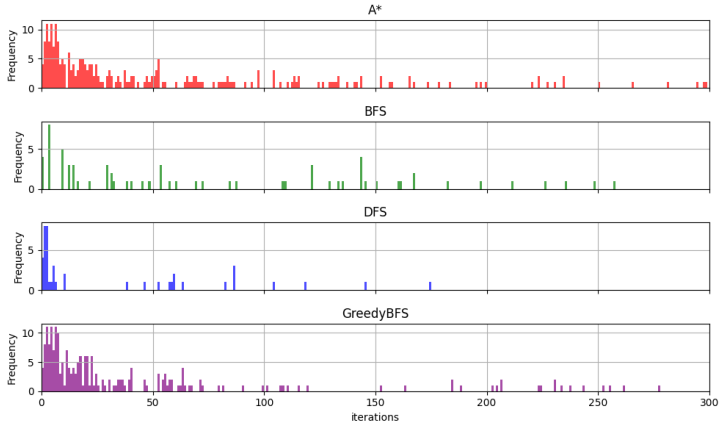
The 4x4 test was run in similar conditions as the previous one. In this case, the iteration limit was set to 10000 cycles. The resulting solving rate is as follows:

- A*: 90.5%
- BFS: 33.25%
- DFS: 12.5%
- GreedyBFS 79%

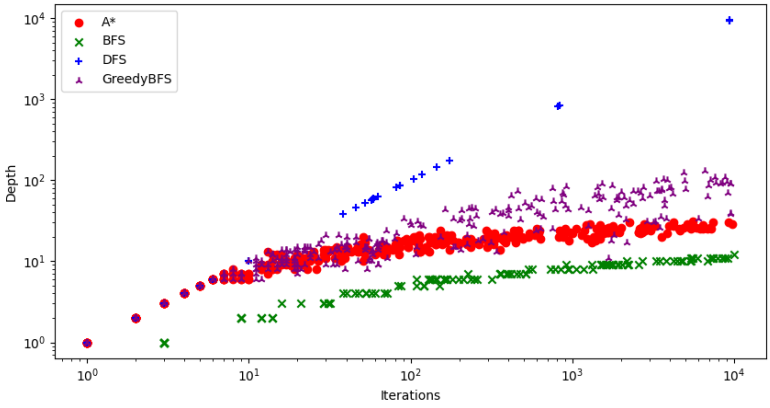
From this result we can infers that probably 10 thousands iteration was not enough for A* to find a solution in every case. However we can note that GreedyBFS is now slightly worse with BFS and DFS in a similar spot as the test before. From the pictures below we can see that the behaviour is almost the same as the previous test, with A* the best algorithm, followed by GreedyBFS which is faster but finds worst solutions. On the other hand, BFS is limited by the iteration limits and finds solutions only up the depth of 32. The worst one, like before, its DFS.



(a) Depth



(b) Iterations



(c) Depth vs Iterations

Figure 12: 3x3 Grid