UNIVERSITY OF Southampton
School of Electronics and Computer Science

Search Algorithms Study

AUTHOR

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

December 12, 2019

Contents

1 Approach

The Blocksworld tile puzzle consists of a 4X4 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 section 2. 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 - Represents the state of the board at a given state. Represented as a 1 dimension array in order to save memory.
- agent - represents the position of the agent. This way, there is no need to find the agent when calculating the descendants of a node. Therefore, saving time.
- parent - node's parent.
- depth - Depth of each node.
- count - keeps track of the number of nodes visited, representing the node's number.
- move - represents the last move made. This allows to make an improvement in the descendants function, so that the descendants of a node don't do a move symmetric to the previous one. This will be explained with more detail in section 4.1.

There were implemented several methods for this class. In order to get the descendants of a node, there is *descendants()*. 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 descendant node, whose parent is the current node, and with depth increased by 1 over the parent's one. This 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. This method is improved as well, as explained in section 4.5. Besides that, there are auxiliary methods: get_position(), which allows to get the position of the blocks; build_hash(), that is used from graph search which is an extra in section 4.4; check_solution() to check if a node is the final state; printing functions to print the path from initial node to the goal.

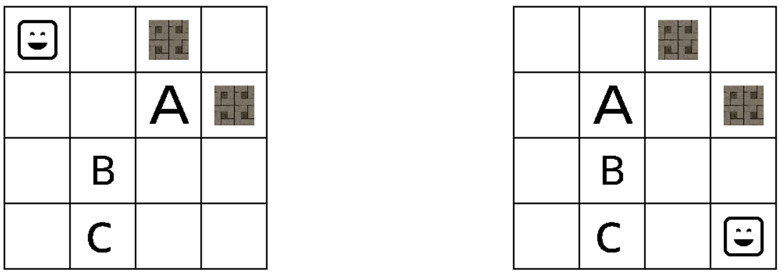
This class is the foundation for the search algorithms. In the search algorithms the only special considerations were the use of a priority queue for the heuristic searches and a set for graphs searches. The first is because the priority queue is implemented as a heap, which allows for a faster access and insertion of elements, where time complexity is 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. The insertion and lookup time are O(1) on average and O(N) in the worst case, which is better than using an array.

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 scalibility, 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 in the goal. Each block is represented by a letter. There is also the introduction of blocks that can't be moved, which are mentioned in section 4.2. 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. The first ones means that the 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 given by heuristics.



(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's going to visit the nodes in order of expansion, which 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 node 2 and 3, which are the descendants 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 descendants of node 2, followed by the descendants of node 3, that are going to have depth 2, and so on. It finds a solution at node 16 with depth 3, and the path can be found by progressively following the parents of each node starting from the solution, which in this case is going to be: 1-2-6-16, as it can be seen in Figure 3.

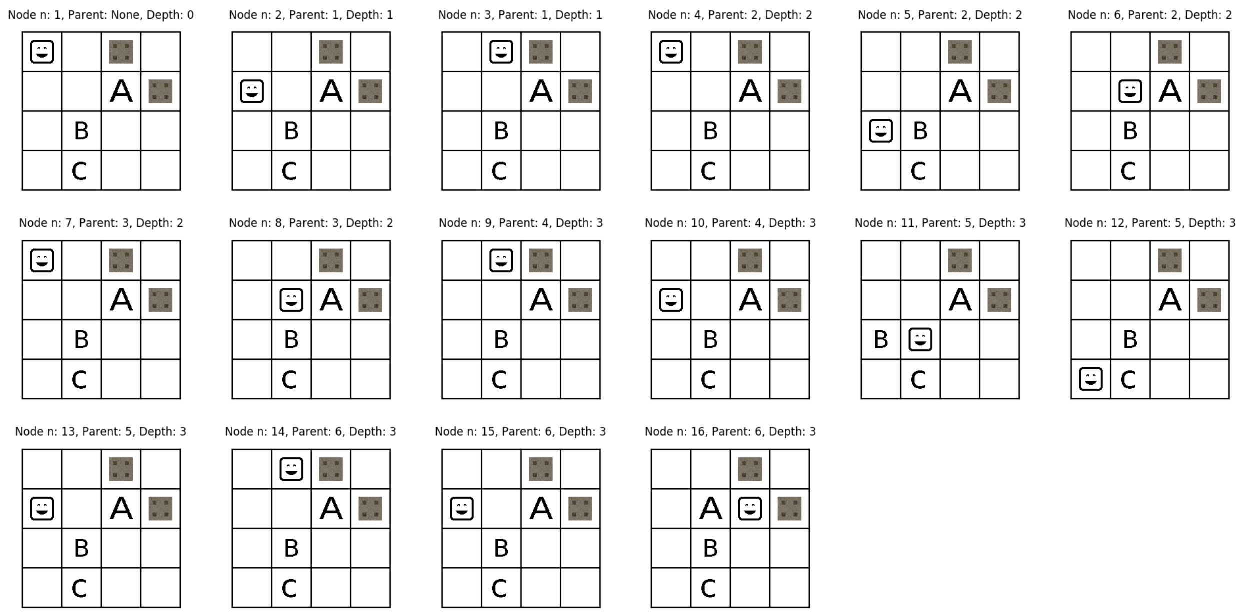

Node n: 1, Parent: None, Depth: 0
Node n: 2, Parent: 1, Depth: 1
Node n: 3, Parent: 1, Depth: 1
Node n: 4, Parent: 2, Depth: 2
Node n: 5, Parent: 2, Depth: 2
Node n: 6, Parent: 2, Depth: 2
Node n: 7, Parent: 3, Depth: 2
Node n: 8, Parent: 3, Depth: 2
Node n: 9, Parent: 4, Depth: 3
Node n: 10, Parent: 4, Depth: 3
Node n: 11, Parent: 5, Depth: 3
Node n: 12, Parent: 5, Depth: 3
Node n: 13, Parent: 5, Depth: 3
Node n: 14, Parent: 6, Depth: 3
Node n: 15, Parent: 6, Depth: 3
Node n: 16, Parent: 6, Depth: 3
A
B
C

Figure 2: BFS visited nodes during search

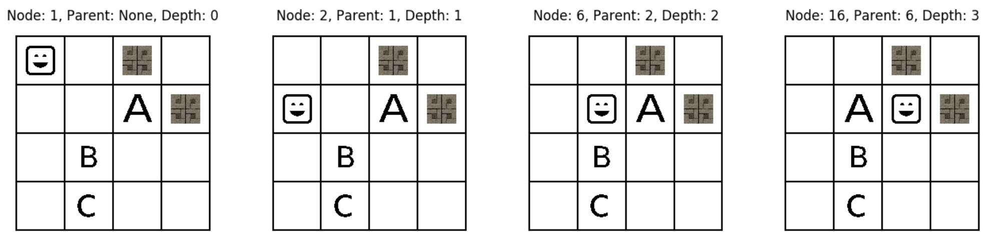

Node: 1, Parent: None, Depth: 0
Node: 2, Parent: 1, Depth: 1
Node: 6, Parent: 2, Depth: 2
Node: 16, Parent: 6, Depth: 3
A
B
C

Figure 3: Solution path for BFS

2.2 Depth-First

Depth-first algorithm expands a branch of the tree in depth. This is accomplished with 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 does backtracking, that is, for the last visited node, it goes to his parent and expands in depth other descendants that haven't been visited. 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 descendants, followed by one of the descendants 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 descendants of its parent. Since none of the descendants is the goal state, is has to backtrack again to find the next node which hasn't been visited, which is one of the descendants of node 6. It keeps doing this until it finds a solution, which is found on the 37th node, at depth 7. By backtracking until the root node, the solution path obtained is the one shown in Figure 5: 1-2-3-4-5-25-33-37.

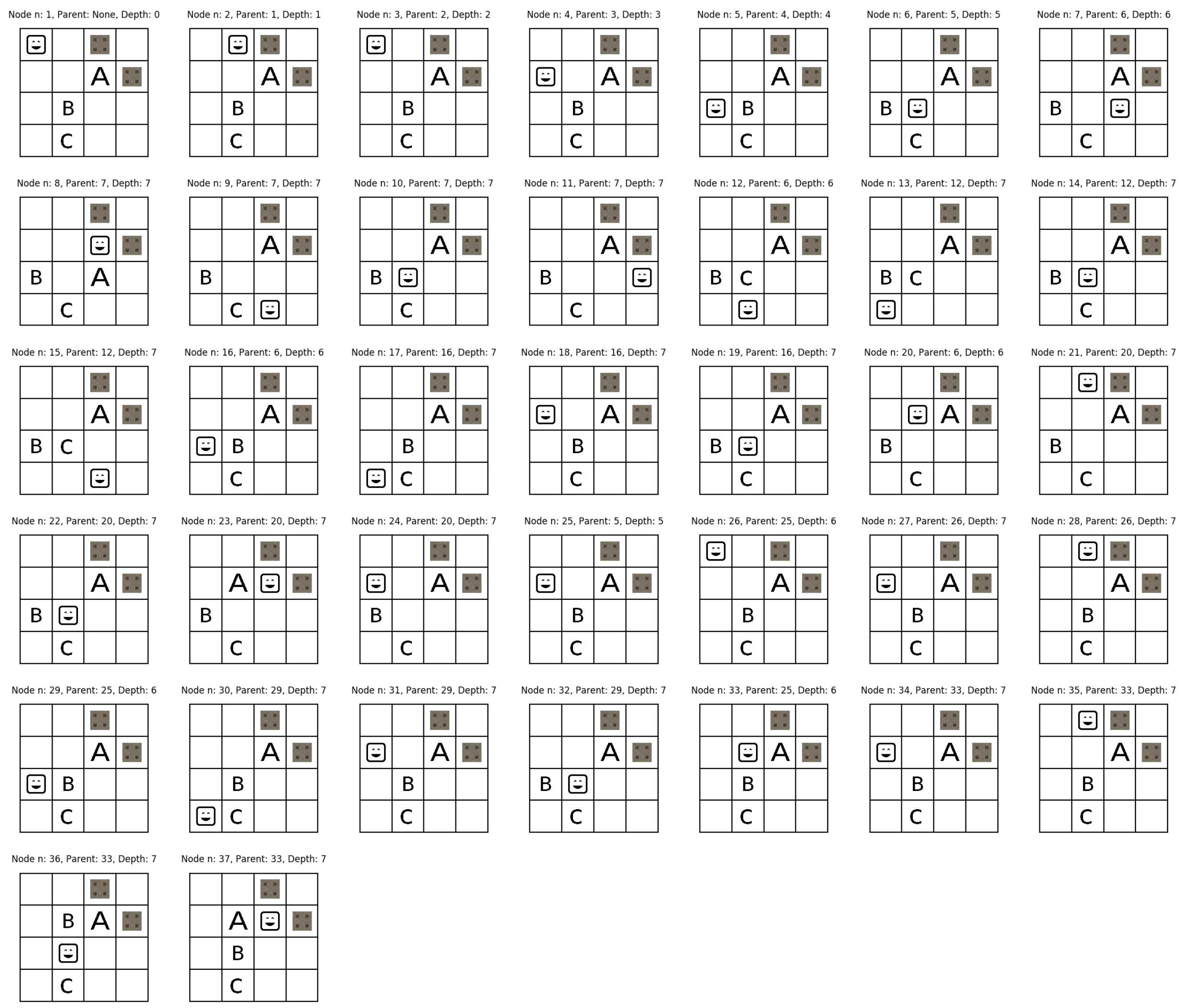

Node n: 1, Parent: None, Depth: 0
Node n: 2, Parent: 1, Depth: 1
Node n: 3, Parent: 2, Depth: 2
Node n: 4, Parent: 3, Depth: 3
Node n: 5, Parent: 4, Depth: 4
Node n: 6, Parent: 5, Depth: 5
Node n: 7, Parent: 6, Depth: 6
Node n: 8, Parent: 7, Depth: 7
Node n: 9, Parent: 7, Depth: 7
Node n: 10, Parent: 7, Depth: 7
Node n: 11, Parent: 7, Depth: 7
Node n: 12, Parent: 6, Depth: 6
Node n: 13, Parent: 12, Depth: 7
Node n: 14, Parent: 12, Depth: 7
Node n: 15, Parent: 12, Depth: 7
Node n: 16, Parent: 6, Depth: 6
Node n: 17, Parent: 16, Depth: 7
Node n: 18, Parent: 16, Depth: 7
Node n: 19, Parent: 16, Depth: 7
Node n: 20, Parent: 6, Depth: 6
Node n: 21, Parent: 20, Depth: 7
Node n: 22, Parent: 20, Depth: 7
Node n: 23, Parent: 20, Depth: 7
Node n: 24, Parent: 20, Depth: 7
Node n: 25, Parent: 5, Depth: 5
Node n: 26, Parent: 25, Depth: 6
Node n: 27, Parent: 26, Depth: 7
Node n: 28, Parent: 26, Depth: 7
Node n: 29, Parent: 25, Depth: 6
Node n: 30, Parent: 29, Depth: 7
Node n: 31, Parent: 29, Depth: 7
Node n: 32, Parent: 29, Depth: 7
Node n: 33, Parent: 25, Depth: 6
Node n: 34, Parent: 33, Depth: 7
Node n: 35, Parent: 33, Depth: 7
Node n: 36, Parent: 33, Depth: 7
Node n: 37, Parent: 33, Depth: 7

Figure 4: DFS visited nodes during search

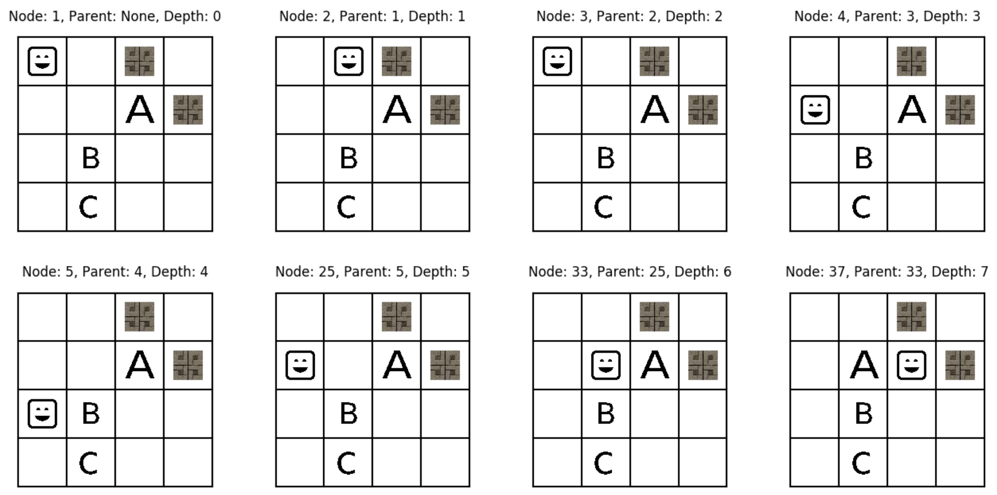

Node: 1, Parent: None, Depth: 0
Node: 2, Parent: 1, Depth: 1
Node: 3, Parent: 2, Depth: 2
Node: 4, Parent: 3, Depth: 3
Node: 5, Parent: 4, Depth: 4
Node: 25, Parent: 5, Depth: 5
Node: 33, Parent: 25, Depth: 6
Node: 37, Parent: 33, Depth: 7
A
B
C

Figure 5: Solution path for DFS

2.3 Iterative deepening depth-first search

This search consists of 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. This search 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 [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 parent 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 parent 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 6.

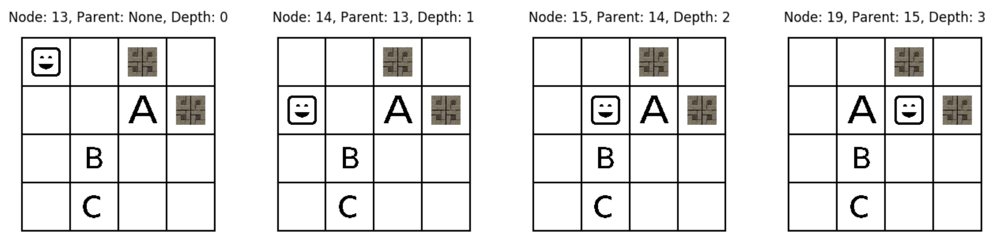

Node: 13, Parent: None, Depth: 0
Node: 14, Parent: 13, Depth: 1
Node: 15, Parent: 14, Depth: 2
Node: 19, Parent: 15, Depth: 3
A
B
C

Figure 6: Solution path for IDFS

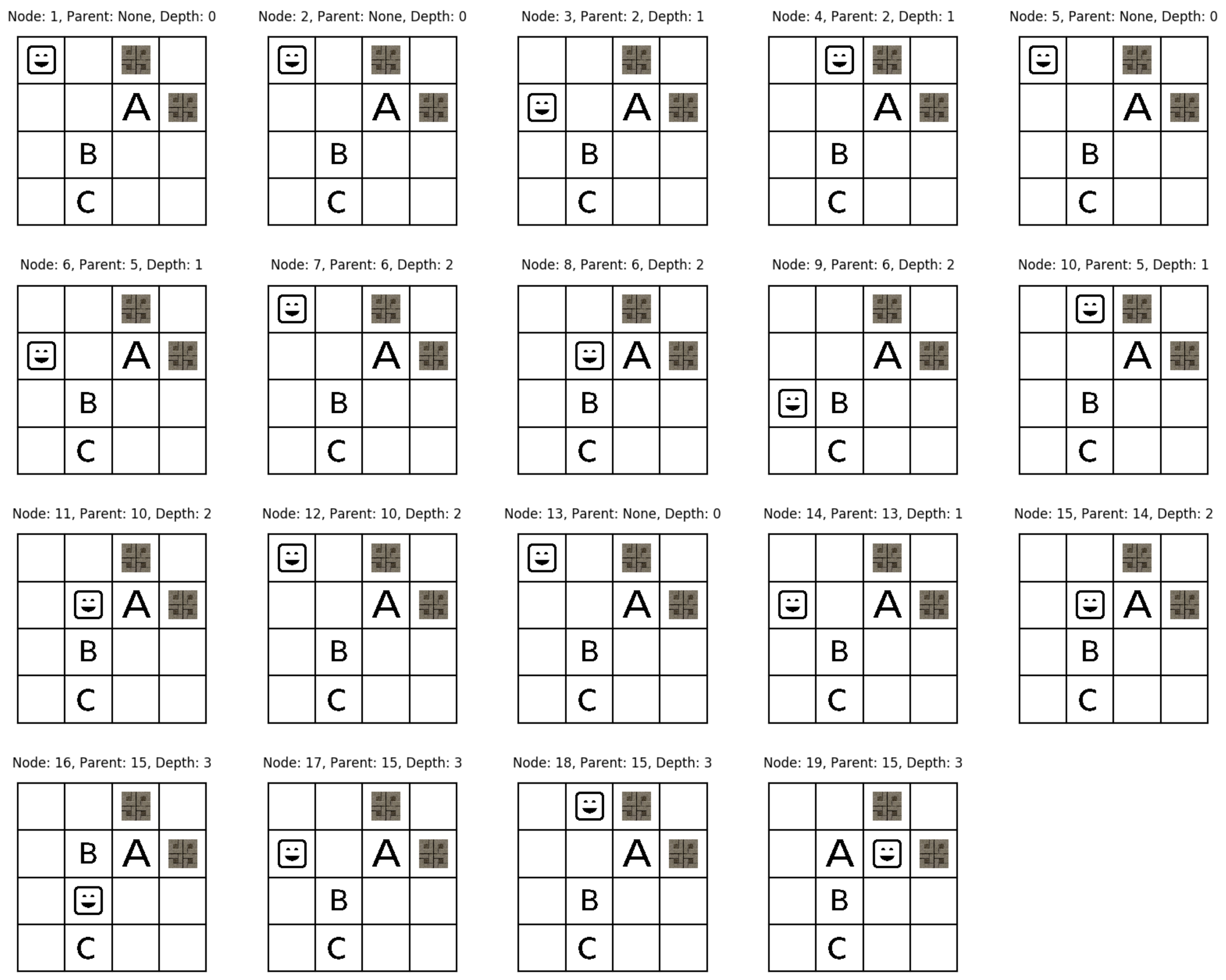

Node: 1, Parent: None, Depth: 0
Node: 2, Parent: None, Depth: 0
Node: 3, Parent: 2, Depth: 1
Node: 4, Parent: 2, Depth: 1
Node: 5, Parent: None, Depth: 0
Node: 6, Parent: 5, Depth: 1
Node: 7, Parent: 6, Depth: 2
Node: 8, Parent: 6, Depth: 2
Node: 9, Parent: 6, Depth: 2
Node: 10, Parent: 5, Depth: 1
Node: 11, Parent: 10, Depth: 2
Node: 12, Parent: 10, Depth: 2
Node: 13, Parent: None, Depth: 0
Node: 14, Parent: 13, Depth: 1
Node: 15, Parent: 14, Depth: 2
Node: 16, Parent: 15, Depth: 3
Node: 17, Parent: 15, Depth: 3
Node: 18, Parent: 15, Depth: 3
Node: 19, Parent: 15, Depth: 3

Figure 7: Iterative deepening depth-first visited nodes during search

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 expand the nodes blindly, but it also takes into account another information about the node. It's a combination of greedy search, which is explained more in depth in section 4.6, with uniform cost search, that is, in each iteration it's going to choose the node which 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 4.5 it's described an improved heuristic, given that *manhattan distance* is not very efficient.

Starting by analysing the parent node, it's going to value 1 because it's 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 descendants are going to be expanded and all are going to have heuristic value 3 because 'A' is still misplaced and are at depth 2. Node 4 is going to generate a descendant which is the goal state and has value 3 because 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.

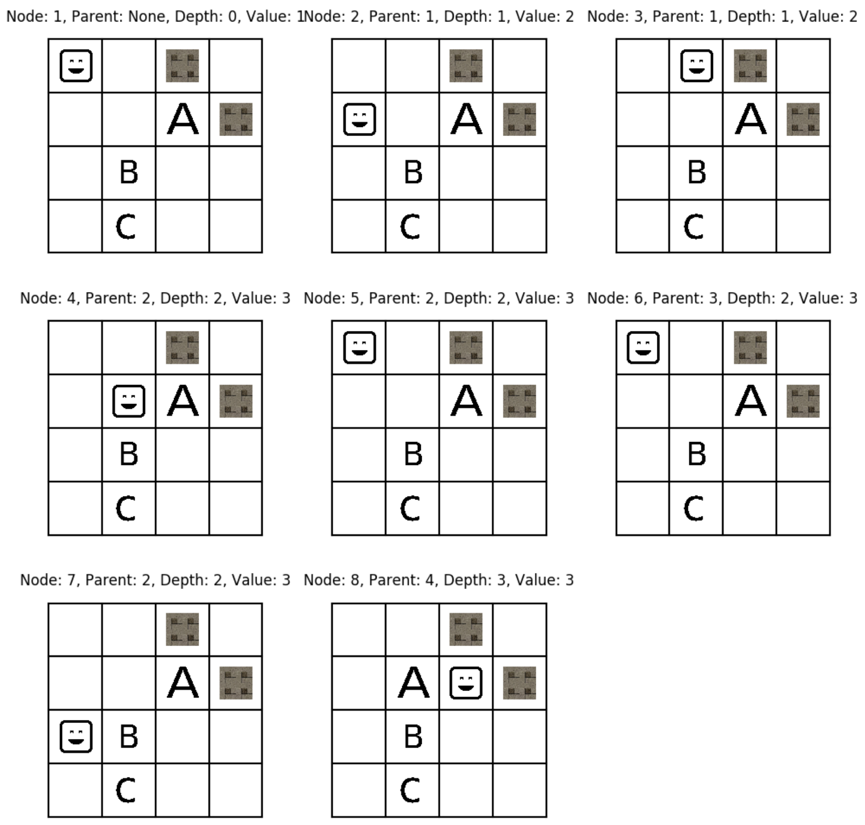

Node: 1, Parent: None, Depth: 0, Value: 1
Node: 2, Parent: 1, Depth: 1, Value: 2
Node: 3, Parent: 1, Depth: 1, Value: 2
Node: 4, Parent: 2, Depth: 2, Value: 3
Node: 5, Parent: 2, Depth: 2, Value: 3
Node: 6, Parent: 3, Depth: 2, Value: 3
Node: 7, Parent: 2, Depth: 2, Value: 3
Node: 8, Parent: 4, Depth: 3, Value: 3
A
B
C

Figure 8: A* visited nodes during search

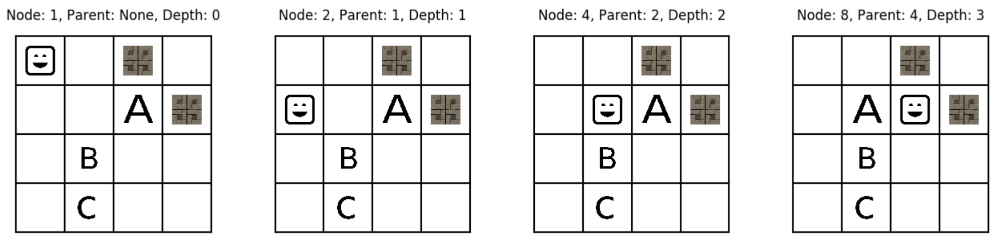

Node: 1, Parent: None, Depth: 0
Node: 2, Parent: 1, Depth: 1
Node: 4, Parent: 2, Depth: 2
Node: 8, Parent: 4, Depth: 3
A
B
C

Figure 9: Solution path for A* search

3 Scalability

In order to study the scalibity 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 descendants 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 shows how the number of nodes expanded changes with the difficulty of the problem. The y-axis is in logarithmic 10 scale, to allow a better analyse of the graph, given that the range of values is very big. Figure 11 represents the depth at which each search found a solution compared to the optimal depth.

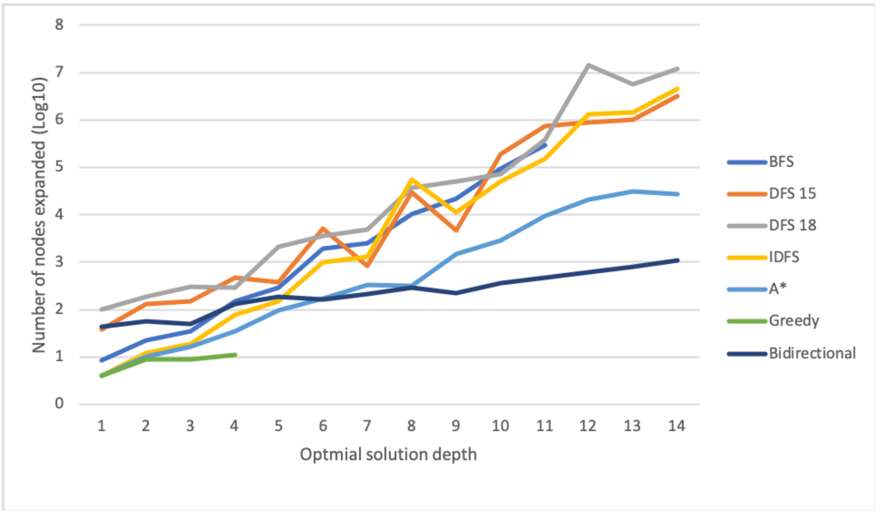


Figure 10: Number of nodes expanded for each search per depth

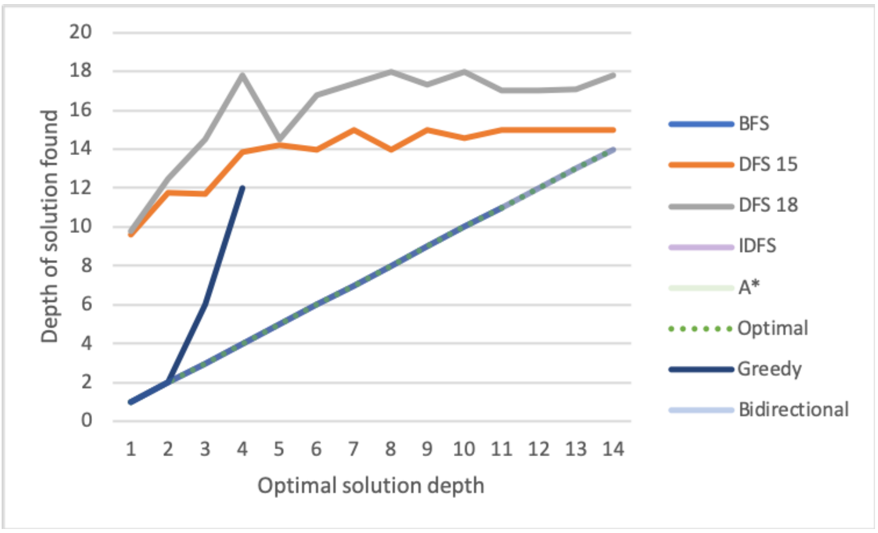


Figure 11: Depth of the solution found for each search

UNIVERSITY OF Southampton
School of Electronics and Computer Science

When analysing the graphs above we can take some conclusions about each search:

- **BFS** - Although in theory is complete, 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 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.

- **DFS** - For depth-first, two depth limits were tested: 15 and 18. This limits were imposed because DFS is impractical for infinite-depth spaces. Both searches were the ones that expanded the most nodes for almost all depths, however it was able to find solution everytime. This is due to the fact that the maximum depth is a relatively small number. The problem with this search is that it's not optimal as it can be seen in Figure 11.

- **IDFS** - Because it iterates throw all the depths, for bigger depths the number of nodes expanded can be slightly bigger than DFS, however it always finds the optimal solution.

- **A*** - Because it is an informed search it outperforms every uninformed search. Was the search with the least number of expanded nodes for each depth, it always found a solution, so it's complete, and the solution was always optimal.

- **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. Because the path cost in 1, this grants optimal solutions. When the heuristic gets improved A* can match this results, as it will be seen in section 4.5.

- **Greedy** - Because the *manhattan distance* heuristic isn't very good for this problem, when it comes to greedy, the performance is really bad. For lower depths, when it was able to find a solution was the best search. However when the depth got higher it would get stuck in a loop and not find a solution. Besides that, greedy is not optimal. In section 4.5 it's going to be discussed a new heuristic which makes greedy more viable.

Concluding, the best overall search was bidirectional search, however it also must be taken into account that the heuristic for A* was not very efficient. The greedy problem has proven to be the fastest algorithm when it can find a solution (for lower depths), however is not optimal.

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's 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-search.

Figure 13 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 initial state (node 1) and the search up starts with the goal state (node 2). Both are going to have two descendants, 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, which is what happens with node 19. This node was already visited in the search up: node 18, so the search stops in that iteration.

In Figure 12 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 because it's the same as the last node of the search down, so it starts by the penultimate node, which in this case is the node 8, and backtracks from there. The depth for these nodes is increased accordingly. Although the last node has depth 6, the real depth of the solution is 3, because 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 from the search down.

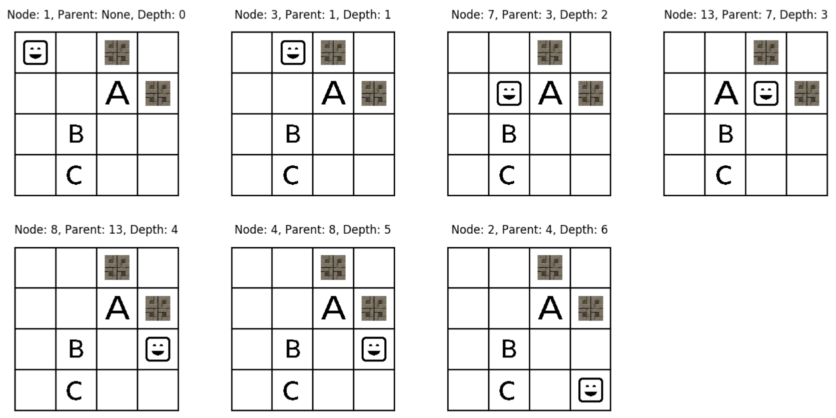

Node: 1, Parent: None, Depth: 0
Node: 3, Parent: 1, Depth: 1
Node: 7, Parent: 3, Depth: 2
Node: 13, Parent: 7, Depth: 3
Node: 8, Parent: 13, Depth: 4
Node: 4, Parent: 8, Depth: 5
Node: 2, Parent: 4, Depth: 6
A
B
C

Figure 12: Solution path for Bidirectional search

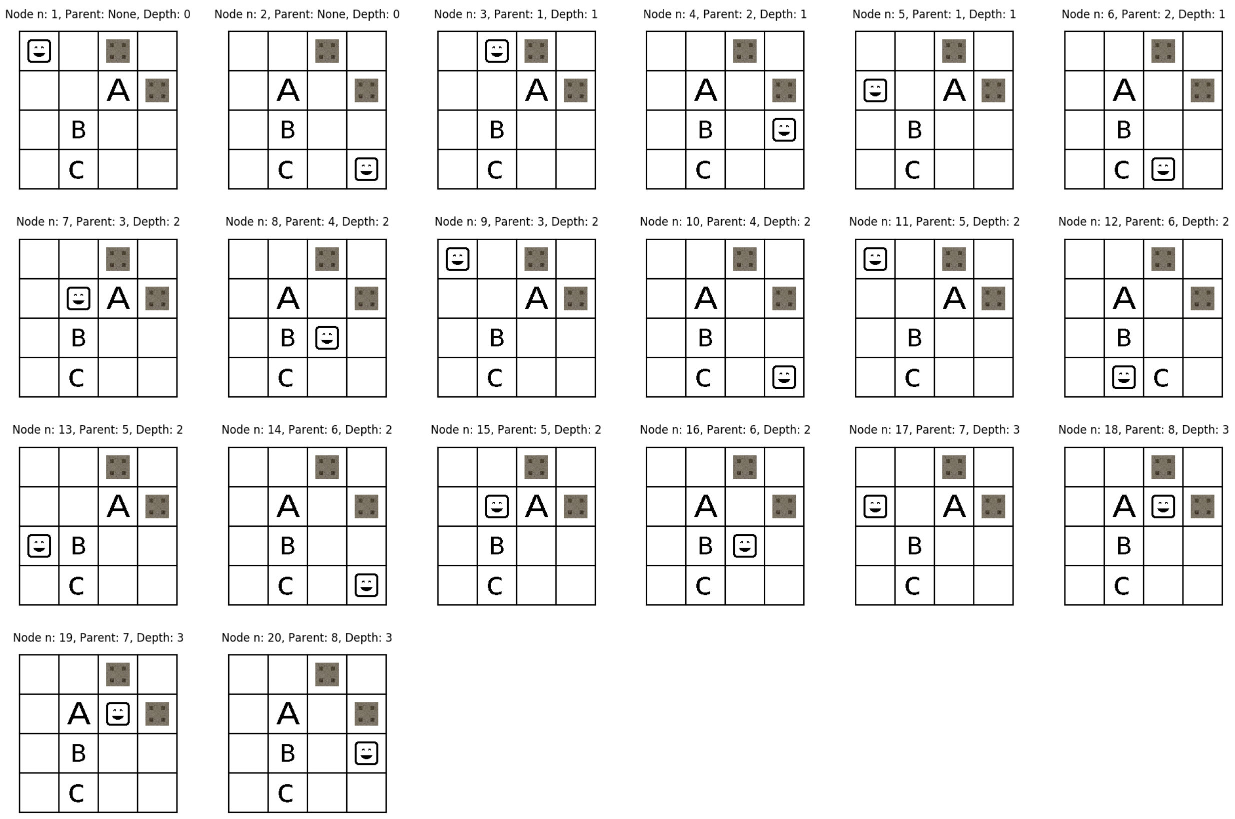

Node n: 1, Parent: None, Depth: 0
Node n: 2, Parent: None, Depth: 0
Node n: 3, Parent: 1, Depth: 1
Node n: 4, Parent: 2, Depth: 1
Node n: 5, Parent: 1, Depth: 1
Node n: 6, Parent: 2, Depth: 1
Node n: 7, Parent: 3, Depth: 2
Node n: 8, Parent: 4, Depth: 2
Node n: 9, Parent: 3, Depth: 2
Node n: 10, Parent: 4, Depth: 2
Node n: 11, Parent: 5, Depth: 2
Node n: 12, Parent: 6, Depth: 2
Node n: 13, Parent: 5, Depth: 2
Node n: 14, Parent: 6, Depth: 2
Node n: 15, Parent: 5, Depth: 2
Node n: 16, Parent: 6, Depth: 2
Node n: 17, Parent: 7, Depth: 3
Node n: 18, Parent: 8, Depth: 3
Node n: 19, Parent: 7, Depth: 3
Node n: 20, Parent: 8, Depth: 3
A
B
C

Figure 13: Bidirectional search visited nodes

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 in A*, greedy search only focuses on the heuristic value of the node, that is, the: f(n) = h(n).

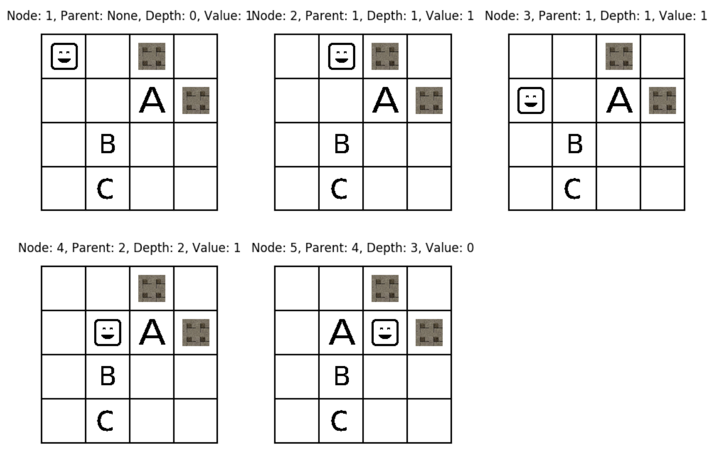

Node: 1, Parent: None, Depth: 0, Value: 1
Node: 2, Parent: 1, Depth: 1, Value: 1
Node: 3, Parent: 1, Depth: 1, Value: 1
Node: 4, Parent: 2, Depth: 2, Value: 1
Node: 5, Parent: 4, Depth: 3, Value: 0
A
B
C

Figure 14: Greedy search visited nodes

The search path is shown in the figure above and the heuristic used for this case it *manhattan distance*. The start node has evaluation value 1, because the only misplaced block is 'A'. This node is going to generate two descendants, whose evaluation value is still 1 because 'A' keeps out of place, so they are taken out of the priority queue in a random way. Node 2 is the first to go out and expands node 4. Because this previous node also has value 1, node 3 is visited first, followed by 4, which expands node 5, that is the next node to leave the priority queue because it has the lowest value, and corresponds to the goal state. The correspondent solution path is in Figure 15.

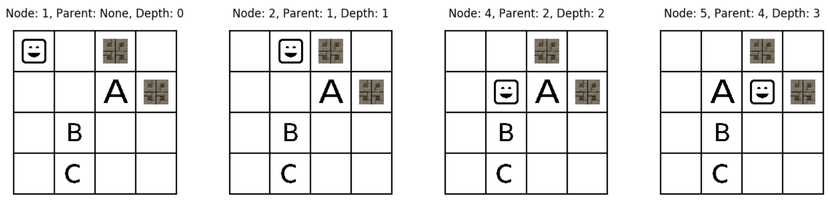

Node: 1, Parent: None, Depth: 0
Node: 2, Parent: 1, Depth: 1
Node: 4, Parent: 2, Depth: 2
Node: 5, Parent: 4, Depth: 3
A
B
C

Figure 15: Solution path for Greedy search

4.4 Graph Search

Graph search was implemented for BFS and A*. This was accomplished with the creation of a set, where the visited nodes would be in, and for each new node expanded it checks if it's already in the set, and if it is is not searched. This improves performance dramatically.

4.5 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. This is done by adding the *manhattan distance* from the agent to the most far 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.

In Figure 16 it can be seen 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 too interpretations: this problem is not good for greedy search, being one of the reasons, the fact that the position of the agent in the goal state doesn't matter, or better heuristics can be found.

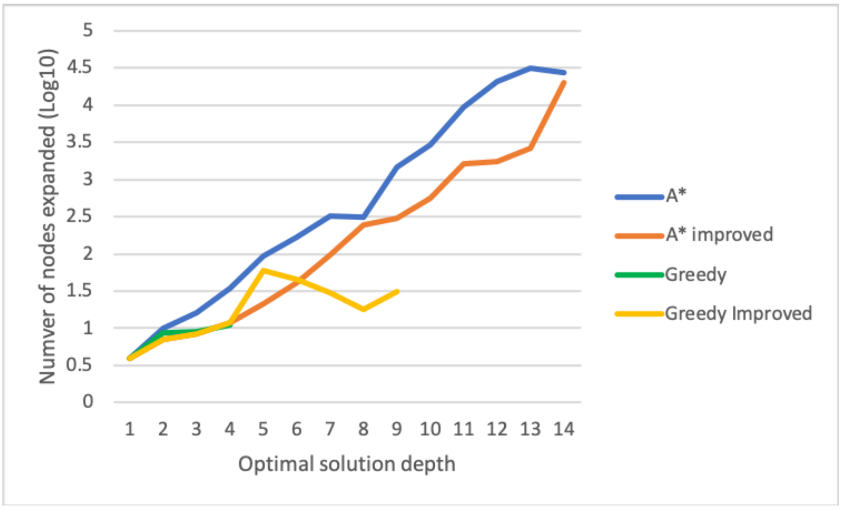


Figure 16: Number of nodes expanded for each heuristic

4.6 Improved get_descendants()

An improvement over the get_descendants() 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.

4.7 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 4.5, this can also be due to the fact that the greedy algorithm is not good for this particular problem.

List of Figures

List of Tables

References

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

[2] “Iterative deepening depth-first search,” Nov 2019.

5 Appendix A