Search Algorithms Study

Author

Diogo Filipe Pinto Pereira - 31422012
dfpp1e19@soton.ac.uk

December 12, 2019

Contents

1 Approach **1**

2 Evidence **2**
- 2.1 Breadth-First . . . 3
- 2.2 Depth-First . . . 4
- 2.3 Iterative deepening depth-first search . . . 6
- 2.4 A* search . . . 7

3 Scalability **8**

4 Extras and limitations **14**
- 4.1 Blocks . . . 14
- 4.2 Bidirectional Search . . . 14
- 4.3 Greedy Search . . . 16
- 4.4 IDA* Search . . . 17
- 4.5 Graph Search . . . 18
- 4.6 Improved Heuristic . . . 19
- 4.7 Improved Successors function . . . 20
- 4.8 Limitations . . . 21

List of Figures **I**

Nomenclature **II**

References **III**

5 Code **IV**
- 5.1 main.py . . . IV
- 5.2 node.py . . . VII
- 5.3 methods.py . . . XV

1 Approach

The Blocksworld tile puzzle consists of a NxN matrix with an agent and different tiles on it. The tiles with letters are blocks, and the goal is to build a tower with them, in a certain order. To reach the goal, the agent can move in 4 directions: top, bottom, left and right. Each time it moves to the position of a block, it switches places with it, that is, the block goes to the agent's previous position.

To solve this problem there were implemented different search algorithms, that will be explained in detail in sections 2 and 4. In order to represent each node for those searches, it was created a Node class. This class allows for a better organisation of the program and is composed by:

- board - State of the board in a node. Represented as a 1 dimension array in order to save memory.
- agent - Position of the agent. This way, there is no need to find the agent when calculating the successors of a node. Therefore, saving time.
- parent - Node's parent.
- depth - Depth of a node.
- count - Keeps track of the number of nodes visited.
- move - Represents the last move made. This allows to make an improvement in the successors function, so that the successors of a node don't do a move symmetric to the previous one. This will be explained with more detail in section 4.7.

There were implemented several methods for this class. In order to get the successors of a node, there is *get_successors()*. This method randomises the moves so that the agent doesn't keep doing the same move in a search like depth-first. When it finds a valid move, it creates a successor node, whose parent is the current node, and with depth increased by 1 over the parent's one. Such method is then improved, as mentioned above. *heuristic_manhattan()* is another method and calculates the heuristic value of a node based on *manhattan distance* for the heuristic searches. It is improved as well, as explained in section 4.6. Besides that, there are auxiliary methods: *get_position()*, which allows to get the position of the blocks; *build_hash()*, that is used for graph search, which is an extra in section 4.5; *check_solution()* to check if a node is the final state; printing functions to print the path from initial node to the goal.

Node class is the foundation of the search algorithms. In the search algorithms the only special considerations were the use of a priority queue for the heuristic searches and a set or hash maps for graphs searches. The first is because the priority queue is implemented as a heap, which allows for a faster access and insertion of elements, with time complexity of $O(1)$ and $O(log(N))$, respectively. This is much faster than sorting an array every time an element is inserted, which would be $O(Nlog(N))$ for each insertion. The use of a set for graph search is also due to time complexity. Insertion and lookup time are $O(1)$ on average and $O(N)$ in the worst case, which is better than using an array. An hash map is used, instead of a set, only for depth-first with limited depth, to guarantee that it always finds a solution, because it allows for a node that was already found to be visited again if it is at smaller depth.

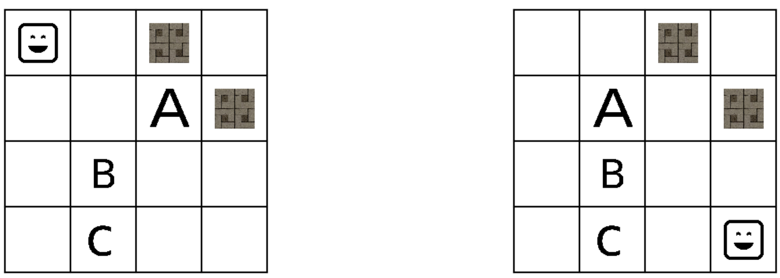
Next section is going to cover each search algorithm and section 4 covers extra algorithms implemented. This work has the objective of exploring the weaknesses and strengths of every method, and how each handles scalability, which will be covered in section 3.

2 Evidence

This section covers the different search algorithms. For each algorithm it briefly explains how it works and it has a proof of correct implementation. The proof is a representation of the states visited by each search, for a given initial state and goal state, which are represented in Figure 1. In this states, the agent is represented at the top left, on the initial and at the bottom right, on the goal. Each block is represented by a letter. There is also the introduction of blocks that can't be moved, that are mentioned in section 4.1. This are block images, that in the initial state can be seen on top and to the right of the A block. The optimal depth for this states is 3, that is, it should only take 3 moves to get to the initial state the goal state.

The first 3 searches are uninformed searches, and the last one is an informed search. This means that the first three strategies have no additional information about states, besides the ones provided in the problem definition [1], whereas the latter has more information about the board, which is given by heuristics.

In the proofs of each search, on the top of each node, there are going to be different pieces of information about it: "N" - number of the nodes; "P" - number of the parent node; "D" - depth, and "H" - heuristic value, used for heuristic searches.



(a) Initial State (b) Goal State

Figure 1: Initial State and Goal State for the searches proof

2.1 Breadth-First

Breadth-first search analyses the tree per level, that is, it only analyses the nodes in a level k after analysing all the nodes from depth k-1. This means that it is going to visit the nodes in order of expansion, and such is accomplished by sorting the nodes in a FIFO queue, so that the nodes are visited in the order they were put in.

In figure 2 there is a demonstration of the algorithm working. It starts by visiting the root node, which has no parent and depth 0. It then visits nodes 2 and 3, which are the successors of the root node that were obtained by moving the agent down and to the right, respectively, and both have depth 1. After this, it visits the successors of node 2, followed by the successors of node 3, that are going to have depth 2, and so on. It finds a solution at node 11 with depth 3, and the path to solution can be found by progressively following the parents of each node starting from the solution node. In this case is going to be: 1-2-5-11, as it can be seen in Figure 3.

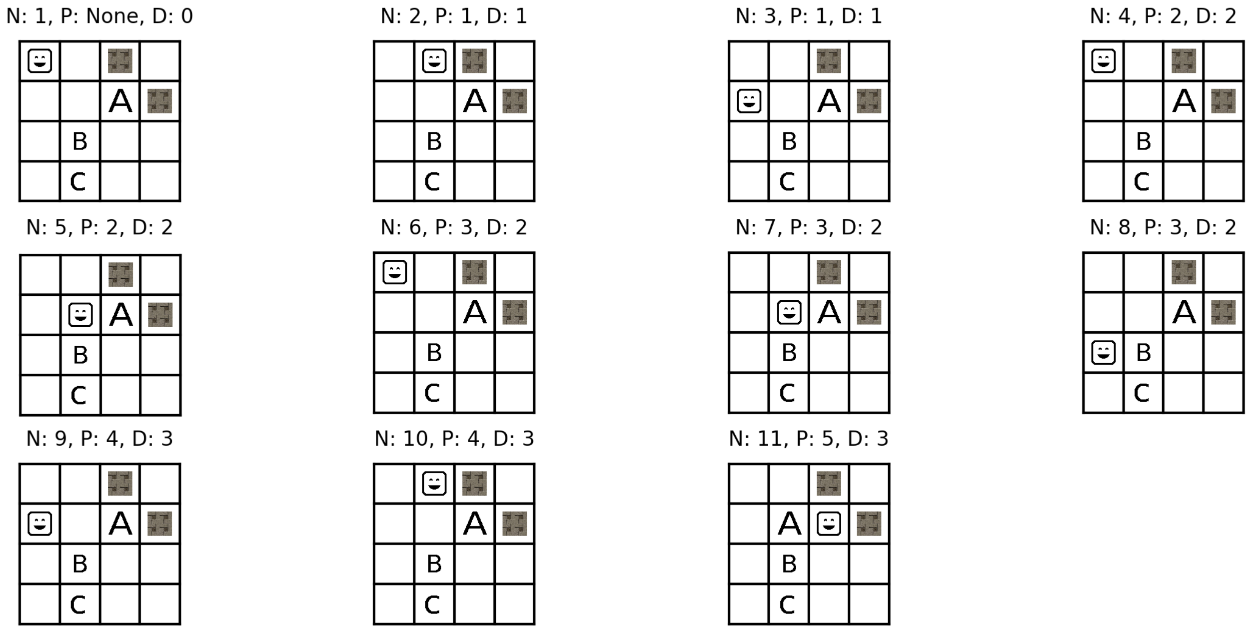


Figure 2: Visited nodes during breadth-first search

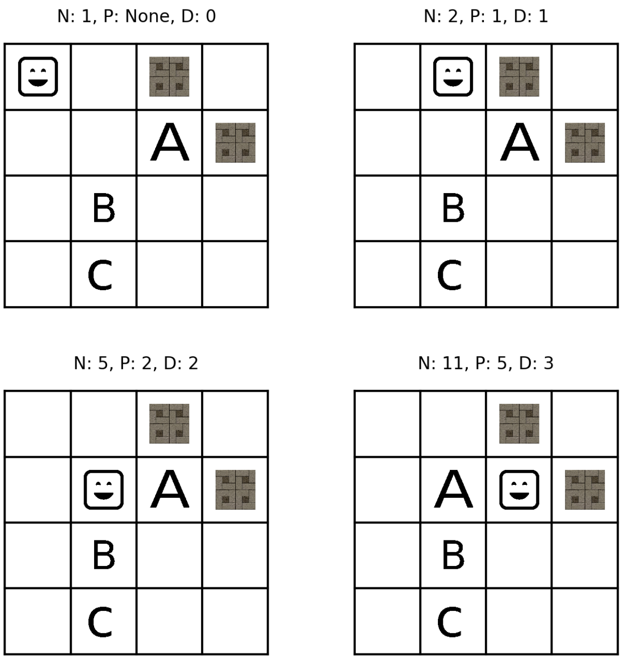


Figure 3: Solution path for breadth-first search

2.2 Depth-First

Depth-first algorithm expands a branch of the tree in depth, which is accomplished with a LIFO queue. The search on that branch stops when it finds a solution or when it can't expand any further, either because there are no possible moves or because it reaches the limit depth. If it doesn't find in that branch then it backtracks, that is, for the last visited node, it goes to his parent and expands in depth other successors that haven't been visited. Usually a limit depth is imposed in this search so that it doesn't end up in an infinite-path problem.

In Figure 4 there is a running example of this search for limit depth 7. After visiting node 1 (root node), visits one of its successors, followed by one of the successors of the second node and so on. When it reaches the depth limit, in this case node 8, because it is not a solution it has to keep searching, so it searches the next successors of its parent. Since none of the successors is the goal state, is has to backtrack again to find the next node which hasn't been visited, which is one of the successors of node 6. It keeps doing this until it finds a solution, which is found on the 32^{th} node, at depth 7. By backtracking until the root node, the solution path obtained is the one shown in Figure 5: 1-2-3-4-5-28-29-32.

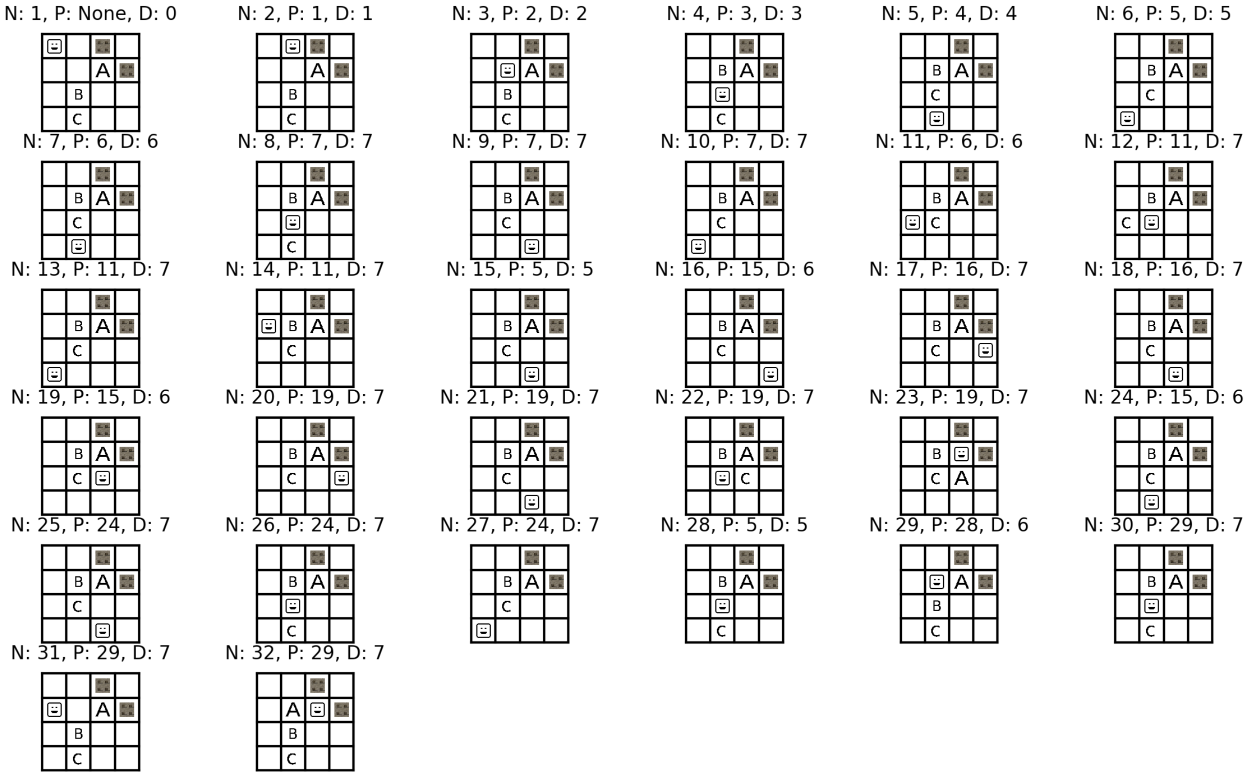


Figure 4: Visited nodes during depth-first search

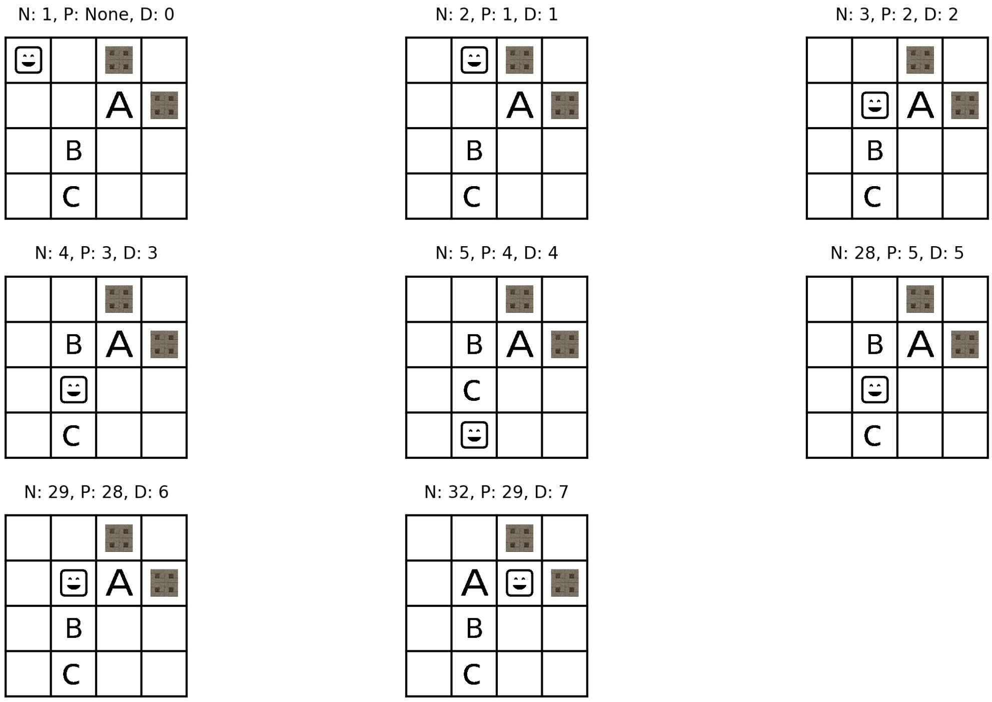


Figure 5: Solution path for depth-first search

UNIVERSITY OF Southampton
School of Electronics and Computer Science

2.3 Iterative deepening depth-first search

This search consists in a depth-first search with an incremental depth limit. It starts with limit depth 0, then 1, and so on, until it finds a solution for a depth limit. When it finishes the search for a depth limit, it discards all the nodes generated in that search [2]. It can be thought of a mix between breadth-first and depth-first search, because it visits the nodes in the same order as depth-first search for each depth, however, the cumulative order in which nodes are first visited is similar to breadth-first [3]. As it is a mix of the two strategies, it doesn't suffer any of the drawbacks of depth-first or breadth-first [2].

Figure 7 shows the order in which the nodes are visited in this search. As it can be seen, it starts by visiting the root node at depth 0, and here ends the search for depth limit 0. Then it starts the search for depth limit 1, and it searches the only two descendants of the root node, and since none of them is the solution, it starts a new depth-first search for depth 2. Again, none of the nodes gets to the goal state, so it has to start another depth-first search for depth limit 3, where finally it finds a solution. This generates the solution path shown on Figure 7.

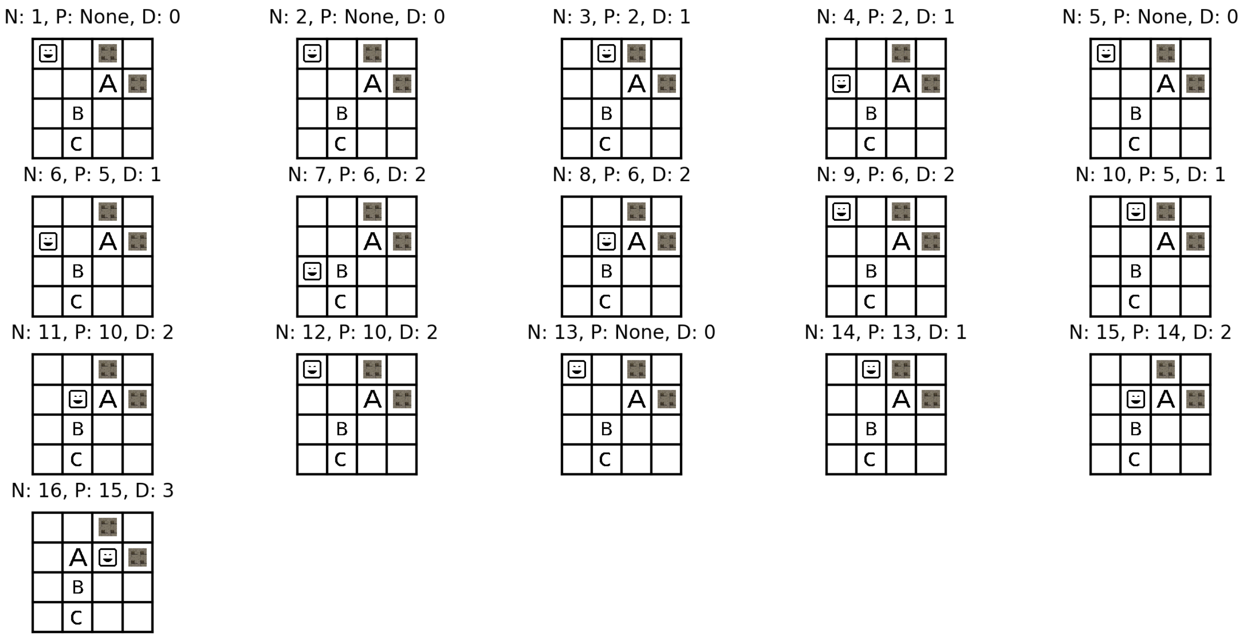


Figure 6: Visited nodes during iterative deepening depth-first

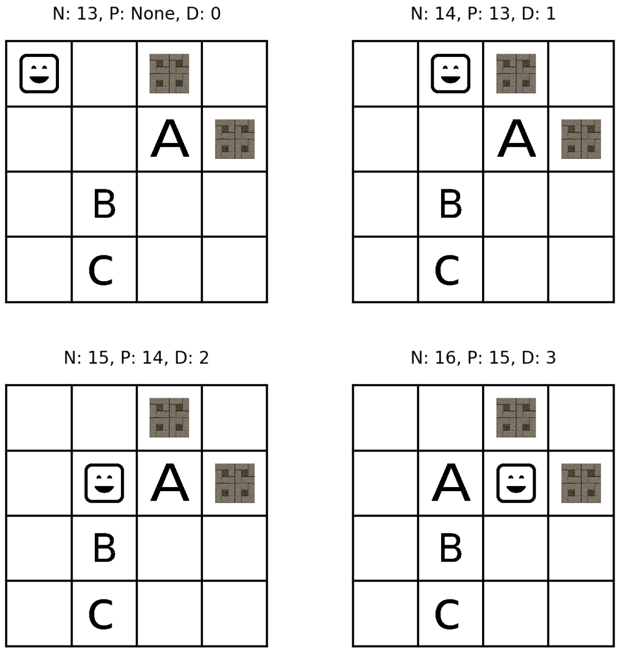


Figure 7: Solution path for iterative deepening depth-first

2.4 A* search

A* search consists of a heuristic search, and it should only be used when there is extra information about the problem. This means that it doesn't just remove the nodes blindly, but it also takes into account another information about the node. It is a combination of greedy search, which is explained more in depth in section 4.3, with uniform cost search. In each iteration it is going to choose the node that is closer to the start node, but at the same time is also closer to the goal state. For each node it calculates the cost to reach that node (g(n): for uniform path cost is equivalent to depth of the node), and the cost to get to the solution (h(n)). So f(n) = g(n) + h(n).

In Figure 8 there is an example of the sequence of visited nodes when applying this search. The value of h(n)+g(n) is shown on the top of each node. The h(n) used in this case is *manhattan distance*, however in section 4.6 it's described an improved heuristic, given that *manhattan distance* is not very efficient.

Starting by analysing the root node, it is going to value 1 because it is at depth 0 and the only misplaced block is 'A', whose *manhattan distance* is 1. The descendants (nodes 1 and 2) are going to have the same h(n) value, however, f(n) is going to have value 2 due to the increase in depth. Because they have the same value, the order in which they are going to leave the priority queue is random. Then, their successors are going to be expanded and are going to have heuristic value 3, as 'A' is still misplaced and they are at depth 2. Node 4 is going to generate a descendant which is the goal state and has value 3 because it is at depth 3 and h(n) is 0, since all blocks are in the correct position. The path from the initial state to goal is shown in Figure 9.

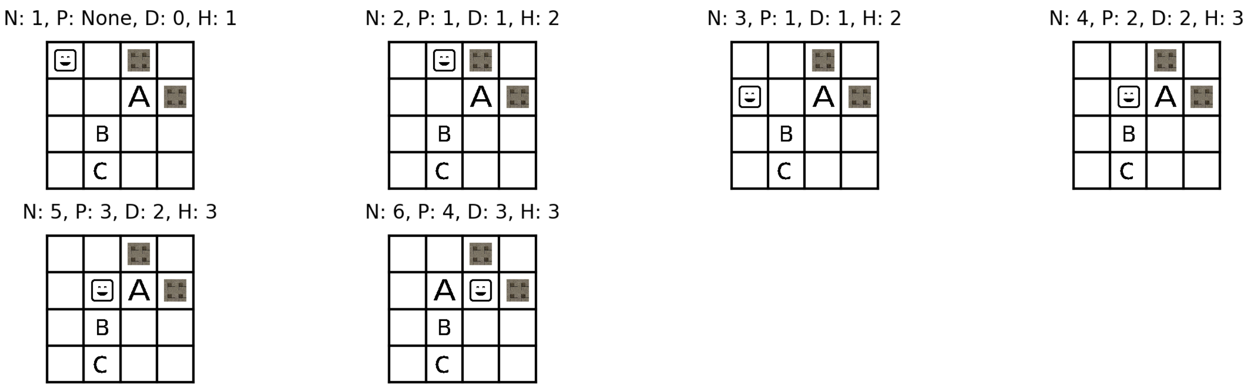


Figure 8: A* visited nodes during search

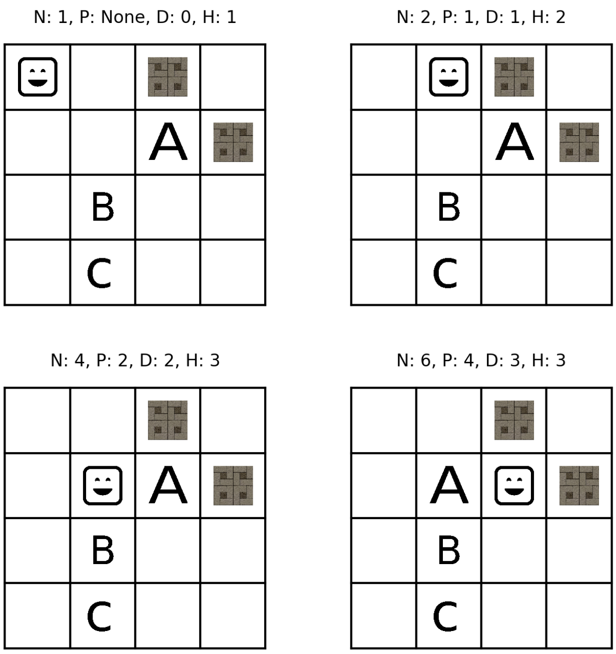


Figure 9: Solution path for A* search

3 Scalability

In order to study the scalability of each search, the problem difficulty was controlled by changing the layout of the initial state. Because the order of expansion of the nodes in the successors function is random, each search was ran 10 times for each depth. An algorithm is considered to fail if it doesn't find a solution in 15 minutes.

Figure 10 and 11 show how the number of nodes expanded changes with the difficulty of the problem. As the range of values is big, in Figure 12 are shown the nodes expanded in logarithmic 10 scale, which helps to make a better comparison. This number of nodes expanded is going to be dependent primarily on the **branching factor** b and the **solution**

depth d [4]. Figure 13 represents the depth at which each search found a solution compared to the optimal depth, and Figure 14 shows the same but for no limit depth-first. Figures 15 and 16 show the memory used by each search.

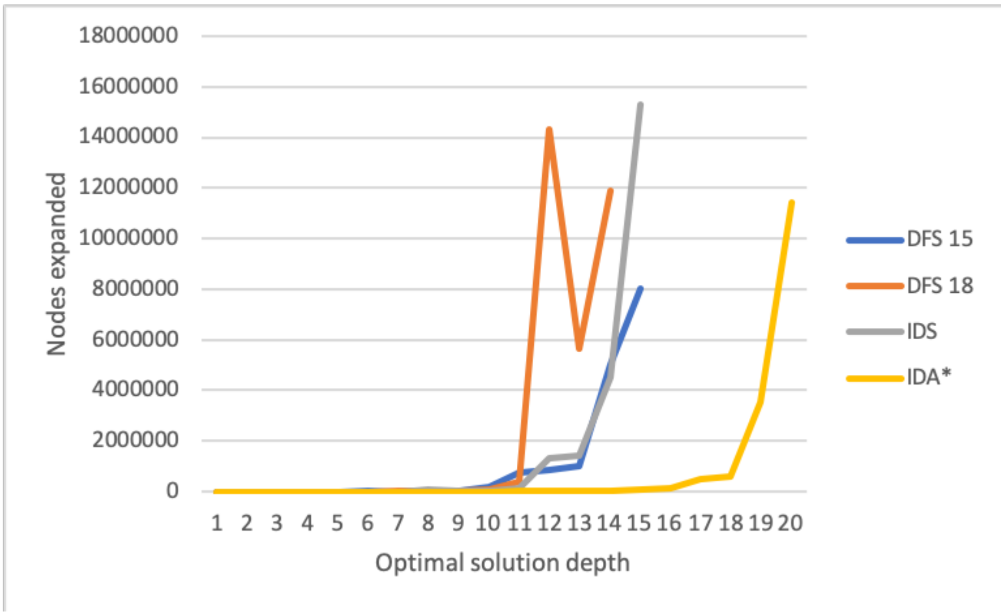


Figure 10: Number of nodes expanded for depth-first limited depth, and iterative-deepening algorithms as a function of the problem depth

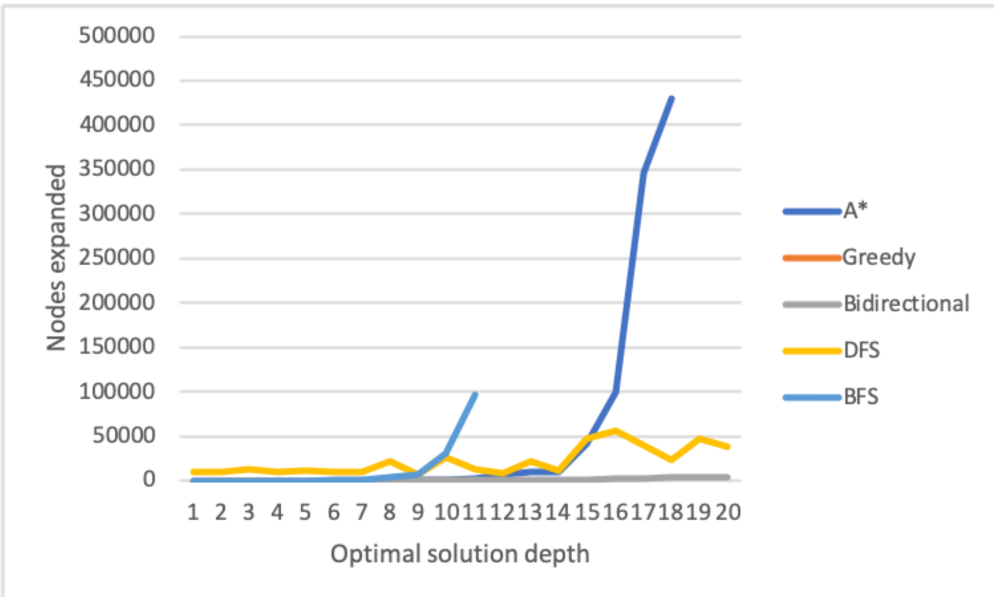


Figure 11: Number of nodes expanded for A*, Greedy, bidirectional, depth-first no limit and breadth-first searches as a function of the problem depth

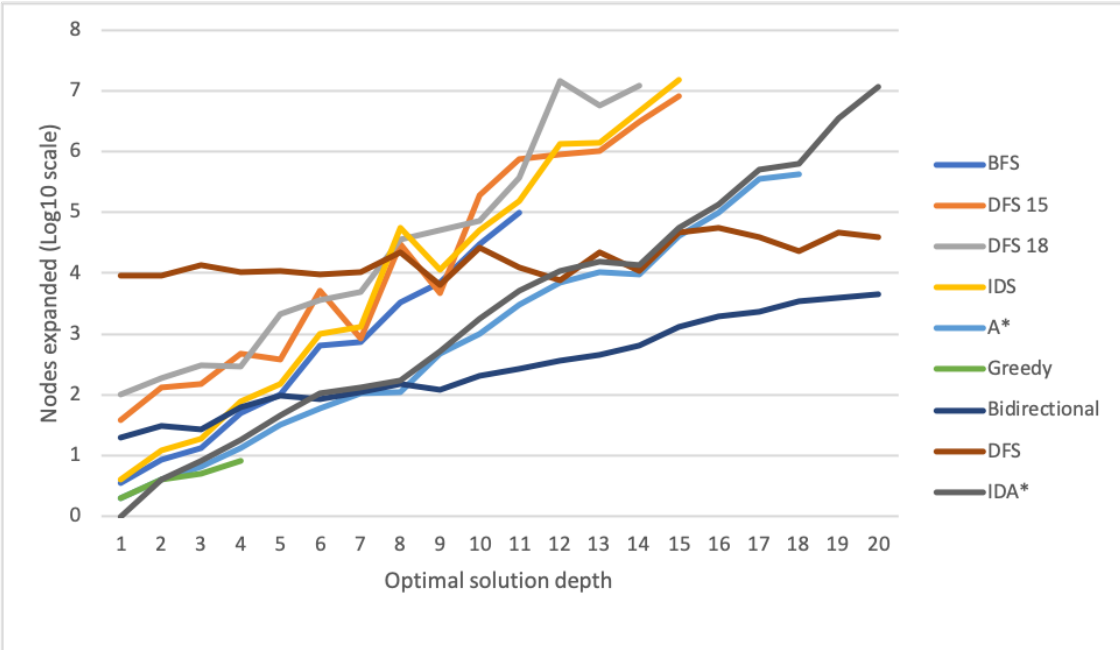


Figure 12: Number of nodes expanded for every search in log_{10} scale as a function of the problem depth

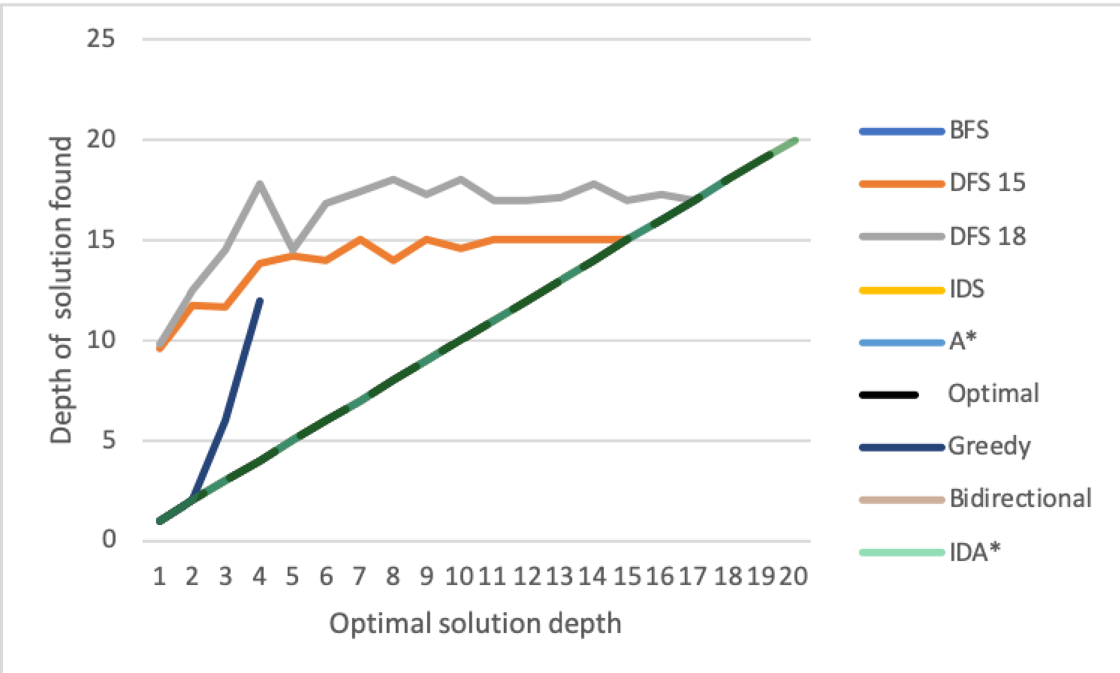


Figure 13: Depth of the solution found for each search as a function of the problem depth

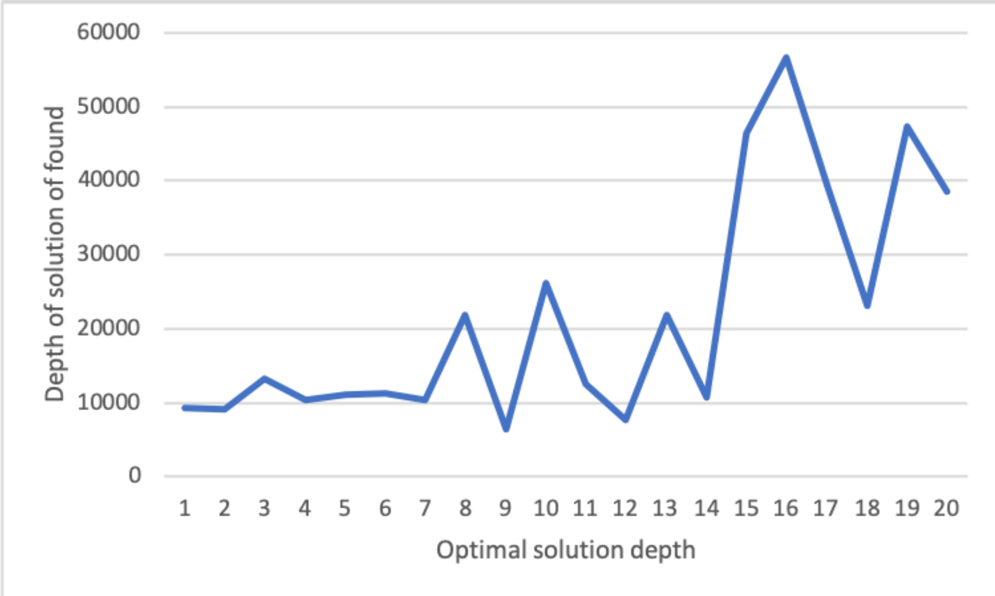


Figure 14: Depth of the solution found for DFS with no limit as a function of the problem depth

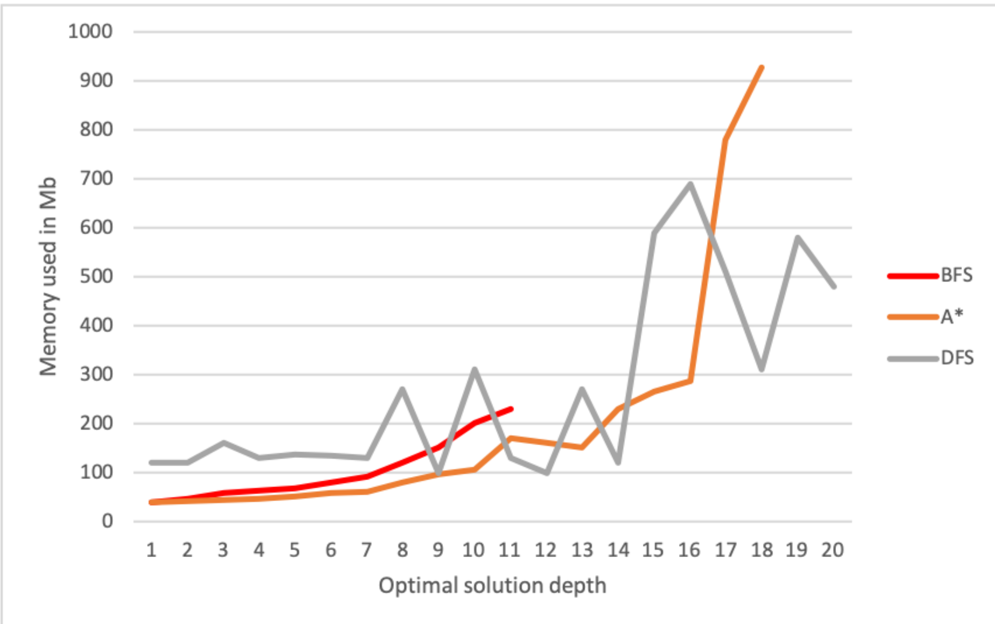


Figure 15: Memory used for breadth-search, depth-first without limit and A* as a function of the problem depth

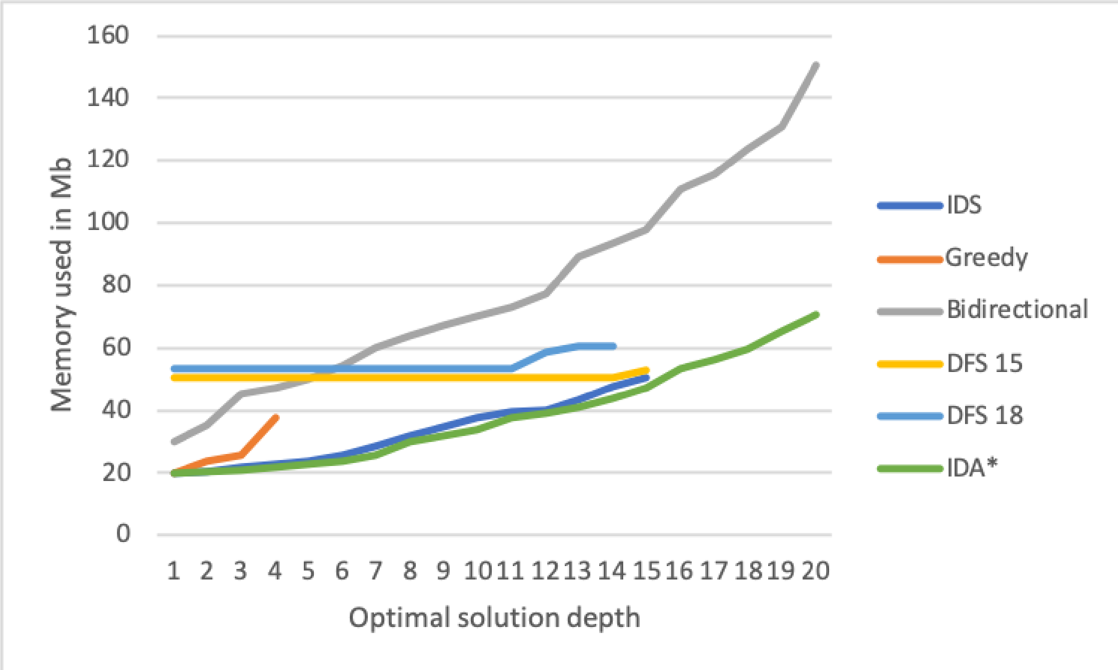


Figure 16: Memory used for depth-first limit 15 and 18, iterative-deepening and bidirectional search as a function of the problem depth

Analysing the graphs above, some conclusions can be taken about the algorithms. **Breadth-first**, although in theory is **complete** [1], that is, always finds a solution, due to time constraints it wasn't able to find solution for depth bigger than 11. For lower depths it is better than depth-first, however, the branching factor starts to be a big factor for higher depths. On the other hand, because the cost per step is uniform, it always finds the optimal solution, so it is **optimal**. In terms of memory it was the one of the worst uninformed algorithms, because it has to store every node in the tree, which consists in a space complexity of $O(b^d)$ [5].

Depth-first was tested two different ways: with and without depth limit. For depth-limit, two depth limits were tested: 15 and 18.

With no depth limit, the algorithm will search through one single path in depth. Even though the number of nodes is relatively constant, it can be a problem in terms of memory, depending on in which depth the solution is found, since the space complexity is given by $(O(bm))$ [1], where m is the depth of the solution found. As it explores only one path down the tree it has to store every node in it, and if the depth at which the solution is found is too big, the memory usage can approximate to the one on breadth-first. On the other hand, the depth limit algorithms guarantee almost constant memory usage, given that only stores the path until the depth limit and some successors of nodes in the path $(O(bl)$, where l is the depth-limit [1]). However, non-limited depth is always able to find a solution in this case. Such happens because the successive actions are chosen randomly, which helps not to be stuck in an infinity tree. Another reason is that in this problem every move is reversible, so if one wrong move is made it can be reverted in posterior moves. This means that it will always be able to find a solution down that path, but that's not possible for every search space, meaning that non-limit depth-first search is generally impractical.

To solver this problem, a depth limit is imposed. The ideal depth limit would be the solution depth [2] yet, it is not always possible to know the real depth of the solution. To show how the depth-limit affects performance, two limit-depths were imposed. For depth 15, the number of nodes expanded were, in general, smaller than for depth 18, which is due to the fact that the number of nodes to search is bigger. However, for depths bigger than 15, the first one won't be able to find any more solutions, which is not the case for depth 18. So, there is always a trade-off between performance and completeness. This, and the fact that it can get stuck in infinity loops for non-limited depth, means that depth-first is **not complete**.

In terms of optimality, both versions are **not optimal**, being the one with no depth limit the worst.

Iterative-deepening depth-first search iterates through all the depths, which means that for bigger depths the number of nodes expanded can be slightly bigger than DFS, but it always finds the optimal solution. In terms of memory performs better than BFS, because it stores nodes like depth-first search does. It is basically a mix between the best part of BFS, which is **optimality** and **completeness**, and the best part of DFS, which is memory. Its optimality is due to the fact that it never expands a node until the shallowest ones have been expanded, so it always grants the shortest path [6].

A*, because it is an informed search, it outperforms every uninformed search of the above. For lower depths the number of nodes expanded is very small, however, it increases slightly for bigger depths, and a big factor for that is that the heuristic is not very good. Because it stores nodes the same way as breadth-first, for big depths, memory starts to be a problem, as it can be seen in Figure 15. Nonetheless, it always grants an **optimal** solution, which is due to the cost added to the heuristic, meaning that if two nodes are the solution, then the shallowest one is always chosen, as it has a lower cost to get to it. This also means that it is **complete**, because it will never get stuck in infinite loops.

To solve A* memory problem, **iterative-deepening A*** was implemented, which has linear space complexity [6]. Even thought the number of nodes expanded can be bigger than A*, because of the overhead of restarting the search for every threshold, this search is preferred over A*. It also conserves the **optimality** and **completeness** of A*. Furthermore, IDA* was able to find solution for every depth in less than 15 minutes, even though it expands more nodes, and that is due to the fact that it doesn't have to order the nodes in a priority queue.

Bidirectional search, overall, was the best search. In terms of nodes expanded was the best searches for big depths, proving that $O(b^{d/2})$ is much better than $O(b^d)$. As it will be explained in section 4.2, bidirectional search was implemented with BFS search from both sides, which makes it **complete**, because it is granted that the paths will intersect. As the path cost is 1, this grants **optimal** solutions. In terms of memory it is worst than limited depth DFS, because it has to store each breadth-first search, however, because the factor d (depth of the solution) is cut in half for each breadth-first search, the number of nodes stored is significantly less to BFS. When the heuristic gets improved, A* can match and outperform this results, as it will be seen in section 4.6.

Greedy's performance was extremely bad. This is due to the fact that *manhattan distance* heuristic isn't very good for this problem. For lower depths, when it was able to find a solution, was the best in terms of nodes expanded. However, when the depth got higher, it would get stuck in a loop and not find a solution, which means that it is **not complete**. Besides that, greedy is **not optimal**. In section 4.6 it's going to be discussed a new heuristic, that makes greedy more viable, and it will actually be the case that greedy is the best search in terms of number of nodes expanded when it finds a solution.

Concluding, the best overall search was bidirectional search, but it also must be taken into account that the heuristic for A* was not very efficient, and the time complexity in A* is an exponential function of the error in the heuristic function [7]. IDA* had the same problem as A* in term of heuristic, but it is an improvement in memory. The greedy search has proven to be the fastest algorithm when it can find a solution (for lower depths), however, is not optimal. In terms of breadth-first, it can only be used for easy search spaces, as it can be really time and memory consuming for harder problems, since this have deeper search spaces. Depth-first can be used if the depth of the pretended solution is known, otherwise, either a solution is not found or the execution time is very big and it leads to a non optimal solution. Iterative-deepening grants an optimal solution and saves memory space, but it can also be very time consuming, especially because of the overhead caused by the repetitions. So, the chosen search algorithm depends on what the primary goal is: optimality or memory. It also must be taken into account if the goal node can be backtracked and if it is completely defined, or if it is possible to find an admissible heuristic or not.

4 Extras and limitations

4.1 Blocks

To help increase the difficulty of the problem immovable blocks were added. This means that there are some tiles to where the agent can't move, limiting the moves the agent is able to do. Although sometimes it helps because the possible moves for some positions is smaller, other times it means that the agent has to take a longer path to the solution.

4.2 Bidirectional Search

Because in this problem it is possible to have a predecessor function, bidirectional search was implemented. In this search, there are two searches occurring simultaneously, and a solution is found when they intercept each other. In both sides the search used is breadth-first search.

Figure 17 shows the visited states during a search. For each iteration, two states are printed, being the first one the state in the search down, and the second one the one in the search up. The search down starts with the root node (node 1) and the search up starts with the goal node (node 2). Both are going to have two successors, which are going to be visited in the next two iterations. This breadth-first search keeps going until a node visited in the search down was already found in the search up or vice-versa. This happens with node 23, that was already visited in search up (node 18), so the algorithm stops here.

In Figure 18 it is represented the solution path. It starts by backtracking the search down nodes, and prints all of them. For the search up, it ignores the last node of the search, as it is the same as the last node of the search down, so it starts by the penultimate node, which in this case is the node 10, and backtracks from there. The depth for these nodes is increased accordingly. Although the solution has depth 6, the real depth of the solution is 3, as in this problem it only matters the location of the blocks, not the location of the agent. This means that for this search the only depth taken into account is the depth until a node with all the blocks in place, except the agent, is found. In this case, that is node 23. Such calculations are made by the function *print_ solution()* in the *methods.py* file.

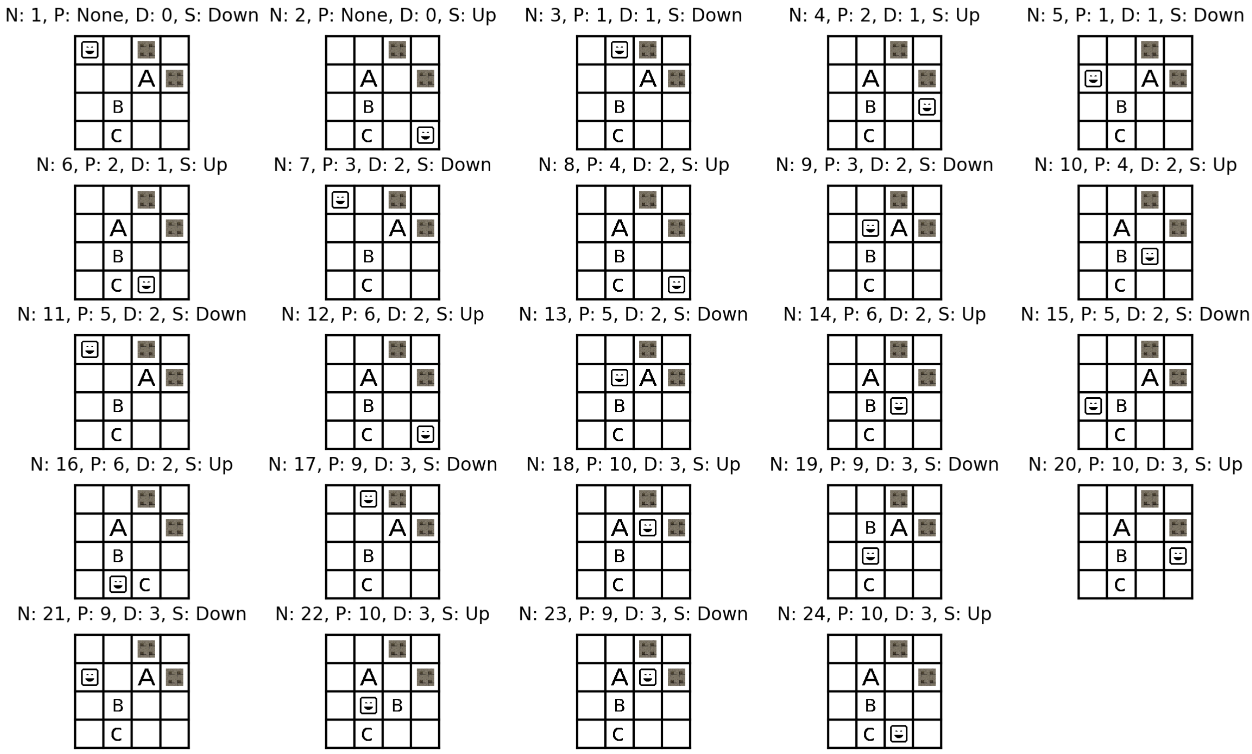


Figure 17: Bidirectional search visited nodes

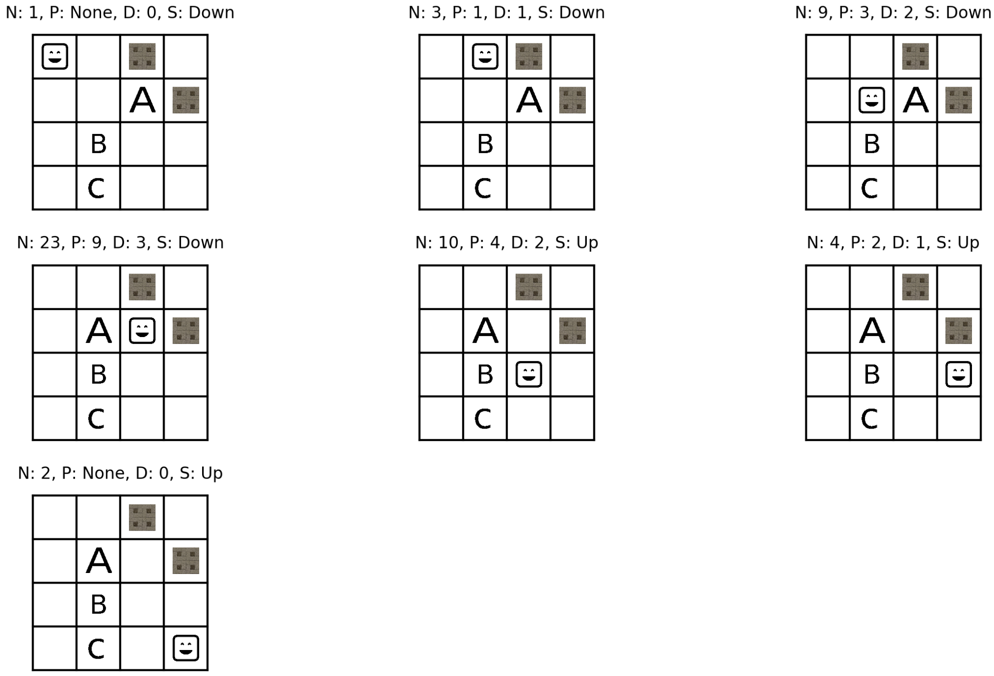


Figure 18: Solution path for Bidirectional search

4.3 Greedy Search

Greedy Search is another informed search. The main difference compared to A* search is the calculation of the function value for each node. Unlike A*, greedy search only focuses on the heuristic value of the node, that is, f(n) = h(n), so it is a special case of *best-first search*.

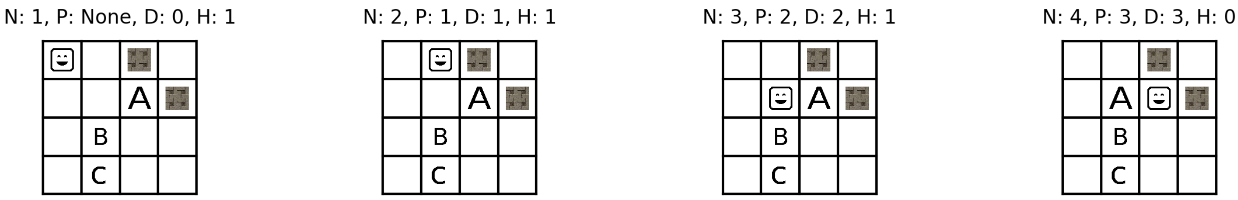


Figure 19: Greedy search visited nodes

The search path is shown in the figure above and the heuristic used for this case is *manhattan distance*. The start node has evaluation value 1, because the only misplaced block is 'A'. This node is going to generate two successors, whose evaluation value is still 1, as 'A' keeps out of place, so they are taken out of the priority queue in a random way. Node 2 gets out of the priority queue, and generates two more nodes, being the next expanded, node 3 that is going to generate the solution node (Node 4). The visited nodes already correspond to the solution path.

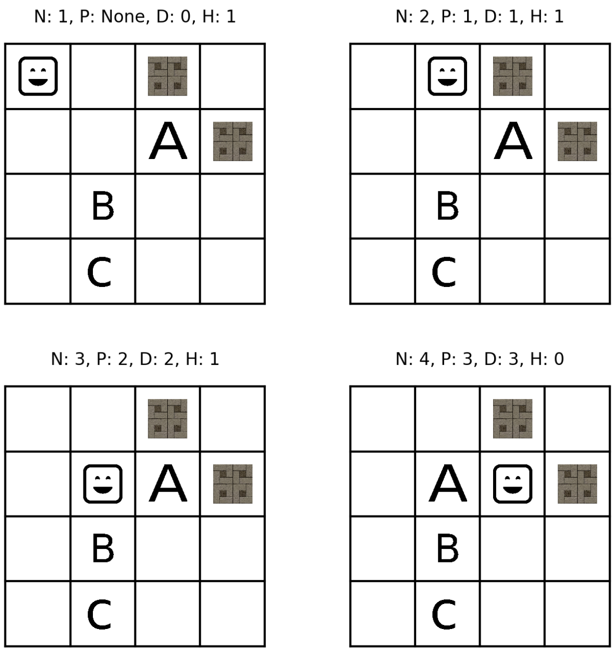


Figure 20: Solution path for Greedy search

4.4 IDA* Search

Iterative-deepening A* is a variant of A* that uses iterative-deepening. However, this version of iterative-deepening is guided and based on a heuristic threshold rather than on depth. The search starts with a limit heuristic, correspondent to the heuristic value of the root node [8]. The nodes with heuristic value bigger than that are cut-off, and if no solution is found, the new heuristic limit is the minimum of the pruned ones during the previous iteration [6]. It keeps increasing until a solution is found. Because it is based on depth-first, the space complexity of this search algorithm is going to be linear.

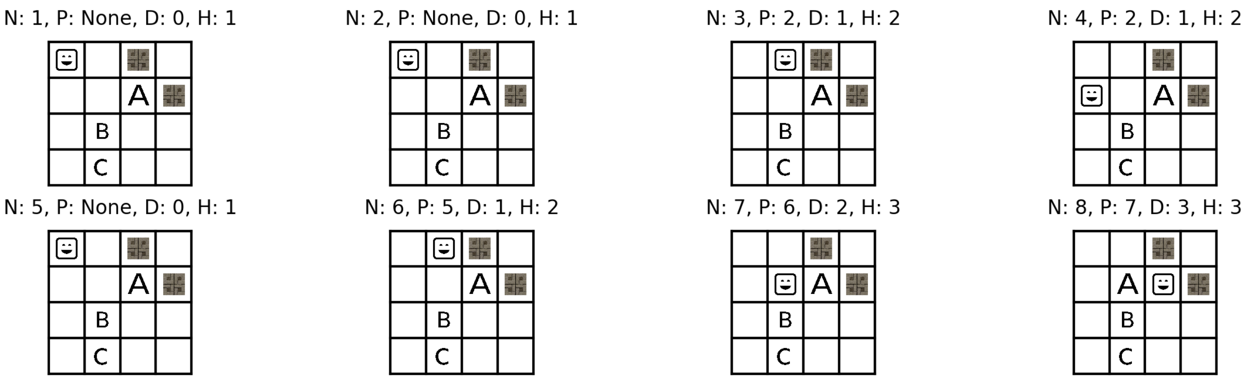


Figure 21: Iterative-deepening A* visited nodes

The search path is shown in Figure 21 with *manhattan distance* as heuristic. The start node has heuristic value 1 because the only misplaced tile is 1 move away from it's position.

Because both generated nodes are going to have heuristic value 2, the search ends there, and starts again with heuristic limit 2. For the new iteration, the successors of the root node are expanded, however, its successors are not because they have heuristic value 3, and the limit is 2. In the final iteration, that starts with node 5, a solution is found in node 13, which has heuristic value 3. By backtracking the parent nodes, the solution shown in figure 22 is obtained.

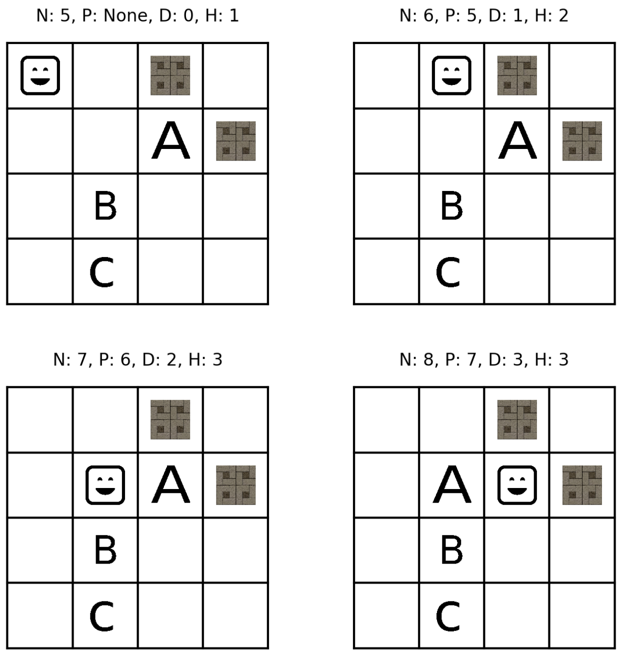


Figure 22: Solution path for iterative-deepening A* search

4.5 Graph Search

Graph search was implemented for BFS, A* and DFS. Such was accomplished with the creation of a set, where the expanded nodes would be in, and for each node that was going to be expanded it would check if it was already expanded before, and it would only expand it otherwise. For depth limit depth-first search it was used a hash-map, instead of set, where a node would still be expanded if it was found at a shallowest depth than before. This ensures that always finds a solution.

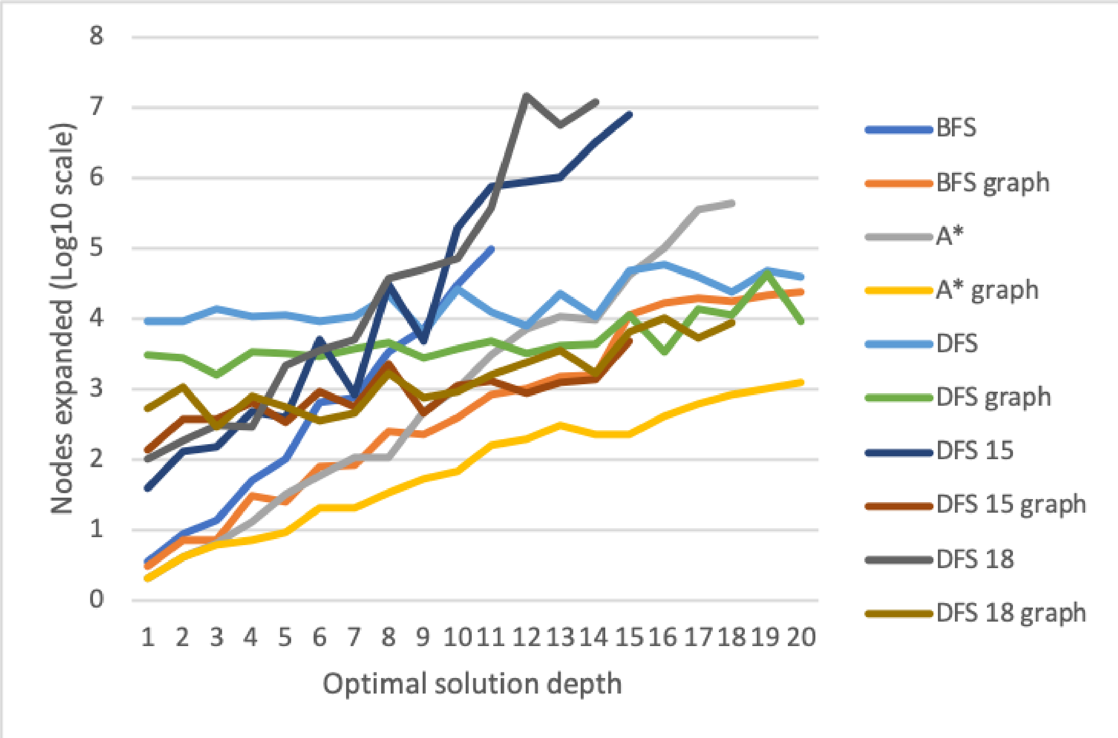


Figure 23: Number of nodes expanded for graph search as a function of the problem depth

In Figure 23, it can be seen that every search significantly improved with graph search. It is the case that even breadth-first graph search is better than normal A*, but this comes at a cost. In order to store the nodes expanded, more memory is needed and for bigger depths that can start to be a problem. This becomes impractical for large problems, so tree search is used for most big problems [4]. Also, for depth-first it wouldn't make sense to store the nodes expanded, as it would remove the only advantage that it has over the other searches, which is memory.

4.6 Improved Heuristic

The *manhattan distance* has proven to be slightly inefficient for this problem, as it can be seen in the results of the greedy search. To improve this results an adaptation of *manhattan distance* was created. Besides taking into account the *manhattan distance* of the misplaced blocks, it also considers the position of the agent. Such is achieved by adding the *manhattan distance* from the agent to the further misplaced block. This is one admissible heuristic because it doesn't overestimate the cost to reach goal. Besides the need to move the blocks a certain number of squares to their goal place, the agent also has to move itself towards the misplaced blocks, so it is never an overestimation of the cost.

Figure 14 shows that the improved heuristic has much better results. Greedy search can now find a solution until depth 10, and A* visits much less nodes. However, it's not totally efficient, because greedy is still getting stuck in loops. This can have two interpretations, it can be either due to the fact that the position of the agent in the goal state doesn't matter, or because a better heuristics can be found. Another thing to notice is the fact that, has said in section 3, greedy search is really the best search in terms of nodes expanded when it finds a solution.

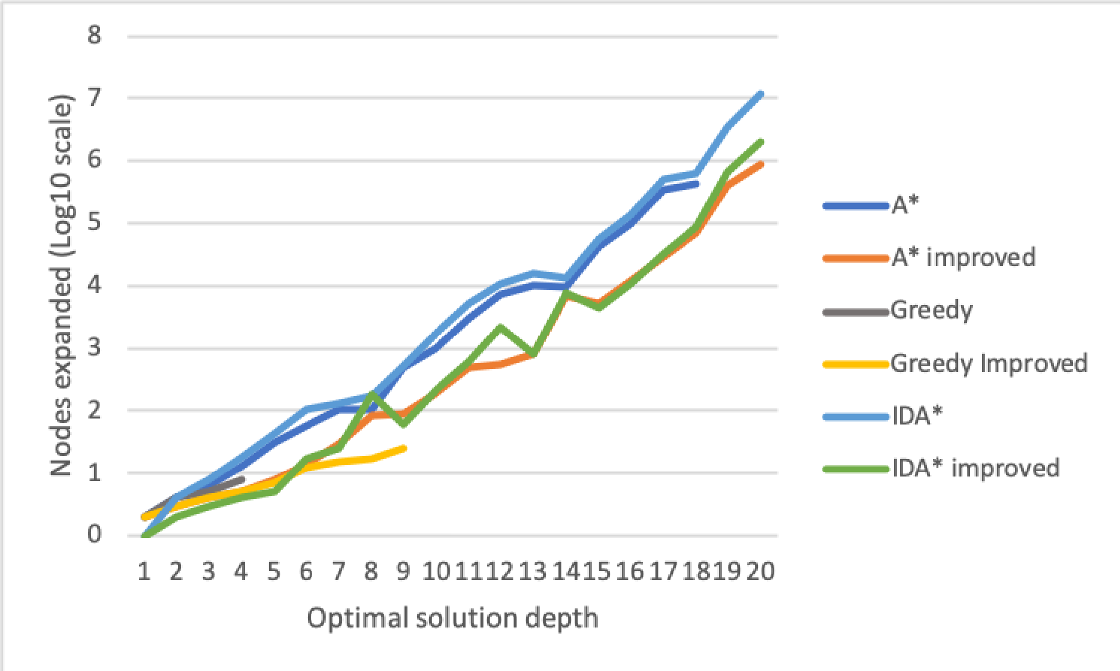


Figure 24: Number of nodes expanded for each heuristic as a function of the problem depth

4.7 Improved Successors function

An improvement over the get_successors() function was made. To prevent doing symmetric moves, that is, making a move on one node and the opposite one on the next node, each node keeps track of the last move made, and when expanding the nodes has that into account. This reduces the branching factor, and makes every algorithm extremely fast, as it can be seen in Figure 25.

Analysing the graph, the best search is greedy search. Such outcome is due to the fact that, by removing symmetric moves, the algorithm stops getting stuck into loops. In terms of uninformed searches, the best one was still bidirectional search.

Taking some conclusions, greedy is the best search if it doesn't end up in the loop, which doesn't happen here because symmetric moves are eliminated. However, it is still not optimal. For optimality and still good performance A*, or IDA* in order to save memory. If details about the problem are not enough and an heuristic can't be generated, then the search is the bidirectional. But, if it is not possible to calculate the predecessor function, and the branching factor of the problem is very big, the best option would be iterative deepening depth-first. This search always finds the optimal solution, and has the space complexity of DFS, which is much better than BFS.

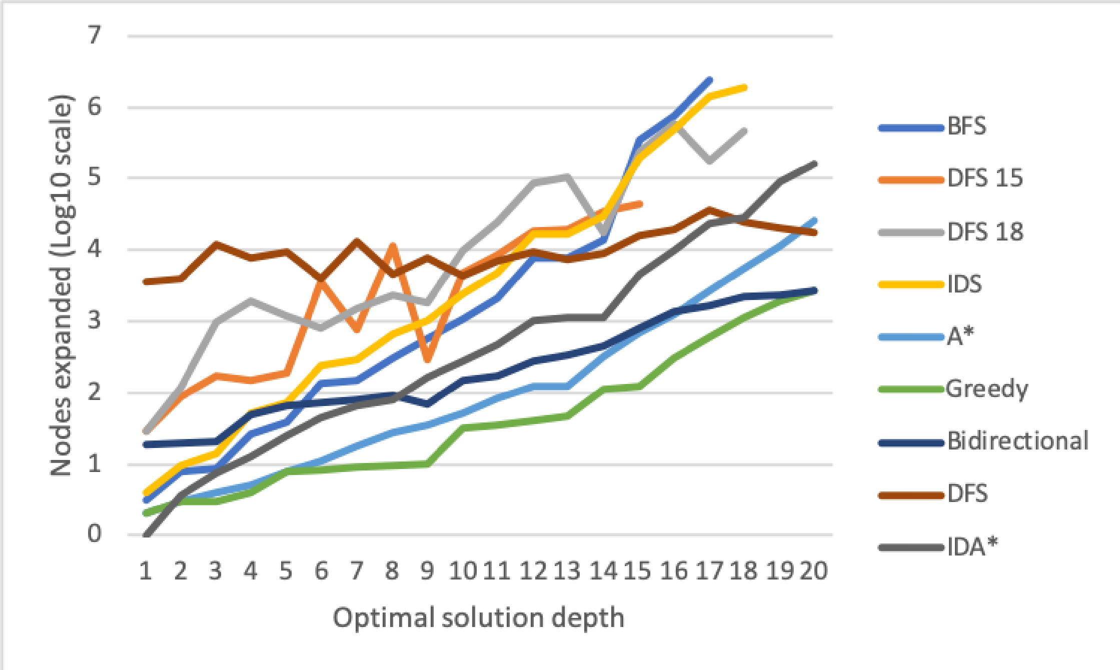


Figure 25: Number of nodes expanded for improved successors function as a function of the problem depth

4.8 Limitations

In terms of limitations, the only one would be that even the improved heuristic is not good enough because greedy still doesn't find solution for bigger depths. However, as mentioned in section 4.6, this can also be due to the fact that the greedy algorithm is not good for this particular problem.

List of Figures

1 Initial State and Goal State for the searches proof 2
2 Visited nodes during breadth-first search 3
3 Solution path for breadth-first search 4
4 Visited nodes during depth-first search 5
5 Solution path for depth-first search 5
6 Visited nodes during iterative deepening depth-first 6
7 Solution path for iterative deepening depth-first 7
8 A* visited nodes during search 8
9 Solution path for A* search 8
10 Number of nodes expanded for depth-first limited depth, and iterative-deepening algorithms as a function of the problem depth 9
11 Number of nodes expanded for A*, Greedy, bidirectional, depth-first no limit and breadth-first searches as a function of the problem depth 9
12 Number of nodes expanded for every search in log_{10} scale as a function of the problem depth 10
13 Depth of the solution found for each search as a function of the problem depth 10
14 Depth of the solution found for DFS with no limit as a function of the problem depth 11
15 Memory used for breadth-search, depth-first without limit and A* as a function of the problem depth 11
16 Memory used for depth-first limit 15 and 18, iterative-deepening and bidirectional search as a function of the problem depth 12
17 Bidirectional search visited nodes 15
18 Solution path for Bidirectional search 16
19 Greedy search visited nodes 16
20 Solution path for Greedy search 17
21 Iterative-deepening A* visited nodes 17
22 Solution path for iterative-deepening A* search 18
23 Number of nodes expanded for graph search as a function of the problem depth 19
24 Number of nodes expanded for each heuristic as a function of the problem depth 20
25 Number of nodes expanded for improved successors function as a function of the problem depth 21

Nomenclature

BFS Breadth-First search

DFS Depth-First search

IDA Iterative Deepening A* Search

IDS Iterative Deepening Search

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

[2] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artif. Intell.*, vol. 27, pp. 97–109, Sept. 1985.

[3] "Iterative deepening depth-first search," Nov 2019.

[4] S. Edelkamp and R. E. Korf, "The branching factor of regular search spaces," in *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, (Menlo Park, CA, USA), pp. 299–304, American Association for Artificial Intelligence, 1998.

[5] A. Chandel and M. Sood, "Searching and optimization techniques in artificial intelligence: A comparative study & complexity analysis," *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume*, vol. 3, 2014.

[6] R. E. Korf, "Algorithms and theory of computation handbook," ch. Artificial Intelligence Search Algorithms, pp. 22–22, Chapman & Hall/CRC, 2010.

[7] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984.

[8] M. Nosrati, R. Karimi, and H. A. Hasanvand, "Investigation of the*(star) search algorithms: Characteristics, methods and approaches," *World Applied Programming*, vol. 2, no. 4, pp. 251–256, 2012.

5 Code

5.1 main.py

Listing 1: main.py

```python
#!/usr/bin/python3

from node import Node
import methods
import sys, argparse
import matplotlib.pyplot as plt
import psutil

#use oned list instead of matrix to optimize space
initial_states = []
initial_states.insert(0, [0,'A','C',0,0,0,0,0,0,0,1,0,'B',0,0,0]) #depth 20
initial_states.insert(0, [0,'A','C',0,0,0,1,0,0,0,0,0,'B',0,0,0]) #depth 19
initial_states.insert(0, [1,0,'A',0,0,0,'C',0,0,0,0,0,'B',0,0,0]) #depth 18
initial_states.insert(0, [0,0,'A',0,0,0,'C',0,1,0,0,0,'B',0,0,0]) #depth 17
initial_states.insert(0, [0,0,'A',0,0,0,'C',0,'B',0,0,0,1,0,0,0]) #depth 16
initial_states.insert(0, [0,0,'A',0,0,0,'C',0,'B',0,0,0,0,0,0,1]) #depth 15
initial_states.insert(0, [0,0,'O',0,0,0,0,'O',0,0,0,0,'A','B','C',1]) #depth 14
initial_states.insert(0, ['A',0,'O',0,0,0,0,'O',0,'B','C',0,0,0,0,1]) #depth 13
initial_states.insert(0, [0,0,'O',0,0,'A','C','O','B',0,0,0,0,0,1,0]) #depth 12
initial_states.insert(0, [0,'A','O',0,0,0,0,'O',0,'B','C',0,0,0,0,1]) #depth 11
initial_states.insert(0, [0,0,'O',0,0,'A',0,'O',0,0,0,'B','C',0,0,1]) #depth 10
initial_states.insert(0, [1,0,'O',0,0,0,'A','O','B',0,0,0,0,0,'C',0]) #depth 9
initial_states.insert(0, [0,0,'O',0,0,'A',0,'O',0,'C',0,'B',0,0,0,1]) #depth 8
initial_states.insert(0, [0,0,'O',0,0,0,'A','O','B',0,0,0,0,'C',0,1]) #depth 7
initial_states.insert(0, [0,0,'O',0,0,0,'A','O',0,'B',0,0,'C',0,1,0]) #depth 6
initial_states.insert(0, [0,0,'O',0,'A',0,0,'O',0,'B',0,0,0,'C',0,1]) #depth 5
initial_states.insert(0, [0,0,'O',0,0,0,'A','O',0,'B',0,0,1,'C',0,0]) #depth 4
```

```
initial_states.insert(0, [1,0,'O',0,0,0,'A','O',0,'B',0,0,0,'C',0,0]) #depth 3
initial_states.insert(0, [0,0,'O',0,1,0,'A','O',0,'B',0,0,0,'C',0,0]) #depth 2
initial_states.insert(0, [0,0,'O',0,0,1,'A','O',0,'B',0,0,0,'C',0,0]) #depth 1
goal_state = [0,0,0,0,0,'A',0,0,0,'B',0,0,0,'C',0,1] #Use for depth bigger than 14
goal_state2 = [0,0,'O',0,0,'A',0,'O',0,'B',0,0,0,'C',0,1] #use for depth lower or equal to 14

def find_agent(initial, goal):
    start_agent = [None]*2
    end_agent = [None]*2
    for i in range(16):
        if initial[i] == 1:
            start_agent[0] = i%4
            start_agent[1] = i // 4

        if goal[i] == 1:
            end_agent[0] = i%4
            end_agent[1] = i // 4
    return start_agent, end_agent

def main():
    sol = False

    parser = argparse.ArgumentParser(description='A tutorial of argparse!')
    parser.add_argument("--m", choices=["BFS", "DFS", "IDS", "Bidirectional", "Astar", "IDA", "Greedy"], required=True, type=str, help="Method to use")
    parser.add_argument("--l", default=None, type=int, help="Depth limit for depth-first search")
    parser.add_argument("--g", choices=[True, False], default=False, type=bool, help="Whether or not to use graph search")
    parser.add_argument("--h", choices=[True, False], default=False, type=bool, help="Whether or not to use improved heuristic, for heuristic searches")
    parser.add_argument("--d", choices=[True, False], default=False, type=bool, help="Whether or not to use improved descendants function")
    parser.add_argument("--s", choices=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20], default=
```

```
            None, type=int, help="Optimal depth of the solution wanted to test")

        args = parser.parse_args()

        method = args.m
        limit = args.l
        graph_search = args.g
        improved_heuristic = args.h
        improved_descendants = args.d
        depth = args.s

        if depth == None:
                start_agent, end_agent = find_agent(initial_states[3], goal_state)
                start_node = Node(initial_states[3], start_agent, 0)
                start_node.count = 1
                end_node = Node(goal_state2, end_agent, 0) #used for bidirectional search
        elif depth <= 14:
                start_agent, end_agent = find_agent(initial_states[depth-1], goal_state)
                start_node = Node(initial_states[depth-1], start_agent, 0)
                start_node.count = 1
                end_node = Node(goal_state2, end_agent, 0) #used for bidirectional search
        else:
                start_agent, end_agent = find_agent(initial_states[depth-1], goal_state)
                start_node = Node(initial_states[depth-1], start_agent, 0)
                start_node.count = 1
                end_node = Node(goal_state, end_agent, 0) #used for bidirectional search

        if method == "DFS":
                sol = methods.dfs(start_node, goal_state, limit = limit, iterative = False, graphSearch = graph_search, improved_descendants = improved_descendants)
        elif method == "BFS":
                sol = methods.bfs(start_node, goal_state, graphSearch = graph_search, improved_descendants = improved_descendants)
        elif method == "IDS":
```

```
            sol = methods.idfs(start_node,goal_state, improved_descendants = improved_descendants)
    elif method == "Bidirectional":
            sol = methods.BidirectionalSearch(start_node, end_node, goal_state2, improved_descendants = improved_descendants)
    elif method == "Astar":
            sol = methods.Astar(start_node, goal_state, graphSearch = graph_search, improved_descendants = improved_descendants, improved_heuristic = improved_heuristic)
    elif method == "IDA":
            sol = methods.IDAstar(start_node, goal_state, improved_descendants= improved_descendants, improved_heuristic= improved_heuristic)
    elif method == "Greedy":
            sol = methods.Greedy(start_node, goal_state, improved_descendants = improved_descendants, improved_heuristic = improved_heuristic)
    else:
            print("Invalid method")

    if not(sol):
            print("No solution")

if __name__ == '__main__':
    main()
```

5.2 node.py

Listing 2: node.py

```
import numpy as np
import matplotlib.pyplot as plt
import random

class Node:

    def __init__(self, board, agent, depth, parent = None, move = None):
            """Creates a new instance of Node.

            Arguments:
                    board {list} -- Current layout of the board
                    agent {list} -- List with two members, the x and y coordinates of the agent.
```

```
                    depth {int} -- Depth of the current node

            Keyword Arguments:
                    parent {Node} -- Node that generated current node (
                       default: {None})
                    move {tuple} -- Describes the last move that lead to
                        this state (default: {None})
            """
            self.board = board
            self.agent = agent
            self.depth = depth
            #keep track of number of nodes visited
            self.count = 0
            self.parent = parent
            #keeps track of the move made previously
            self.move = move

    def __lt__(self, other):
            """When two nodes have the same herusitic value, this
               function is the tiebracker

            Arguments:
                    other {Node} -- Node with the same heuristic value.

            Returns:
                    [bool] -- True if the current node is better than
                       the other node, False otherwise.
            """
            letters_list = ['A', 'B', 'C']
            dist_1 = 10
            dist_2 = 10
            for tile in range(0,16):
                    if self.board[tile] in letters_list:
                            current_x =  tile % 4
                            current_y =  tile // 4
                            distance_to_agent = abs(current_x - self.
                               agent[0]) + abs(current_y - self.agent
                               [1])
                            dist_1 = min(dist_1, distance_to_agent)
                    if other.board[tile] in letters_list:
                            current_x =  tile % 4
                            current_y =  tile // 4
                            distance_to_agent = abs(current_x - other.
                               agent[0]) + abs(current_y - other.agent
```

```
                                        [1])
                                dist_2 = min(dist_2, distance_to_agent)

                if dist_2 > dist_1:
                        return True
                else:
                        return False

        #find all possible descendants
        def successors(self, improved= False):
                """Calculates the successors nodes of the current node.

                Keyword Arguments:
                        improved {bool} -- If True, takes into account the last move made, and it does not do the symmetric one (default: {False})

                Returns:
                        [list] -- List of the successors nodes.
                """
                desc = []
                #up, down, right and left, respectively
                possibleMoves = [(0,1),(0,-1),(1,0),(-1,0)]

                #randomize chosen moves
                random.shuffle(possibleMoves)

                for i in range(4):
                        if improved and self.move != None:
                                #if move is symmetric to the one done previously
                                if possibleMoves[i] == tuple(np.multiply((-1,-1), self.move)):
                                        continue

                        current_x_position = self.agent[0]
                        current_y_position = self.agent[1]

                        new_x_position = current_x_position + possibleMoves[i][0]
                        new_y_position = current_y_position + possibleMoves[i][1]

```

```python
                if new_x_position < 0 or new_y_position < 0 or new_x_position > 3 or new_y_position > 3:
                    continue

                board = self.board.copy()

                #check the value before change
                old_value = board[new_y_position * 4 + new_x_position]
                #obstacle, cannot pass
                if(old_value == 'O'):
                    continue

                board[current_y_position * 4 + current_x_position] = old_value
                board[new_y_position * 4 + new_x_position] = 1

                new_agent = [new_x_position,new_y_position]

                if improved:
                    desc.append(Node(board,new_agent,self.depth+1,self,possibleMoves[i]))
                else:
                    desc.append(Node(board,new_agent,self.depth+1,self))

        return desc

    def heuristic_manhattan(self, end_state, boost = False):
        """Calculates the manhattan heuristic from the current node.

        Arguments:
            end_state {list} -- Represents the final layout of the board.

        Keyword Arguments:
            boost {bool} -- When True, calculates a better version of manhattan heurisitc, where it takes into account
            the distance of the agent to the further misplaced tile. (default: {False})
        """
        value = 0
        letters_accounted = 0
```

```
                agent_distance_to_misplaced_tile = 0

                final_pos = self.get_position(end_state)

                for tile in range(0,16):
                        value_of_tile = self.board[tile]
                        if value_of_tile == 'A':
                                final_tile = final_pos[0]
                        elif value_of_tile == 'B':
                                final_tile = final_pos[1]
                        elif value_of_tile == 'C':
                                final_tile = final_pos[2]
                        else:
                                continue

                        #gets x and y coordinates of the tile in current node
                        current_x =  tile % 4
                        current_y =  tile // 4

                        #gets x and y coordinates of the tile in final node
                        final_x = final_tile % 4
                        final_y = final_tile // 4

                        #manhattan distance
                        distance = abs(current_x - final_x) + abs(current_y - final_y)
                        value += distance

                        #distance to the agent to the further misplaced tile
                        if distance != 0:
                                agent_distance_to_misplaced_tile = max(agent_distance_to_misplaced_tile, abs(current_x - self.agent[0]) + abs(current_y - self.agent[1]))

                        letters_accounted += 1
                        # no need to stay in the loop if already found the 3 letters
                        if letters_accounted == 3:
                                break

                #improved heuristic, that also takes into account the location of the agent
```

```
            if boost:
                return value + agent_distance_to_misplaced_tile
            else:
                return value

    def get_position(self, end_state):
        """Gets position of the tiles in goal state.

        Arguments:
            end_state {list} -- Represents the final layout of the board.

        Returns:
            [list] -- list with the position of 'A', 'B' and 'C', respectively.
        """
        list_posi = [None] * 3

        for tile in range(0,16):
            if end_state[tile] == 'A':
                list_posi[0] = tile
            elif end_state[tile] == 'B':
                list_posi[1] = tile
            elif end_state[tile] == 'C':
                list_posi[2] = tile

        return list_posi

    def build_hash(self):
        """Builds the hash of current node.

        Returns:
            [list] -- Hash of current node
        """
        board_hash = ""
        for i in range(0,16):
            board_hash += str(self.board[i])
        return board_hash

    def check_solution(self, end_state):
        """Checks if current node is te solution node.

        Arguments:
```

```
            end_state {list} -- Represents the final layout of the board.

        Returns:
            [bool] -- True if it is the solution node, False otherwise.
        """
        letters = ['A', 'B', 'C']
        for i in range(16):
            if self.board[i] != end_state[i]:
                if (self.board[i] in letters) or (end_state[i] in letters):
                    return False
        return True

    def print_board(self, fig, dimensions, count):
        """Prints the board.

        Arguments:
            fig {matplotlib.pyplot.figure} -- The figure to attach the board to.
            dimensions {int} -- the dimensions of the subfigure.
            count {int} -- The number of printed states until now.
        """
        ax = fig.add_subplot(dimensions, dimensions, count)

        for x in range(5):
            ax.plot([x, x], [0,4], 'k')
        for y in range(5):
            ax.plot([0, 4], [y,y], 'k')

        agent = plt.imread('../Images/agent.png')
        obstacle = plt.imread('../Images/block.jpg')
        A = plt.imread('../Images/A_letter.png')
        B = plt.imread('../Images/B_letter.png')
        C = plt.imread('../Images/C_letter.png')
        extent = np.array([-0.3, 0.3, -0.3, 0.3])
        ax.set_axis_off()

        for i in range(16):
            x_coord = i % 4 + 0.5
            y_coord = 3.5 - i // 4
            if(self.board[i] == 'A'):
```

```python
                ax.imshow(A, extent=extent + [x_coord, x_coord, y_coord, y_coord])
            elif(self.board[i] == 'B'):
                ax.imshow(B, extent=extent + [x_coord, x_coord, y_coord, y_coord])
            elif(self.board[i] == 'C'):
                ax.imshow(C, extent=extent + [x_coord, x_coord, y_coord, y_coord])
            elif(self.board[i] == 'O'):
                ax.imshow(obstacle, extent=extent + [x_coord, x_coord, y_coord, y_coord])
            elif(self.board[i] == 1):
                ax.imshow(agent, extent=extent + [x_coord, x_coord, y_coord, y_coord])

        ax.set(xticks=[], yticks=[])
        ax.axis('image')

        if self.parent != None:
            ax.set_title("N: " + str(self.count) + ", P: " + str(self.parent.count) + ", D: " + str(self.depth))
        else:
            ax.set_title("N: " + str(self.count) + ", P: None" + ", D: " + str(self.depth))

    def print_path(self, fig, dimensions, count):
        """Backtracks the path from the solution to the start node.

        Arguments:
            fig {matplotlib.pyplot.figure} -- The figure to attach the board to.
            dimensions {int} -- the dimensions of the subfigure.
            count {int} -- The number of nodes backtracked until now.
        """
        if self.parent != None:
            self.parent.print_path(fig, dimensions, count = count - 1)

        self.print_board(fig, dimensions, count)

    def print_path_reserse(self, fig, dimensions, count):
        """Backtracks the path for the bottom-up search in Bidirectional search. Besides that, it calculates
```


University of Southampton School of Electronics and Computer Science

```
                how many nodes of the bottom-up search belong to the actual solution, that is, when it reaches a state with
                all the tiles in the correct place ignoring the position of the agent.

                Arguments:
                        fig {matplotlib.pyplot.figure} -- The figure to attach the board to.
                        dimensions {int} -- the dimensions of the subfigure.
                        count {int} -- The number of nodes backtracked until now.
                """
                self.print_board(fig, dimensions, count)

                if self.parent != None:
                        depth_extra = self.parent.print_path_reserse(fig, dimensions, count + 1)
```

5.3 methods.py

Listing 3: methods.py

```
import numpy as np
import time, sys
import math
import matplotlib.pyplot as plt
from queue import PriorityQueue

sys.setrecursionlimit(9000)

def print_solution(state1, number_nodes_expanded, goal_state, state2 = None):
        """When solution is found, this method is called to print the solution path

        Arguments:
                state1 {Node} -- The final node of the search, where the solution was found
                number_nodes_expanded {int} -- Total number of nodes expanded during search
                goal_state {Node} -- final layout of the board, used for Bidirectional search to find actual depth of solution

```

```
    Keyword Arguments:
        state2 {Node} -- If the search used was Bidirectional search, it gives a second node, correspondent to the final
        node in the bottom-up search (default: {None})

    Returns:
        {int} -- In case the search was Bidirectional, it calculates the actual depth of the solution.
    """

    if state2 != None:
        total_depth = state1.depth + state2.depth
    else:
        total_depth = state1.depth
        print("Solution found at depth: " + str(total_depth))

    dimensions = int(math.sqrt(total_depth)) + 1

    fig = plt.figure(figsize=[4 * dimensions, 4 * dimensions])

    state1.print_path(fig, dimensions, state1.depth + 1)

    if state2 != None:
        state2.parent.print_path_reserse(fig, dimensions, state1.depth + 2)
        middle_depth = state1.depth
        found = False
        while True:
            if state1.check_solution(goal_state):
                middle_depth = state1.depth
                found = True
                #check if the solution can still be find in previous nodes
                state1 = state1.parent
            else:
                if state1.parent == None:
                    break
                else:
                    state1 = state1.parent

        state2 = state2.parent
        while not(found):
            if state2.check_solution(goal_state):
                middle_depth += 1
```

```
                                found = True
                        else:
                                middle_depth += 1
                                state2 = state2.parent

                print("Solution found at depth: " + str(middle_depth))
                plt.show()
                return middle_depth
        else:
                plt.show()
                return None

def bfs(start_node, goal_state, graphSearch = False, improved_descendants =
    False):
        """Runs breadth-first search.

        Arguments:
                start_node {Node} -- Start node, which describes where the
                    search starts.
                goal_state {list} -- Goal state, which represents the final
                    layout of the board.

        Keyword Arguments:
                graphSearch {bool} -- When set to True, does BFS graph
                    search, where it doesn't expanded previously expanded
                    noded (default: {False})
                improved_descendants {bool} -- When set to True, uses the
                    improved version of descendants function (default: {False
                    })

        Returns:
                {bool} -- Returns True if it was able to find a solution,
                    and False otherwise.
        """
        fringe = [start_node]
        number_nodes_expanded = 0
        number_nodes_visited = 1

        child_nodes = []

        if graphSearch:
                closed = set()

        t0 = time.time()
```

```
    while len(fringe) > 0:
        node = fringe.pop(0)
        node.count = number_nodes_visited
        number_nodes_visited += 1

        t1 = time.time()
        if (t1 - t0) > 900:
            print("It took more than 15 min")
            return False

        if node.check_solution(goal_state):
            print("Expanded nodes: " + str(number_nodes_expanded))
            _ = print_solution(node, number_nodes_expanded, goal_state)
            return True

        if graphSearch:
            if node.build_hash() not in closed:
                closed.add(node.build_hash())
                number_nodes_expanded += 1
                child_nodes = node.successors(improved_descendants)
                for i in range(len(child_nodes)):
                    fringe.append(child_nodes[i])

        else:
            number_nodes_expanded += 1
            child_nodes = node.successors(improved_descendants)
            for i in range(len(child_nodes)):
                fringe.append(child_nodes[i])

    return False

def dfs(start_node, goal_state, limit = None, iterative = False, graphSearch = False, improved_descendants = False):
    """Runs depth-first tree search.

    Arguments:
        start_node {Node} -- Start node, which describes where the search starts.
        goal_state {list} -- Goal state, which represents the final layout of the board.

```

```
        Keyword Arguments:
                limit {int} -- Limits the depth to which DFS goes. (default:
                     {None})
                iterative {bool} -- When set to True, uses DFS as the search
                     method in iterative deepening search (default: {False})
                graphSearch {bool} -- When set to True, does DFS graph
                   search, where it doesn't expanded previously expanded
                   noded (default: {False})
                improved_descendants {bool} -- When set to True, uses the
                   improved version of descendants function (default: {False
                   })

        Returns:
                {bool} -- if iterative argument is set to False, Returns
                   True if it was able to find a solution, and False
                   otherwise.
                {bool, int, int} - if iterative argument is set to True,
                   returns True or False,
                depending if it is able to find a solution or not, and
                   number of nodes expanded and depth of solution
        """
        fringe = [start_node]
        number_nodes_expanded = 0
        number_nodes_visited = 0

        t0 = time.time()

        if graphSearch:
                closed = {} #hash_map

        while len(fringe) > 0:
                number_nodes_visited += 1
                node = fringe.pop()
                node.count = number_nodes_visited

                t1 = time.time()
                if (t1 - t0) > 900:
                        print("It took more than 15 min")
                        if iterative:
                                return False
                        else:
                                return False

                if node.check_solution(goal_state):
```

```
                        _ = print_solution(node, number_nodes_expanded,
                            goal_state)
                        if iterative:
                                return True, number_nodes_visited
                        print("Expanded nodes: " + str(number_nodes_expanded
                            ))
                        return True


                if limit == None or node.depth < limit:
                        if graphSearch:
                                node_hash = node.build_hash()
                                node_depth = node.depth
                                #can also add if it's found i at smaller
                                    depth. Grants solution every time
                                if node_hash not in closed or closed[
                                    node_hash] > node_depth:
                                        closed[node_hash] = node_depth
                                        number_nodes_expanded += 1
                                        child_nodes = node.successors(
                                            improved_descendants)
                                        for i in range(len(child_nodes)):
                                                fringe.append(child_nodes[i
                                                    ])
                        else:
                                number_nodes_expanded += 1
                                child_nodes = node.successors(
                                    improved_descendants)
                                for i in range(len(child_nodes)):
                                        fringe.append(child_nodes[i])

        if iterative:
                return False, number_nodes_visited

        return False

def idfs(start_node, goal_state, improved_descendants = False):
        """Runs iterative-deepening depth-first search.

        Arguments:
                start_node {Node} -- Start node, which describes where the
                    search starts.
                goal_state {list} -- Goal state, which represents the final
                    layout of the board.
```

```

	Keyword Arguments:
		improved_descendants {bool} -- When set to True, uses the improved version of descendants function (default: {False})

	Returns:
		{bool} -- Returns True if it was able to find a solution, and False otherwise.
	"""
	number_nodes_expanded = 0
	t0 = time.time()

	for lim in range(21): #from depth 0 to 20
		solution, number_nodes_expanded_iter = dfs(start_node, goal_state,  lim, iterative= True, improved_descendants= improved_descendants)
		number_nodes_expanded += number_nodes_expanded_iter

		t1 = time.time()
		if (t1 - t0) > 900:
			print("It took more than 15 min")
			return False

		if solution:
			print("Expanded nodes: " + str(number_nodes_expanded))
			return True

	return False


def BidirectionalSearch(start_node, end_node, goal_state, improved_descendants = False):
	"""Runs Bidirectional Search, with BFS search in each of the directions.

	Arguments:
		start_node {Node} -- Start node, which describes where the search starts.
		end_node {Node} -- Goal Node, which is the node with the goal board layout, first on the bottom-up search.
		goal_state {list} -- Goal state, which represents the final layout of the board.
```

```

        Keyword Arguments:
                improved_descendants {bool} -- When set to True, uses the improved version of descendants function (default: {False})

        Returns:
                {bool} -- Returns True if it was able to find a solution, and False otherwise.
        """
        queue_down = [start_node]
        queue_up = [end_node]

        visited_nodes_down = set()
        visited_nodes_up = set()

        number_nodes_expanded = 0
        number_nodes_visited = 0

        child_nodes_down = []
        child_nodes_up = []

        hash_value_down = {}
        hash_value_up = {}

        t0 = time.time()

        while len(queue_down) > 0 or len(queue_up) > 0:
                top_expanded = False
                bottom_expanded = False

                #if the search down still has nodes to expand
                if len(queue_down) > 0:
                        node_down = queue_down.pop(0)
                        bottom_expanded = True
                        number_nodes_visited += 1
                        node_down.count = number_nodes_visited

                #if the search up still has nodes to expand
                if len(queue_up) > 0:
                        node_up = queue_up.pop(0)
                        top_expanded = True
                        number_nodes_visited += 1
                        node_up.count = number_nodes_visited
```

```

        t1 = time.time()
        if (t1 - t0) > 900:
            print("It took more than 15 min")
            return False

        if bottom_expanded:
            node_down_hash = node_down.build_hash()

            if node_down_hash not in visited_nodes_down:
                number_nodes_expanded += 1
                visited_nodes_down.add(node_down_hash)
                hash_value_down[node_down_hash] = node_down
                child_nodes_down = node_down.successors(
                    improved_descendants)

                for i in range(len(child_nodes_down)):
                    queue_down.append(child_nodes_down[i
                        ])
            else:
                child_nodes_down = []

        if top_expanded:
            node_up_hash = node_up.build_hash()
            if node_up_hash not in visited_nodes_up:
                visited_nodes_up.add(node_up_hash)
                hash_value_up[node_up_hash] = node_up

                number_nodes_expanded += 1
                child_nodes_up = node_up.successors(
                    improved_descendants)

                for i in range(len(child_nodes_up)):
                    queue_up.append(child_nodes_up[i])
            else:
                child_nodes_up = []

        #The node expanded on the search down was already expanded
            in the search up or vice-versa
        if bottom_expanded and (node_down_hash in visited_nodes_up):
            print("Expanded nodes: " + str(number_nodes_expanded
                ))
            depth_found = print_solution(node_down,
                number_nodes_expanded, goal_state, hash_value_up[
```

```
node_down_hash])
                        return True
                elif top_expanded and (node_up_hash in visited_nodes_down):
                        print("Expanded nodes: " + str(number_nodes_expanded))
                        depth_found = print_solution(hash_value_down[node_up_hash], number_nodes_expanded, goal_state, node_up)
                        return True

        return False

def Astar(start_node, goal_state, graphSearch = False, improved_descendants = False, improved_heuristic = False):
        """Runs A* tree search.

        Arguments:
                start_node {Node} -- Start node, which describes where the search starts.
                goal_state {list} -- Goal state, which represents the final layout of the board.

        Keyword Arguments:
                graphSearch {bool} -- When set to True, does BFS graph search, where it doesn't expanded previously expanded noded (default: {False})
                improved_descendants {bool} -- When set to True, uses the improved version of descendants function (default: {False})
                improved_heuristic {bool} -- When set to True, uses the improved version of manhattan distance heuristic (default: {False})

        Returns:
                {bool} -- Returns True if it was able to find a solution, and False otherwise.
        """
        prior_queue = PriorityQueue()
        prior_queue.put((start_node.heuristic_manhattan(goal_state, improved_heuristic), start_node))

        number_nodes_expanded = 0
        number_nodes_visited = 0

```


University of Southampton School of Electronics and Computer Science

```
    t0 = time.time()

    if graphSearch:
            closed = set()

    while not prior_queue.empty():
        _, node = prior_queue.get()
        number_nodes_visited += 1
        node.count = number_nodes_visited

        t1 = time.time()
        if (t1 - t0) > 900:
            print("It took more than 15 min")
            return False

        if node.check_solution(goal_state):
            print("Expanded nodes: " + str(number_nodes_expanded))
            _ = print_solution(node, number_nodes_expanded, goal_state)
            return True

        if graphSearch:
            if node.build_hash() not in closed:
                closed.add(node.build_hash())
                number_nodes_expanded += 1
                child_nodes = node.successors(improved_descendants)
                for child in child_nodes:
                    child_h = child.heuristic_manhattan(goal_state, improved_heuristic)
                    child_f = child_h + child.depth
                    prior_queue.put((child_f, child))
        else:
            number_nodes_expanded += 1
            child_nodes = node.successors(improved_descendants)
            for child in child_nodes:
                child_h = child.heuristic_manhattan(goal_state, improved_heuristic)
                child_f = child_h + child.depth
                prior_queue.put((child_f, child))

    return False
```

```python
def DFSAstar(start_node, goal_state, threshold, improved_descendants = False, improved_heuristic = False):
    """Runs the different depth-first searches for IDA*.

    Arguments:
        start_node {Node} -- Start node, which describes where the search starts.
        goal_state {list} -- Goal state, which represents the final layout of the board.
        threshold {int} -- Threshold for the search. Nodes with bigger heuristic value that this are cut-off.

    Keyword Arguments:
        improved_descendants {bool} -- When set to True, uses the improved version of descendants function (default: {False})
        improved_heuristic {bool} -- When set to True, uses the improved version of manhattan distance heuristic (default: {False})

    Returns:
        {bool} -- Returns True if it was able to find a solution, and False otherwise.
        {int} -- Number of nodes expanded in the depth-first search
    """
    fringe = [start_node]
    number_nodes_expanded = 0
    number_nodes_visited = 0
    child_nodes = []

    t0 = time.time()
    new_threshold = sys.maxsize

    while len(fringe) > 0:
        node = fringe.pop()
        number_nodes_visited += 1
        node.count = number_nodes_visited

        t1 = time.time()
        if (t1 - t0) > 900:
            print("It took more than 15 min")
            return False, number_nodes_expanded, new_threshold

        if node.check_solution(goal_state):
```

```
                _ = print_solution(node, number_nodes_expanded, goal_state)
                return True, number_nodes_expanded, new_threshold

        child_nodes = node.successors(improved_descendants)
        number_nodes_expanded += 1

        for child in child_nodes:
            child_h = child.heuristic_manhattan(goal_state, improved_heuristic)
            child_f = child_h + child.depth

            if child_f <= threshold:
                fringe.append(child)
            else:
                new_threshold = min(new_threshold, child_f)

    return False, number_nodes_expanded, new_threshold

def IDAstar(start_node, goal_state, improved_descendants = False, improved_heuristic = False):
    """Runs Iterative—deppening A* .

    Arguments:
        start_node {Node} —— Start node, which describes where the search starts.
        goal_state {list} —— Goal state, which represents the final layout of the board.

    Keyword Arguments:
        improved_descendants {bool} —— When set to True, uses the improved version of descendants function (default: {False})
        improved_heuristic {bool} —— When set to True, uses the improved version of manhattan distance heuristic (default: {False})

    Returns:
        {bool} —— Returns True if it was able to find a solution, and False otherwise.
    """
    threshold = start_node.heuristic_manhattan(goal_state, improved_heuristic)
    number_nodes_expanded = 0
```

```
    t0 = time.time()

    while True:
        sol, number_nodes, new_treshold = DFSAstar(start_node, goal_state, threshold, improved_descendants, improved_heuristic)
        number_nodes_expanded += number_nodes
        t1 = time.time()

        if (t1 - t0) > 900:
            print("Took more than 15 minutes")
            return False

        if new_treshold == sys.maxsize:
            return False

        if sol:
            print("Number of nodes: " + str(number_nodes_expanded))
            return True
        else:
            threshold = new_treshold

    return False

def Greedy(start_node, goal_state, improved_descendants = False, improved_heuristic = False):
    """Runs Greedy tree search.

    Arguments:
        start_node {Node} -- Start node, which describes where the search starts.
        goal_state {list} -- Goal state, which represents the final layout of the board.

    Keyword Arguments:
        improved_descendants {bool} -- When set to True, uses the improved version of descendants function (default: {False})
        improved_heuristic {bool} -- When set to True, uses the improved version of manhattan distance heuristic (default: {False})

    Returns:
```

```
        {bool} -- Returns True if it was able to find a solution, and False otherwise.
    """
    prior_queue = PriorityQueue()
    prior_queue.put((start_node.heuristic_manhattan(goal_state, improved_heuristic), start_node))

    number_nodes_expanded = 0
    number_nodes_visited = 0

    t0 = time.time()

    while not prior_queue.empty():
        _, node = prior_queue.get()
        number_nodes_visited += 1
        node.count = number_nodes_visited

        t1 = time.time()

        if (t1 - t0) > 900:
            print("It took more than 15 min")
            return False

        if node.check_solution(goal_state):
            print("Expanded nodes: " + str(number_nodes_expanded))
            _ = print_solution(node, number_nodes_expanded, goal_state)
            return True

        number_nodes_expanded += 1

        child_nodes = node.successors(improved_descendants)

        for child in child_nodes:
            child_f = child.heuristic_manhattan(goal_state, improved_heuristic)
            prior_queue.put((child_f, child))

    return False
```