COMP6231 - Comparing uninformed and heuristic search methods

Pier Paolo Ippolito
University of Southampton
ppi1u16@ecs.soton.ac.uk

December 12, 2019

1 Approach

In this report, are going to be detailed the research results obtained analysing and comparing different search algorithm used in order to solve the "Blocksworld tile puzzle" game by moving an agent (X) around a grid. In Figure 1, are shown the initial and final state of this problem.

$$\underset{\text{Initial State}}{\begin{bmatrix} - & - & - & - \\ - & - & - & - \\ - & - & - & - \\ A & B & C & X \end{bmatrix}} \quad \underset{\text{Final State}}{\begin{bmatrix} - & - & - & - \\ - & A & - & - \\ - & B & - & - \\ - & C & - & - \end{bmatrix}}$$

Figure 1: Blocksworld tile puzzle

Different tree searching techniques have been examined throughout this experiment such as: Breadth First Search (BFS), Depth First Search (DFS), Depth Limited Search (DLS), Iterative Deepening (ID) and their respective graph search equivalents. Additionally, an implementation of Bidirectional Search using two Breadth First Searchers starting from the initial and goal state will be provided. Of these different algorithms, just the Depth First Search implementations, the Iterative Deepening Graph Search and the Bidirectional Search proved to be not optimal (all the other implementations were able to find the shortest path to solve the problem).

In order to produce these results, Python has been chosen as preferred programming language. Throughout this experiment, different versions of the same code have been implemented in order to make sure the different algorithm were able to run as fast as possible and using the least possible amount of memory. As an example, a naive implementation of the Breath First Search algorithm took about 7 hours to correctly solve this task, while the latest code version was able to solve this same problem in under 8 minutes (eg. by making a better use of data structures, creating a custom Deep Copy function and replacing for loops with list comprehension). An exhaustive demonstration of example outputs with a graphical demonstrations for the different tree and graph search methods is available in Appendix A.

Additionally, has been created a simple user interface in which the user can interactively choose which algorithm to use to solve the game, input the preferred board size, decide if or not to add obstacles and even try to solve the game himself (a simple example of the user interface is available in Appendix B). Furthermore, as a demonstration that both the tree and graph searchers were working as expected, in Appendix F is available a graphical representation of the first two tree levels of the A Star Search methods implemented. All the code used in order to reproduce these searching techniques is available in Appendix G.

In order to examine how the time complexity varies increasing the problem size/difficulty different approaches have been taken such as: varying the number of moves away from the goal state, the board size and the number of blocks in the problem.

2 Evidence

In this section, we will examine how well/bad different uninformed search strategies such as BFS, DFS and Iterative Deepening can perform in solving this taks. Additionally, we will also compare them to an informed search strategy such as A Star. A collection of code outputs for these algorithms is available in Appendix A.

In order to implement these different algorithms, the "Artificial Intelligence: A Modern Approach" book [1] by Stuart Russel and Peter Norvig has been used as reference in conjunction with the course material and "Introduction to Algorithms" by Thomas H. Cormen et al. [2]. In Appendix C is additionally available a graphical representation of the steps taken by the agent in order to optimally solve this task.

The code used to create these algorithms has been divided into three main classes: Space, Game and MakeNode. The space class was used to create different grids in which the agent can move in, and was taking as input parameters the number of rows and columns we wanted our world to be formed of and a Boolean value indicating if we wanted to add or no obstacles in the grid. The Game class was instead inheriting the parameters from the Space class and was used to create the interactive graphical interface and to test if the different algorithms were working as expected. Finally, the MakeNode class was used to create new tree nodes and to store the state, parent, action, path depth and estimated cost associated with each node.

Each of the different tree algorithms, has then been created inside a single function which was taking as input parameter the initialised grid and the search mode the user wanted to use (eg. breath first, depth first, etc...). Depending on the mode selected it might have then been necessary to add other additional parameters (for example in Iterative Deepening was necessary to specify the mode and the maximum depth to reach). This same approach has also been used later on in order to create the graph search and bidirectional search methods (in other two separate functions).

All the tree search methods have been successfully tested using the start and goal states as shown in Figure 1.

2.1 Breadth First Search (BFS)

When using Breadth First Search, all the nodes are expanded at a given depth in the tree, before expanding any of the nodes at the next level. This can by implemented in Python by using a First-In-First-Out (FIFO) queue at the tree frontier. In this way, old nodes gets expanded first than newer nodes (which are deeper down in the tree).

The implementation results using BFS are shown in Listing 1. These results have been obtained by using Up, Left, Down, Right as nodes order expansion. Changing the order of these four operations would alternatively lead to different time complexities needed in order to solve the problem. In all the cases, BFS has proved to be able to always find the optimal path to the solution.

```
Scored Computational Time: 5251318
node Depth to reach goal state: 14
Estimated Path Cost: 14
Moves used to reach goal state ( 14 ) :
Root , Up , Left , Left , Down , Left , Up , Right , Down , Right , Up , Up , Left , Down , Left
```

Listing 1: Breadth First Search Solution

2.2 Depth First Search (DFS)

In Depth First Search, we aim instead to expand first the deepest node in the frontier (until there are no more successors available). In order to recreate this behaviour in Python, was made use of a Last-In-First-Out (LIFO) queue (expending always the most recently generated nodes). One of the main problems associated with DFS, is that it is not complete (it might get stuck in an infinite loop). This can be fixed by using instead DFS graph implementation (as shown in the "Extras and limitations" section).

In Listing 2, are shown the result from DFS. In this case, the algorithm was able to solve the task but with a not-optimal solution. It has been necessary to run multiple times the method to record these results, as the algorithm was at times getting stuck in infinite loops (making exhaust all the memory available). In this case, the order of node expansion used in DFS has been randomised.

```
Scored Computational Time: 694
node Depth to reach goal state: 693
Estimated Path Cost: 693
Moves used to reach goal state ( 693 ) :
Root , Up , Up , Down , Up , Up , Down , Down , Down , Up , Left , Up , Up , Down , Right , ...
```

Listing 2: Depth First Search Solution

2.3 Iterative Deepening (ID)

When using Iterative Deepening, we aim to find the best depth limit (this is done by incrementally increasing the depth limit until a solution is found). In this way, we can be able to solve the problem by scoring a similar time complexity than BFS but reducing the space complexity needed. Also this time (as shown in Listing 3), Iterative Deepening has been proven to always find the optimal solution. In this occasion, I implemented this algorithm in Python by recursively calling the DFS function and setting a stop limit which is incremented by one every time the function is called (this time not randomising the order of node expansion).

```
Scored Computational Time: 12227545
node Depth to reach goal state: 14
Estimated Path Cost: 14
Moves used to reach goal state ( 14 ) :
Root , Up , Left , Left , Down , Left , Up , Right , Down , Right , Up , Up , Left , Down , Left
```

Listing 3: Iterative Deepening Solution

2.4 A Star Search

A Star Search is a type of informed search strategy which makes use of an evaluation function in order to considerably reduce the search space (Equation 1). In the evaluation function, the ***g(n)*** represent the cost accumulated so far to reach a node and ***h(n)*** represents the cost estimated using an heuristic to move from the current node to the end goal. In this case, the Manhattan distance (Equation 2) has been used as the heuristic of choice. I decided to make use of the Manhattan distance as heuristic, because it does never overestimate the cost to reach the goal, making A Star Tree search always optimal.

This heuristic has been implemented in Python by converting the current world state and the goal state into a one dimensional list, comparing the indices positions of the different respective letters in the world and taking the absolute value for each of them.

$$EvaluationFunction = g(n) + h(n) \tag{1}$$

$$ManhattanDistance = \sum_{i=1}^{n} |x[i] - y[i]| \tag{2}$$

The results obtained using A Star, are available in Listing 4. Additionally, as further evidence of the A Star method working, is available in Appendix F a graphical representation of the first two levels of the A Star tree and graph search.

```
Scored Computational Time: 2989
node Depth to reach goal state: 14
Estimated Path Cost: 14
Moves used to reach goal state ( 14 ) :
Root , Up , Left , Left , Down , Left , Up , Right , Down , Right , Up , Up , Left , Down , Left
```

Listing 4: A Star Search Solution

3 Scalability

Different approaches have been taken in order to examine how each of these different algorithms performs when varying the difficulty of the task. In this section, we will examine how increasing the node depth (the number of moves the agent is far from the goal state) affects the number of expanded nodes. In the "Extras and limitations" section, we will additionally also explore how changing the size of the board and the number of blocks in the problem can affect the scalability of the graph search methods.

In Figure 2, is examined the time complexity of Breadth First Search, Depth Fist Search, Iterative Deepening and A Star (in this graph the Y axis is in logarithmic scale to clearly show the differences between the different search methods).

The results obtained for DFS cannot be considered comparable with the other search methods due to the fact that this algorithm is not guaranteed to find the optimal solution (shortest path), therefore the Node Depth can't be controlled in this case. Additionally, due to the nature of this algorithm, has been necessary to run multiple times this method in order to obtained these graphs. In some cases, the execution of the algorithm had additionally to be stopped because of memory limitations and infinite loops, therefore, Figure 2 shows a quite optimistic summary of the time complexity taken by DFS to solve this task.

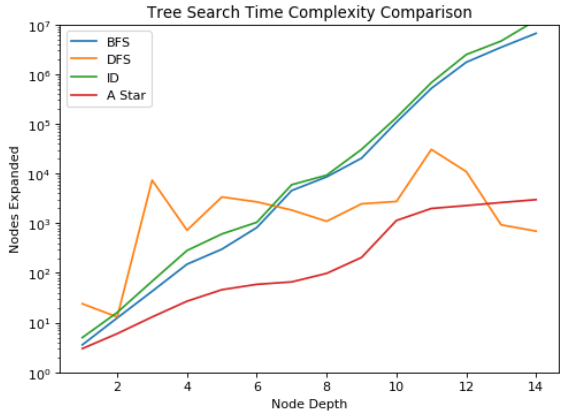


Figure 2: Tree Search Time Complexity Comparison

As we can clearly see form Figure 2, A Star is the algorithm which performed best in solving this task with the least time complexity (thanks to the Manhattan Distance Heuristic). In addition to this, we can also see that Iterative Deepening scored almost the same time complexity as Breath First Search to solve the problem. In the case of Breadth First Search, it was necessary to run this experiment multiple times, changing averytime the order of nodes expansion to take an average of the time complexity.

This same analysis performed instead on BFS, DFS, ID, A Star graph search equivalents is available in the "Extras and limitations" section. Graph Search algorithms have been implemented in Python by creating a list storing all the visited world states and when expanding a node checking if any of the new states is already present in the visited list (in this way avoiding to visit a same state twice).

In Table 1, are summarised the main characteristics of the Uninformed Search methods covered in this section. For both Time and Space complexity has been made use of the Big O notation and of the following abbreviations:

- **b** = maximum branching factor.
- **d** = least cost solution depth.
- **m** = maximum state space depth.

Criterion	Breadth-First	Depth-First	Iterative Deepening
Worst Time	O(b^d)	O(b^m)	O(b^d)
Worst Space	O(b^d)	O(bm)	O(b^{bd})
Complete	Yes (if b finite)	No	Yes
Optimal	Yes (if cost=1 per step)	No	Yes (if cost=1 per step)

Table 1: Measuring problem-solving performance

A Star search has not been included in the Table 1 since it is an Heuristic Search and using different types of heuristics would lead to different time and space complexities. However, A Star has in general the following properties: the computational time is exponential in length of the optimal solution, in terms of space complexity it keeps all the expanded node in memory, it is complete and optimal.

The results obtained in this experiment perfectly matched with the summary in Table 1. In fact, Breadth First, Iterative Deepening and A Star demonstrated to be optimal and complete (as shown in the "Evidence" section) while Depth First not (eg. presence of infinite loops). Additionally, from the results obtained in Figure 2, we can see that as expected Breadth First and Iterative Deepening scored a similar time complexity when varying the problem difficulty (just outperformed by A Star thanks to the use of the Manhattan Distance as heuristic).

Overall, from this experiment we can clearly see that A Star demonstrated to be the search method which was best able to scale for different size problems while Depth Fist was the algorithm which had most problems in this ambit. One reason why Depth First, demonstrated to be not a suitable choice to solve this problem is that the maximum state space depth is larger than the least cost solution depth (**m** is larger than **d**). One of the main advantages although of using Depth First Search is his polynomial space complexity.

4 Extras and limitations

At completion of this project, different extras have been realised. Some examples are:

- Graph Search (BFS, DFS, Depth Limited Search, Iterative Deepening, A Star).
- Bidirectional Search (BFS-BFS).
- Varied Board Size.
- Varied Number of blocks in the game.
- Inclusion of obstacles in the world.
- User interface to let an user solve the different problems on their own.

In order to create the different graph search implementations, a list of the visited states has been created so to avoid to make the algorithm visit the same state twice. In this section, are provided three different methods to examine the scalability of the different graph search techniques: varying the number of moves away from the goal state, the number of blocks in the problem and the board size.

4.1 Varying the number of moves away from the goal state

As we can see from Figure 3, using graphs methods can considerably reduce the number of nodes expanded. In this case, also the Iterative Deepening graph search version demonstrated to not be optimal (this can be fixed by specifying in the algorithm to add in the visited list always the shortest path in case of any conflict). Additionally, on this graph has also been plotted the time complexity results scored using Depth Limited Graph Search with a limit of of 50 for node depth. As shown in Figure 3, also in this case A Star demonstrated to be the algorithm which was best able to cope with different problem sizes.

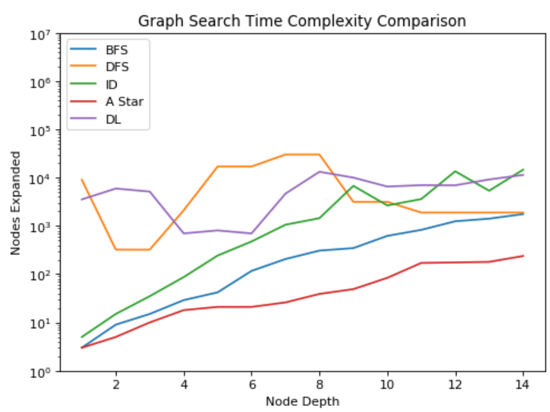


Figure 3: Graph Search Time Complexity Comparison

4.2 Varying the number of blocks in the problem and the board size

In this subsection, we will now examine how increasing iteratively the number of tiles of in the grid (Figure 4 (a)) and the board size (Figure 4 (b)) will increase the time complexity of the different algorithms implemented.

The time complexity registered using these two different approaches are shown respectively in Figure 5 (a) and (b). Also in this case, Depth First Search demonstrated to be the algorithm which performed less well when increasing the problem size. A Star instead demonstrated again to be the algorithm which was able to best scale.

In addition to this, in Appendix E (Graph Search problem difficulty with and without obstacles) is also available an analysis of how adding obstacles in the grid can affect the problem difficulty and time complexity needed in order to solve the Blocksworld tile puzzle in Graph and Bidirectional Search. Another example output demonstrating the use of obstacles in the grid is also available in Appendix B (Example Output: User Interface).

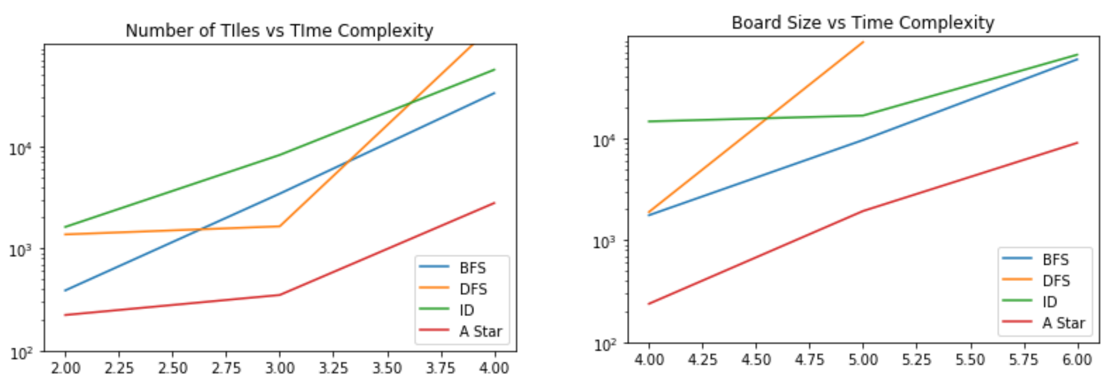
```
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['D', 'C', '-', '-'],
 ['A', 'B', '-', 'x']]
[['-', 'D', '-', '-'],
 ['-', 'A', '-', '-'],
 ['-', 'B', '-', '-'],
 ['-', 'C', '-', '-']]
```

(a) Tiles Positions Example (Start and Goal State)

```
Board Size:  6 x 6
[['-', '-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '-', '-'],
 ['A', 'B', 'C', '-', '-', 'x']]
[['-', '-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '-', '-'],
 ['-', '-', '-', 'A', '-', '-'],
 ['-', '-', '-', 'B', '-', '-'],
 ['-', '-', '-', 'C', '-', '-']]
```

(b) Increased Board Size Example (Start and Goal State)

Figure 4: Varying complexity examples



(a) Varying Number of Tiles Time Complexity

(b) Varying Board Size Time Complexity

Figure 5: Time Complexity Varying problem difficulty

4.3 Bidirectional Search and Further Developments

Bidirectional Search was implemented by running simultaneously two different searchers (one starting from the start state and one starting at the goal state). In both searchers, a list of the visited states is stored and the search terminates if the frontiers of the two searches intersect (meaning that the two searchers met in the middle). The first intersection between the two searchers might although not be optimal, requiring therefore to do some additional search to check if there is any other short-cut across the gap. In this implementation, I decided to use two different Breadth First Searchers as searching algorithms. Examples of use of Bidirectional Search are available in Appendix A (A.10), Appendix D (D.6) and Appendix E.

Overall, this project had a successful outcome providing multiple insights about the different searching methods and in the comparison between the results expected from the theory and the actual ones from implementations. Although, some additional features in order to enhance this analysis can still potentially be added. Some examples of further advancements which can be added to this project are: space complexity analysis, use of a better heuristic for A Star and improving some of the graph search methods which were not optimal to become optimal by adding in the visited list always the shortest path in case of any conflict.

References

[1] Artificial Intelligence: A Modern Approach (Third Edition). Stuart Russel and Peter Norvig. Accessed at: *https://www.cin.ufpe.br/ tfl2/artificial-intelligence-modern-approach.9780131038059.25368.pdf*, Nov 2019.

[2] Introduction to Algorithms (Third Edition). Thomas H. Cormen et al. Accessed at: *http://kddlab.zjgsu.edu.cn:7200/students/lipengcheng/%E7%AE%97%E6%B3%95%E5%AF%BC%E8%AE%BA%EF%BC%88%E8%8B%B1%E6%96%87%E7%AC%AC%E4%B8%89%E7%89%88%EF%BC%89.pdf*, Nov 2019.

Appendix A Example Output: Tree & Graph Algorithms in action

In this section are provided example code outputs of the different tree algorithm in action. In the case of Breadth First search and A Star (Figure 2 and 3), is highlighted the difference between the two algorithms when trying to solve the Blocksworld tile puzzle. In the first case (Figure 2), is given equal importance to each branch of the tree, in the second case (Figure 3) the algorithm instead start early to focus on the branch which will most likely lead to the optimal solution. In all the other examples, will be instead shown the tree structure used to solve a simple problem (Figure 1), in this way it will be possible to show the full tree from the start state to the goal state. All these graphs have been created by storing the relevant results of each algorithm in a dictionary and plotting the results using the networkx Python library.

$$\underset{\text{Initial State}}{\begin{bmatrix} - & X & - & - \\ - & - & A & - \\ - & B & - & - \\ - & C & - & - \end{bmatrix}} \quad \underset{\text{Final State}}{\begin{bmatrix} - & - & - & - \\ - & A & - & - \\ - & B & - & - \\ - & C & - & - \end{bmatrix}}$$

Figure 1: Simple Grid

A.1 Tree Breadth First Search

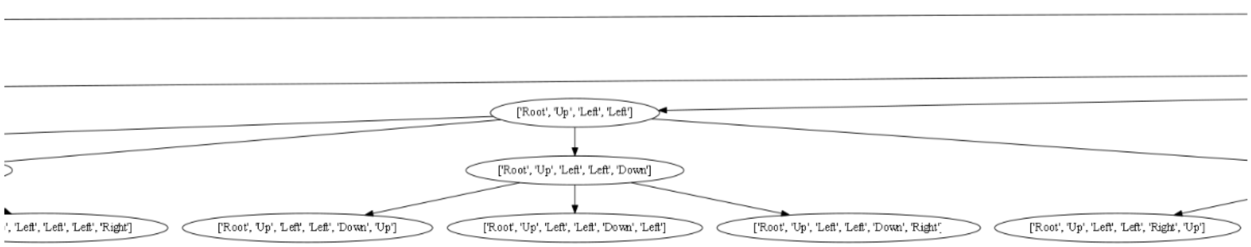


Figure 2: Tree Breadth First Search

A.2 Tree A Star Search

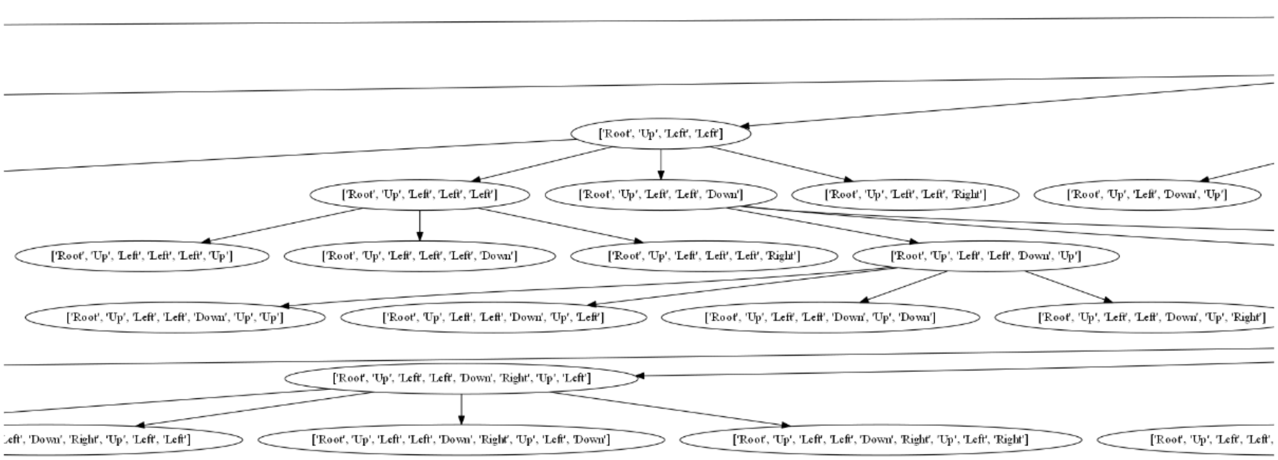


Figure 3: Tree A Star Search

A.3 Tree Depth First Search

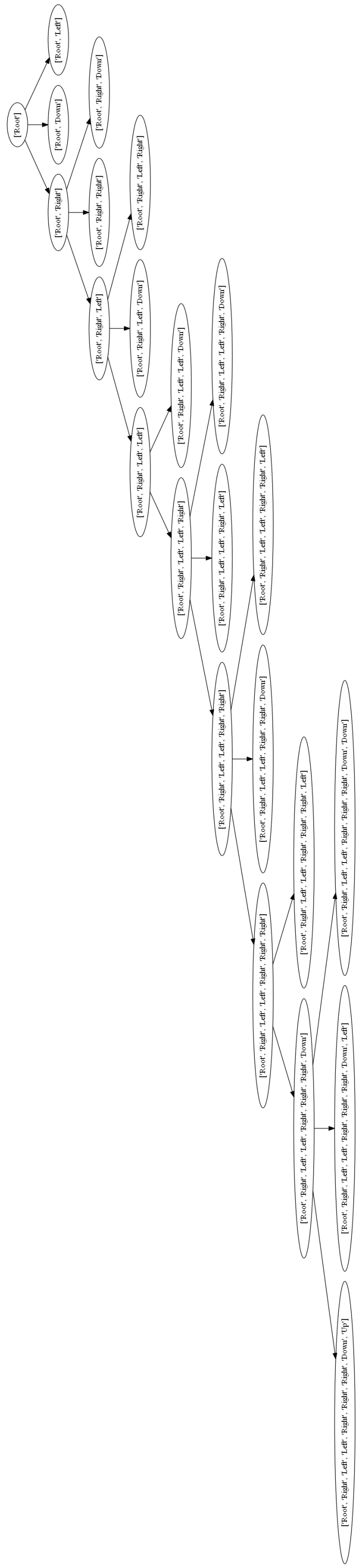


Figure 4: Tree Depth First Search

A.4 Tree Iterative Deepening Search

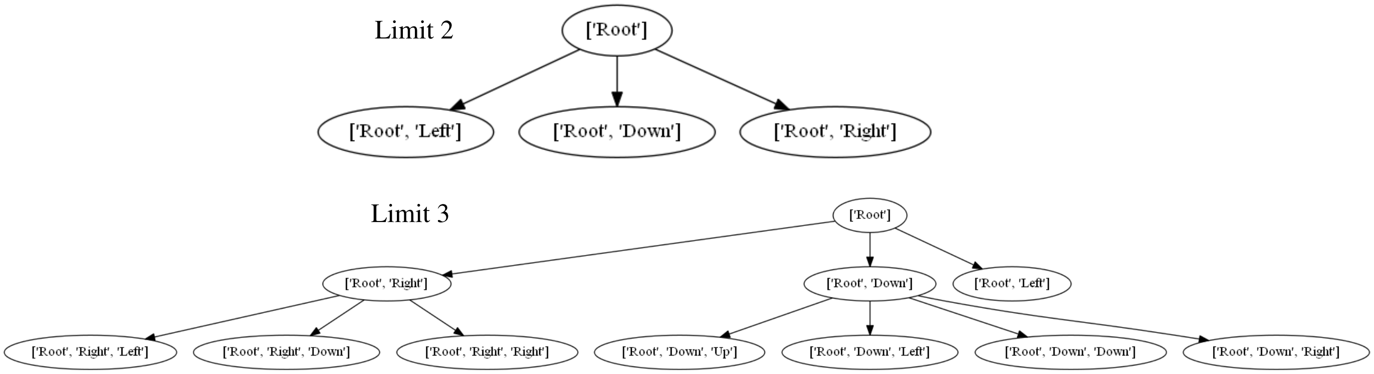


Figure 5: Tree Iterative Deepening Search

A.5 Graph Breadth First Search

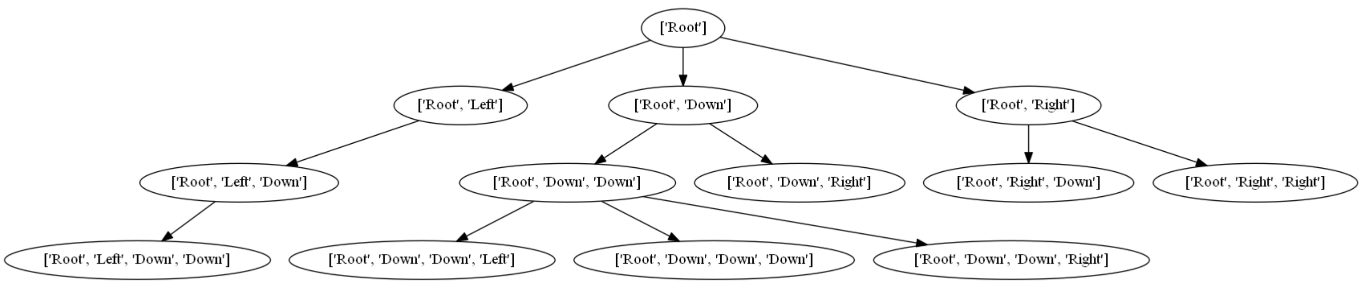


Figure 6: Graph Breadth First Search

A.6 Graph Depth First Search

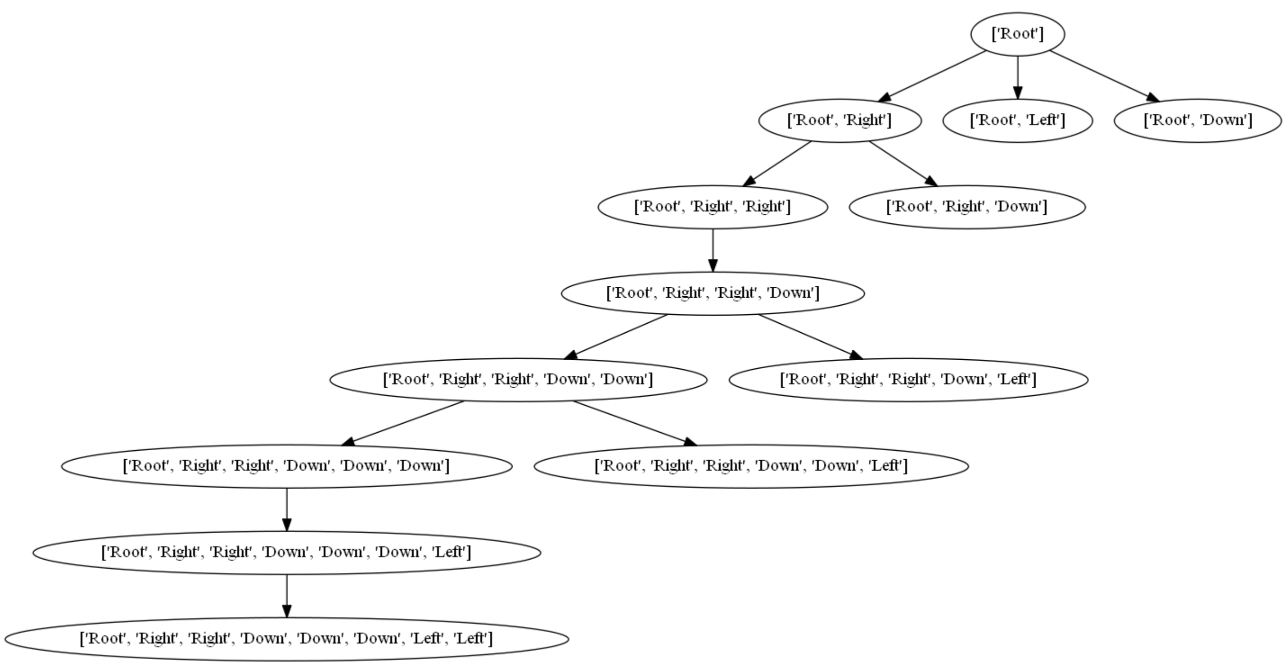


Figure 7: Graph Depth First Search

A.7 Graph Depth Limited Search

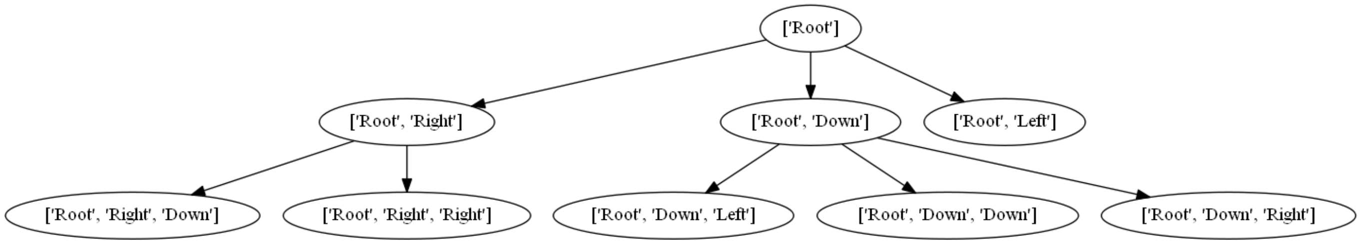


Figure 8: Graph Depth Limited Search

A.8 Graph Iterative Deepening Search

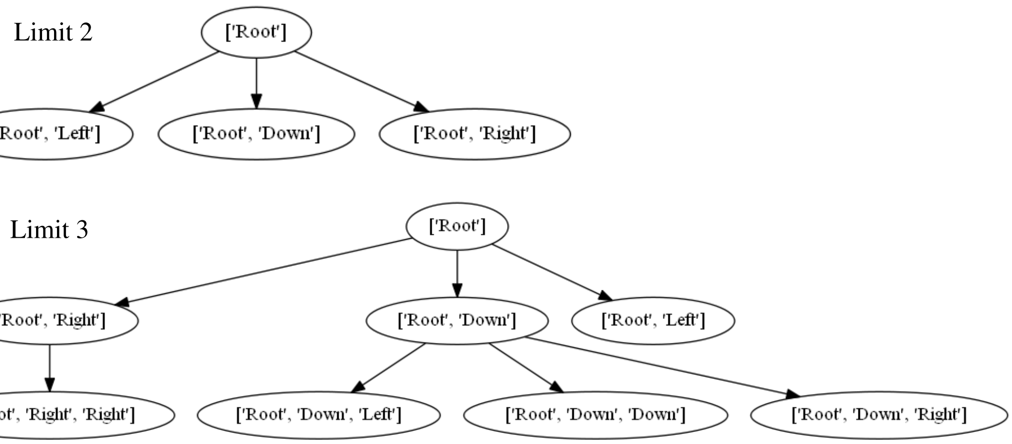


Figure 9: Graph Iterative Deepening Search

A.9 Graph A Star Search

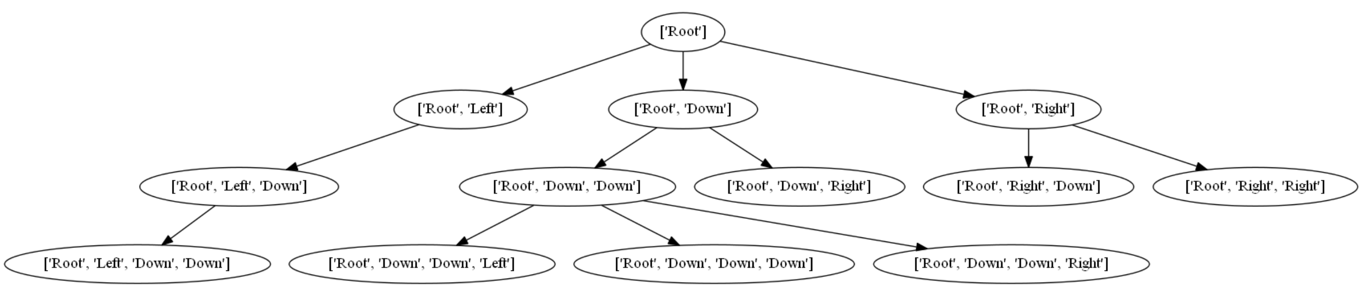


Figure 10: Graph A Star Search

A.10 Bidirectional Search

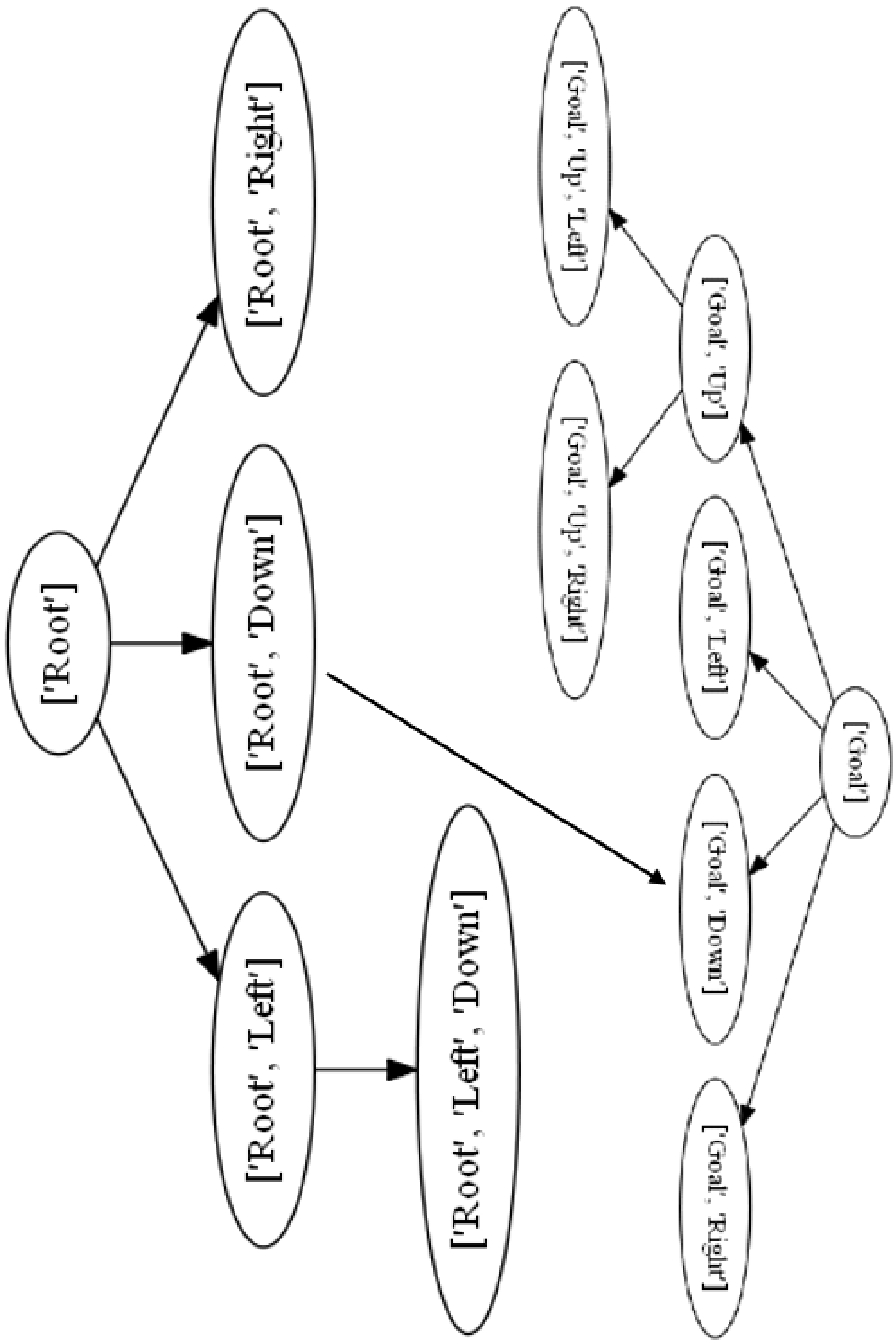


Figure 11: Bidirectional Search

Appendix B Example Output: User Interface

```
Welcome to my Blocksworld tile puzzle solver
What game matrix size do you prefer? (minimum 4*4)
Number of rows: 5
Number of columns: 5
Do you want to use Tree (0), Graph Search (1) or Bilateral Search (2)?
1
Do you want to add obstacles in the world? (yes/no)yes
Do you want to play the game yourself? (yes/no)
yes
Use a to move left, d to move right, w to move up, z to move down and e to escape
    the game
[['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', 'x']]
Choose Direction: w
[['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '%', 'x'],
 ['-', 'A', 'B', 'C', '-']]
Choose Direction: w
[['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', 'x'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', '-']]
Choose Direction: a
[['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', 'x', '-'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', '-']]
Choose Direction: z
Invalid move, try again
Choose Direction: d
[['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', 'x'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', '-']]
Choose Direction: e
End of the game!
Which algorithm do you want to use to solve the Blocksworld tile puzzle?
 0: Breadth First Search
 1: Depth First Search or Limited Depth Search
 2: Iterative Deepening
 3: A Star

3
[['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', 'x']]
[['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', 'A', '-', '-'],
 ['-', '-', 'B', '-', '-'],
 ['-', '-', 'C', '-', '-']]
Nodes Generated (Space Complexity: 721 )
```

```
Scored Computational Time: 494
Node Depth to reach goal state: 18
Estimated Path Cost: 18
Moves used to reach goal state ( 18 ) :
 Root, Up, Up, Left, Left, Down, Down, Left, Up, Right, Down, Right, Right, Up, Up, Left, Left, Down, Left
Graphical representation of moves:
[0,
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', 'x'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '%', 'x'],
 ['-', 'A', 'B', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', 'x'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', 'x', '-'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', 'x', '-', '-'],
 ['-', '-', '-', '%', '-'],
 ['-', 'A', 'B', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', 'x', '%', '-'],
 ['-', 'A', 'B', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', 'B', '%', '-'],
 ['-', 'A', 'x', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', 'B', '%', '-'],
 ['-', 'x', 'A', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', 'x', 'B', '%', '-'],
 ['-', '-', 'A', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', 'B', 'x', '%', '-'],
 ['-', '-', 'A', 'C', '-'],
 ['-', '-', '-', '-', '-'],
 ['-', '-', '-', '-', '%'],
 ['-', '-', '-', '-', '-'],
 ['-', 'B', 'A', '%', '-'],
 ['-', '-', 'x', 'C', '-'],
 ['-', '-', '-', '-', '-'],
```

```
['-', '-', '-', '-', '%'],
['-', '-', '-', '-', '-'],
['-', 'B', 'A', '%', '-'],
['-', '-', 'C', 'x', '-'],
['-', '-', '-', '-', '-'],
['-', '-', '-', '-', '%'],
['-', '-', '-', '-', '-'],
['-', 'B', 'A', '%', '-'],
['-', '-', 'C', '-', 'x'],
['-', '-', '-', '-', '-'],
['-', '-', '-', '-', '%'],
['-', '-', '-', '-', '-'],
['-', 'B', 'A', '%', 'x'],
['-', '-', 'C', '-', '-'],
['-', '-', '-', '-', '-'],
['-', '-', '-', '-', '%'],
['-', '-', '-', '-', 'x'],
['-', 'B', 'A', '%', '-'],
['-', '-', 'C', '-', '-'],
['-', '-', '-', '-', '-'],
['-', '-', '-', '-', '%'],
['-', '-', '-', 'x', '-'],
['-', 'B', 'A', '%', '-'],
['-', '-', 'C', '-', '-'],
['-', '-', '-', '-', '-'],
['-', '-', '-', '-', '%'],
['-', '-', 'x', '-', '-'],
['-', 'B', 'A', '%', '-'],
['-', '-', 'C', '-', '-'],
['-', '-', '-', '-', '-'],
['-', '-', '-', '-', '%'],
['-', '-', 'A', '-', '-'],
['-', 'B', 'x', '%', '-'],
['-', '-', 'C', '-', '-'],
['-', '-', '-', '-', '-'],
['-', '-', '-', '-', '%'],
['-', '-', 'A', '-', '-'],
['-', 'x', 'B', '%', '-'],
['-', '-', 'C', '-', '-']]
```

Listing 5: Graphical User Interface

Appendix C Example Output: Optimal Solution

```
Graphical representation of moves:
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', 'x'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', 'x'],
 ['A', 'B', 'C', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', 'x', '-'],
 ['A', 'B', 'C', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', 'x', '-', '-'],
 ['A', 'B', 'C', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', 'B', '-', '-'],
 ['A', 'x', 'C', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', 'B', '-', '-'],
 ['x', 'A', 'C', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['x', 'B', '-', '-'],
 ['-', 'A', 'C', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['B', 'x', '-', '-'],
 ['-', 'A', 'C', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['B', 'A', '-', '-'],
 ['-', 'x', 'C', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['B', 'A', '-', '-'],
 ['-', 'C', 'x', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['B', 'A', 'x', '-'],
 ['-', 'C', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', 'x', '-'],
 ['B', 'A', '-', '-'],
 ['-', 'C', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', 'x', '-', '-'],
 ['B', 'A', '-', '-'],
 ['-', 'C', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', 'A', '-', '-'],
 ['B', 'x', '-', '-'],
 ['-', 'C', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', 'A', '-', '-'],
 ['x', 'B', '-', '-'],
 ['-', 'C', '-', '-']]
```

Listing 6: Graphical Representation of the optimal path to reach the goal state

Appendix D Graph Search evidences

In this section, are provided evidences of the graph search methods running (in the same way as it was done in the "Evidence" section for the Tree search methods). The shortest action sequence for all the complete methods has also been provided. Also in this case, the start and goal state are the ones represented in Figure 1.

D.1 Graph Breadth First Search

```
Scored Computational Time: 1757
Node Depth to reach goal state: 14
Estimated Path Cost: 14
Moves used to reach goal state ( 14 ) :
Root, Up, Left, Left, Down, Left, Up, Right, Down, Right, Up, Up, Left, Down, Left
```

Listing 7: Graph Breath First Search

D.2 Graph Depth First Search

```
Scored Computational Time: 1890
Node Depth to reach goal state: 1836
Estimated Path Cost: 1836
Moves used to reach goal state ( 1836 ) :
Root, Left, Left, Left, Up, Right, Right, Right, Down, Left, Left, Left, Up, Up,
Right, Right, ...
```

Listing 8: Graph Depth First Search

D.3 Graph Depth Limited Search

```
Scored Computational Time: 11385
Node Depth to reach goal state: 49
Estimated Path Cost: 49
Moves used to reach goal state ( 49 ) :
Root, Left, Left, Left, Up, Right, Right, Right, Down, Left, Left, Left, Up, Up,
Right, Right, Right, Down, Down, Left, Up, Right, Down, Left, Up, Right, Up, Left,
Left, Down, Right, Up, Left, Down, Right, Right, Up, Left, Up, Left, Down, Down,
Right, Down, Left, Up, Right, Up, Left, Up
```

Listing 9: Graph Depth Limited Search

D.4 Graph Iterative Deepening Search

```
Scored Computational Time: 14595
Node Depth to reach goal state: 20
Estimated Path Cost: 20
Moves used to reach goal state ( 20 ) :
Root, Left, Left, Up, Left, Down, Right, Right, Right, Up, Left, Left, Down, Right
,Up, Left, Left, Up, Right, Down, Right
```

Listing 10: Graph Iterative Deepening Search

D.5 Graph A Star Search

```
Scored Computational Time: 238
Node Depth to reach goal state: 14
Estimated Path Cost: 14
Moves used to reach goal state ( 14 ) :
Root, Up, Left, Left, Down, Left, Up, Right, Down, Right, Up, Up, Left, Down, Left
```

Listing 11: Graph A Star Search

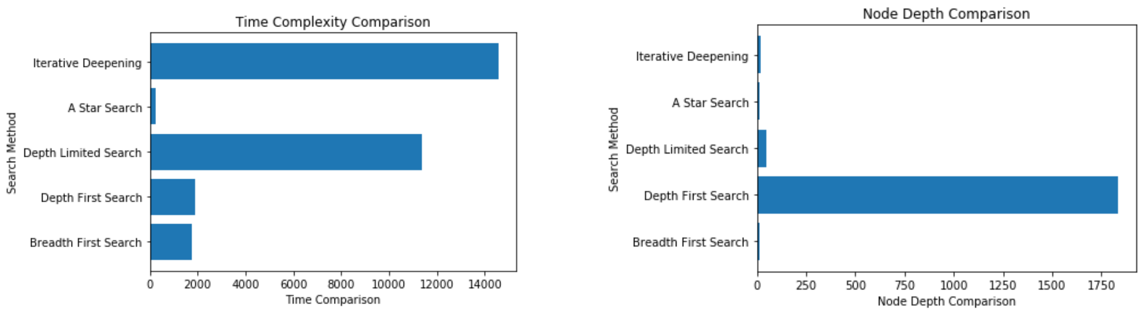
D.6 Bidirectional Search

```
Scored Computational Time: 4812
Node Depth to reach goal state: 30
Estimated Path Cost: 30
Moves used to reach goal state ( 30 ) :
Root , Up , Left , Left , Down , Left , Up , Right , Down , Right , Up , Left , Left , Down ,
Right , Up , Left , Down Down , Down , Left , Left , Up , Right , Up , Right , Down , Down ,
Left , Left , Up ,
```

Listing 12: Bidirectional Search

Appendix E Graph Search problem difficulty with and without obstacles

In Figure 12, are shown the time complexity and the node depth required by the different algorithms in order to solve the Blocksworld tile puzzle.



(a) Time Complexity

(b) Node depth to find the solution

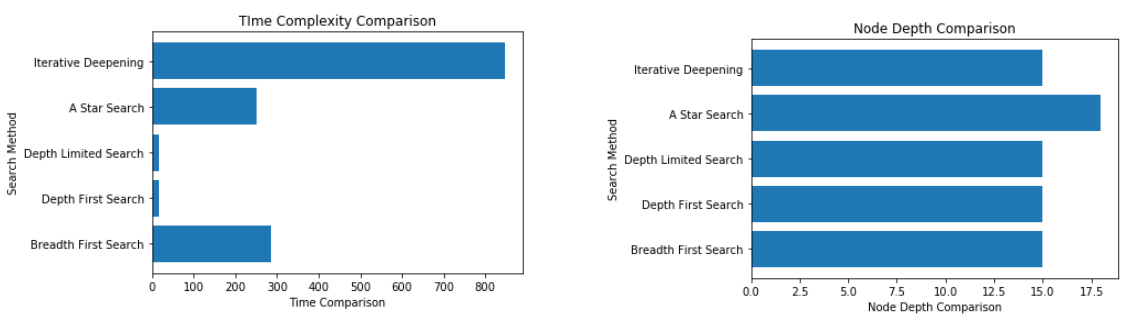
Figure 12: Graph Search to solve Blocksworld tile puzzle without obstacles

Adding obstacles in the world (Figure 13) led instead to the results shown in Figure 14. Therefore, in this example adding obstacles in the grid made overall easier for the different graph algorithms to solve this problem.

Initial State with Obstacles

$$\begin{bmatrix} - & - & - & \% \\ - & - & - & - \\ - & - & \% & - \\ A & B & C & X \end{bmatrix}$$

Figure 13: Blocksworld tile puzzle with Obstacles

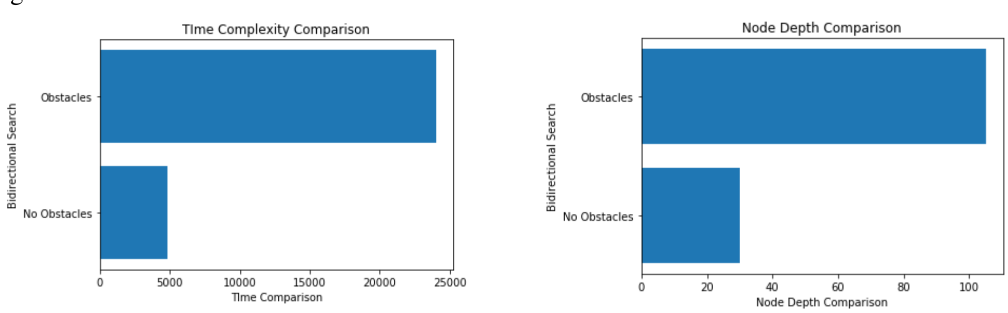


(a) Time Complexity

(b) Node depth to find the solution

Figure 14: Graph Search to solve Blocksworld tile puzzle with obstacles

Trying to solve this same problem, with and without obstacles, using Bidirectional Search led instead to the results shown in Figure 15.



(a) Time Complexity

(b) Node depth to find the solution

Figure 15: Bidirectional Search to solve Blocksworld tile puzzle with and without obstacles

Appendix F Example Output: A Star

Figure 16 and 17 have been realised by taking the code output shown respectively in Figure 18 (a) and (b) and rearranging it in a tree like structure. This example output has been created by taking the first two levels of the tree used by A Star in order to solve the Blocksworld tile puzzle.

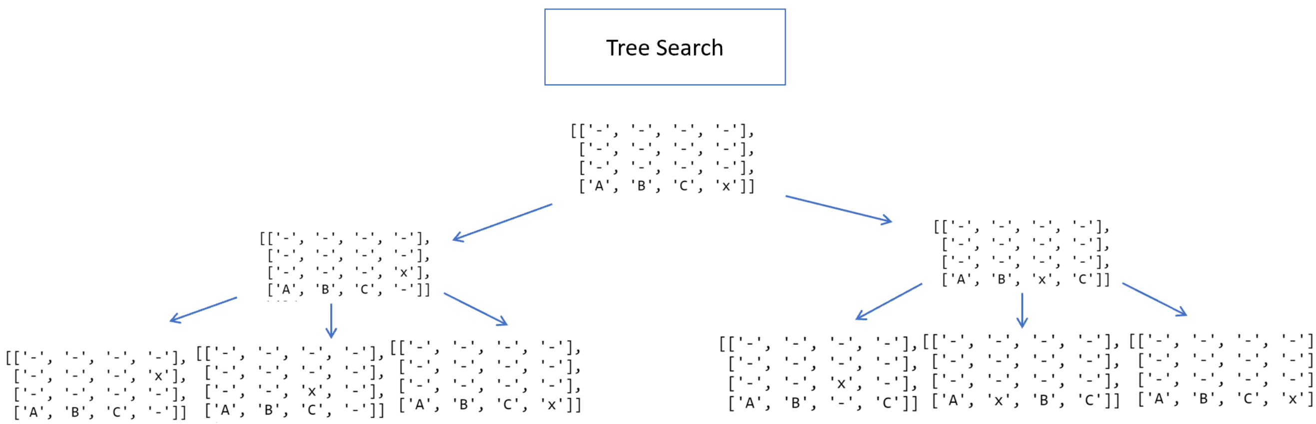


Figure 16: A Star Tree Search

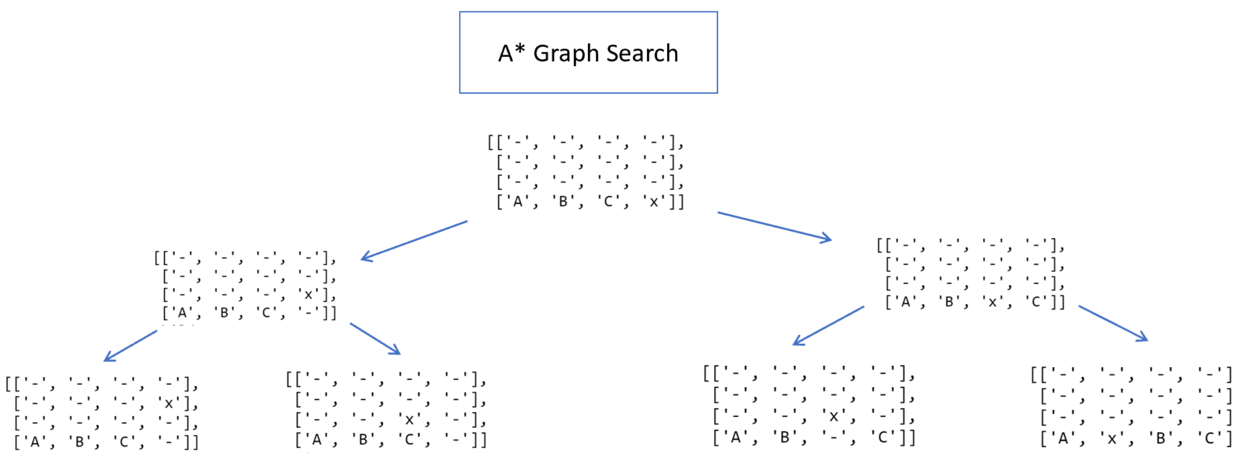


Figure 17: A Star Graph Search

```
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', 'x']]
[['-', '-', '-', '-'],
 ['-', 'A', '-', '-'],
 ['-', 'B', '-', '-'],
 ['-', 'C', '-', '-']]
Parent
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', 'x']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', 'x'],
 ['A', 'B', 'C', '-']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'x', 'C']]
Parent
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', 'x'],
 ['A', 'B', 'C', '-']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', 'x'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', '-']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', 'x', '-'],
 ['A', 'B', 'C', '-']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', 'x']]
Parent
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'x', 'C']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', 'x', '-'],
 ['A', 'B', '-', 'C']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'x', 'B', 'C']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', 'x']]
```

(a) Varying Number of Tiles Time Complexity

```
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', 'x']]
[['-', '-', '-', '-'],
 ['-', 'A', '-', '-'],
 ['-', 'B', '-', '-'],
 ['-', 'C', '-', '-']]
Parent
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', 'x']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', 'x'],
 ['A', 'B', 'C', '-']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'x', 'C']]
Parent
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', 'x'],
 ['A', 'B', 'C', '-']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', 'x'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'C', '-']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', 'x', '-'],
 ['A', 'B', 'C', '-']]
Parent
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'B', 'x', 'C']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', 'x', '-'],
 ['A', 'B', '-', 'C']]
Children
[['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['-', '-', '-', '-'],
 ['A', 'x', 'B', 'C']]
```

(b) Varying Board Size Time Complexity

Figure 18: Time Complexity Varying problem difficulty

Appendix G Code

This project has been structured in 2 main files: Foundations.py and Blocksworld_tile_puzzle.py. In Foundations.py are included some of the basic functions which are used to create and update the world the agent is moving in and check if the current world state is equal to the goal state. In Blocksworld_tile_puzzle.py are instead implemented the different search techniques. A third file (start.py), used to create the interactive user interface, is additionally included in the attached Zip Folder (this has not been added here because of it's length). In addition to this, in the zip folder are also available the Jupyter Notebooks used in order to create the graphs used in this report.

```
import numpy as np


def world(n, m):
    grid = np.chararray((n, m))
    grid = [['-' for j in i] for i in grid]
    return grid


def move_up(w, player):
    if player[0] > 0:
        if w[player[0] - 1][player[1]] != '%':
            a = w[player[0] - 1][player[1]]
            w[player[0] - 1][player[1]] = w[player[0]][player[1]]
            w[player[0]][player[1]] = a
            player[0] = player[0] - 1
            return w, player
        else:
            return 0, 1
    else:
        return 0, 1


def move_down(w, player):
    if player[0] < len(w) - 1:
        if w[player[0] + 1][player[1]] != '%':
            a = w[player[0] + 1][player[1]]
            w[player[0] + 1][player[1]] = w[player[0]][player[1]]
            w[player[0]][player[1]] = a
            player[0] = player[0] + 1
            return w, player
        else:
            return 0, 1
    else:
        return 0, 1


def move_left(w, player):
    if player[1] > 0:
        if w[player[0]][player[1] - 1] != '%':
            a = w[player[0]][player[1] - 1]
            w[player[0]][player[1] - 1] = w[player[0]][player[1]]
            w[player[0]][player[1]] = a
            player[1] = player[1] - 1
            return w, player
        else:
            return 0, 1
    else:
        return 0, 1


def move_right(w, player):
    if player[1] < len(w) - 1:
        if w[player[0]][player[1] + 1] != '%':
```

```python
            a = w[player[0]][player[1] + 1]
            w[player[0]][player[1] + 1] = w[player[0]][player[1]]
            w[player[0]][player[1]] = a
            player[1] = player[1] + 1
            return w, player
        else:
            return 0, 1
    else:
        return 0, 1


def solution_check(w, sol):
    h = [k for i, j in zip(w, sol) for k, z in zip(i, j) if k != z]
    return len(h)
```

Listing 13: Foundations.py

```python
import pprint
import random
import numpy as np
from Foundations import world, move_up, move_down, move_left, move_right,
    solution_check
from IPython.display import clear_output
from collections import deque
import time
from operator import attrgetter
import copy

start = time.time()


class Space:
    def __init__(self, m, n, obstacles=False):
        self.m = m
        self.n = n

        grid = world(self.n, self.m)

        if obstacles is False:
            grid[self.n - 1][self.m - 2] = 'C'
            grid[self.n - 1][self.m - 3] = 'B'
            grid[self.n - 1][self.m - 4] = 'A'
            grid[self.n - 1][self.m - 1] = 'x'
            self.player = [self.n - 1, self.m - 1]
        else:
            grid[self.n - 1][self.m - 2] = 'C'
            grid[self.n - 1][self.m - 3] = 'B'
            grid[self.n - 1][self.m - 4] = 'A'

            grid[self.n - 4][self.m - 1] = '%'
            grid[self.n - 2][self.m - 2] = '%'
            grid[self.n - 1][self.m - 1] = 'x'
            self.player = [self.n - 1, self.m - 1]

        self.w = grid

    def solution(self):
        grid = world(self.n, self.m)

        grid[self.n - 1][self.m - 3] = 'C'
        grid[self.n - 2][self.m - 3] = 'B'
        grid[self.n - 3][self.m - 3] = 'A'
        return grid


```

```python
class Space2:
    def __init__(self, m, n, obstacles=False):
        self.m = m
        self.n = n

        grid = world(self.n, self.m)

        if obstacles is False:
            # Final Grid
            grid[self.n - 1][self.m - 3] = 'C'
            grid[self.n - 2][self.m - 3] = 'B'
            grid[self.n - 3][self.m - 3] = 'A'
            grid[self.n - 2][self.m - 2] = 'x'
            self.player = [self.n - 2, self.m - 2]
        else:
            # Obstacles Grid
            grid[self.n - 1][self.m - 3] = 'C'
            grid[self.n - 2][self.m - 3] = 'B'
            grid[self.n - 3][self.m - 3] = 'A'
            grid[self.n - 4][self.m - 1] = '%'
            grid[self.n - 2][self.m - 2] = '%'
            grid[self.n - 3][self.m - 2] = 'x'
            self.player = [self.n - 3, self.m - 2]

        self.w = grid


class Game(Space):
    def __init__(self, m, n, obstacles=False):
        super().__init__(m, n, obstacles)

    def __repr__(self):
        game = True
        pprint.pprint(self.w)
        clear_output(wait=True)
        while game is True:
            val = input("Choose Direction: ")
            if val == 'a':
                clear_output(wait=True)
                check1, check2 = move_left(self.w, self.player)
                if check1 != 0:
                    self.w, self.player = check1, check2
                    pprint.pprint(self.w)
                else:
                    print("Invalid move, try again")
            elif val == 'd':
                clear_output(wait=True)
                check1, check2 = move_right(self.w, self.player)
                if check1 != 0:
                    self.w, self.player = check1, check2
                    pprint.pprint(self.w)
                else:
                    print("Invalid move, try again")
            elif val == 'w':
                clear_output(wait=True)
                check1, check2 = move_up(self.w, self.player)
                if check1 != 0:
                    self.w, self.player = check1, check2
                    pprint.pprint(self.w)
                else:
                    print("Invalid move, try again")
            elif val == 'z':
                clear_output(wait=True)
                check1, check2 = move_down(self.w, self.player)
```

```python
                if check1 != 0:
                    self.w, self.player = check1, check2
                    pprint.pprint(self.w)
                else:
                    print("Invalid move, try again")
            elif val == 'e':
                game = False
                return ('End of the game!')
            else:
                print('Wrong Entry')


# play = Game(4, 4)
# print(play)


class MakeNode:
    def __init__(self, state, parent, action, path_depth, estimated_cost):
        self.state = state
        self.parent = parent
        self.path_depth = path_depth
        self.action = action
        self.estimated_cost = estimated_cost


def heuristic(world, sol, depth):
    w = [item for sublist in world for item in sublist]
    s = [item for sublist in sol for item in sublist]
    manhattan = 0
    manhattan += abs(w.index('A') - s.index('A')) + abs(w.index('B') - s.index('B')) + abs(w.index('C') - s.index('C'))
    return manhattan + depth


def expand_node(problem, node, mode, sol, visited=0):
    if mode == 3:
        up = MakeNode(move_up([x[:] for x in problem.w], [x for x in problem.player]), node.parent + problem.w,
                         node.action + ', Up', node.path_depth + 1,
                         heuristic(problem.w, sol, node.path_depth))
        down = MakeNode(move_down([x[:] for x in problem.w], [x for x in problem.player]), node.parent + problem.w,
                         node.action + ', Down', node.path_depth + 1,
                         heuristic(problem.w, sol, node.path_depth))
        left = MakeNode(move_left([x[:] for x in problem.w], [x for x in problem.player]), node.parent + problem.w,
                         node.action + ', Left', node.path_depth + 1,
                         heuristic(problem.w, sol, node.path_depth))
        right = MakeNode(move_right([x[:] for x in problem.w], [x for x in problem.player]), node.parent + problem.w,
                         node.action + ', Right', node.path_depth + 1,
                         heuristic(problem.w, sol, node.path_depth))
        res = [node for node in [up, left, down, right] if node.state[0] != 0]
        if visited != 0:
            res = [i for i in res if i.state[0] not in visited]
        return res
    else:
        up = MakeNode(move_up([x[:] for x in problem.w], [x for x in problem.player]), node.parent + problem.w,
                        node.action + ', Up', node.path_depth + 1, node.estimated_cost + 1)
        down = MakeNode(move_down([x[:] for x in problem.w], [x for x in problem.player]), node.parent + problem.w,
```

```
                          node.action + ', Down', node.path_depth + 1, node.estimated_cost + 1)
        left = MakeNode(move_left([x[:] for x in problem.w], [x for x in problem.player]), node.parent + problem.w,
                          node.action + ', Left', node.path_depth + 1, node.estimated_cost + 1)
        right = MakeNode(move_right([x[:] for x in problem.w], [x for x in problem.player]), node.parent + problem.w,
                           node.action + ', Right', node.path_depth + 1, node.estimated_cost + 1)
        res = [node for node in [up, left, down, right] if node.state[0] != 0]
        if mode == 0:
            return res
        elif mode == 1:
            res = random.sample(res, len(res))
            return res
        elif mode == 4:
            res = [i for i in res if i.state[0] not in visited]
            return res


def search(problem, mode, depth=np.inf, obstacles=False):
    computational_time = 0
    gen = []
    sol = [x[:] for x in problem.solution()]
    if mode == 3:
        fringe = [MakeNode([problem.w, problem.player], [0], 'Root', 0, heuristic(problem.w, sol, 0))]
    else:
        fringe = deque([MakeNode([problem.w, problem.player], [0], 'Root', 0, 0)])
    diff = solution_check(problem.w, sol)
    if ((diff == 1) and (obstacles is False)) or ((diff == 3) and (obstacles is True)):
        return 0, computational_time, None, None, None
    elif mode == 0:
        while True:
            if len(fringe) == 0:
                return np.inf, computational_time, None, None, None
            else:
                node = fringe.popleft()
                problem.w, problem.player = node.state[0], node.state[1]
                computational_time += 1
            diff = solution_check(node.state[0], sol)
            if ((diff == 1) and (obstacles is False)) or ((diff == 3) and (obstacles is True)):
                print("Nodes Generated (Space Complexity:", sum(gen), ")")
                return node.path_depth, computational_time, node.action, node.parent + node.state[0], node.estimated_cost
            else:
                new = expand_node(problem, node, 0, sol)
                fringe.extend(new)
                gen.append(len(new))
                if (computational_time % 50000) == 0:
                    print('Time:', computational_time)
    elif mode == 1:
        while True:
            if len(fringe) == 0:
                return np.inf, computational_time, None, None, None
            else:
                node = fringe.pop()
                problem.w, problem.player = node.state[0], node.state[1]
                computational_time += 1
            diff = solution_check(node.state[0], sol)
```

```
            if ((diff == 1) and (obstacles is False)) or ((diff == 3) and (obstacles is True)):
                print("Nodes Generated (Space Complexity:", sum(gen), ")")
                return node.path_depth, computational_time, node.action, node.parent + node.state[0], node.estimated_cost
            else:
                if node.path_depth < depth:
                    if depth == np.inf:
                        new = expand_node(problem, node, 1, sol)
                        fringe.extend(new)
                        gen.append(len(new))
                    else:
                        new = expand_node(problem, node, 0, sol)
                        fringe.extend(new)
                        gen.append(len(new))
                    if (computational_time % 50000) == 0:
                        print('Time:', computational_time)
    elif mode == 2:
        iteration = 0
        r = copy.deepcopy(problem)
        for i in range(depth):
            print("Depth: ", i)
            depth, complexity, moves, history, cost = search(copy.deepcopy(r), 1, i)
            iteration += complexity
            if moves is not None:
                return depth, iteration, moves, history, cost
        return 1, computational_time, None, None, None
    elif mode == 3:
        while True:
            if len(fringe) == 0:
                return np.inf, computational_time, None, None, None
            else:
                fringe.sort(key=attrgetter('estimated_cost'))
                node = fringe.pop(0)
                problem.w, problem.player = node.state[0], node.state[1]
                computational_time += 1
            diff = solution_check(node.state[0], sol)
            if ((diff == 1) and (obstacles is False)) or ((diff == 3) and (obstacles is True)):
                print("Nodes Generated (Space Complexity:", sum(gen), ")")
                return node.path_depth, computational_time, node.action, node.parent + node.state[0], node.estimated_cost
            else:
                new = expand_node(problem, node, mode, sol)
                fringe.extend(new)
                gen.append(len(new))
                if (computational_time % 50000) == 0:
                    print('Time:', computational_time)
    else:
        print("Select an adequate mode: \n")
        print(" 0: Breadth First Search \n 1: Depth First Search or Limited Depth Search \n "
              "2: Iterative Deepening \n 3: A Star \n")
        return None, None, None, None, None


def graph_search(problem, mode, depth=np.inf, obstacles=False):
    computational_time = 0
    gen = []
    sol = [x[:] for x in problem.solution()]
    if mode == 3:
        fringe = [MakeNode([problem.w, problem.player], [0], 'Root', 0, heuristic(problem.w, sol, 0))]
```

```python
        visited = [problem.w]
    else:
        fringe = deque([MakeNode([problem.w, problem.player], [0], 'Root', 0, 0)])
        visited = [problem.w]
    diff = solution_check(problem.w, sol)
    if ((diff == 1) and (obstacles is False)) or ((diff == 3) and (obstacles is True)):
        return 0, computational_time, None, None, None
    elif mode == 0:
        while True:
            if len(fringe) == 0:
                return np.inf, computational_time, None, None, None
            else:
                node = fringe.popleft()
                problem.w, problem.player = node.state[0], node.state[1]
                computational_time += 1
            diff = solution_check(node.state[0], sol)
            if ((diff == 1) and (obstacles is False)) or ((diff == 3) and (obstacles is True)):
                print("Nodes Generated (Space Complexity:", sum(gen), ")")
                return node.path_depth, computational_time, node.action, node.parent + node.state[0], node.estimated_cost
            else:
                new = expand_node(problem, node, 4, sol, visited)
                fringe.extend(new)
                gen.append(len(new))
                visited.extend([i.state[0] for i in new])
                if (computational_time % 50000) == 0:
                    print('Time:', computational_time)
    elif mode == 1:
        while True:
            if len(fringe) == 0:
                return np.inf, computational_time, None, None, None
            else:
                node = fringe.pop()
                problem.w, problem.player = node.state[0], node.state[1]
                computational_time += 1
            diff = solution_check(node.state[0], sol)
            if ((diff == 1) and (obstacles is False)) or ((diff == 3) and (obstacles is True)):
                print("Nodes Generated (Space Complexity:", sum(gen), ")")
                return node.path_depth, computational_time, node.action, node.parent + node.state[0], node.estimated_cost
            else:
                if node.path_depth < depth:
                    new = expand_node(problem, node, 4, sol, visited)
                    fringe.extend(new)
                    gen.append(len(new))
                    visited.extend([i.state[0] for i in new])
                    if (computational_time % 50000) == 0:
                        print('Time:', computational_time)
    elif mode == 2:
        iteration = 0
        r = copy.deepcopy(problem)
        for i in range(depth):
            print("Depth: ", i)
            depth, complexity, moves, history, cost = graph_search(copy.deepcopy(r), 1, i)
            iteration += complexity
            if moves is not None:
                return i, iteration, moves, history, cost
        return 1, computational_time, None, None, None
    elif mode == 3:
        while True:
```

```
            if len(fringe) == 0:
                return np.inf, computational_time, None, None, None
            else:
                fringe.sort(key=attrgetter('estimated_cost'))
                node = fringe.pop(0)
                problem.w, problem.player = node.state[0], node.state[1]
                computational_time += 1
            diff = solution_check(node.state[0], sol)
            if ((diff == 1) and (obstacles is False)) or ((diff == 3) and (obstacles is True)):
                print("Nodes Generated (Space Complexity:", sum(gen), ")")
                return node.path_depth, computational_time, node.action, node.parent + node.state[0], node.estimated_cost
            else:
                new = expand_node(problem, node, 3, sol, visited)
                fringe.extend(new)
                gen.append(len(new))
                visited.extend([i.state[0] for i in new])
                if (computational_time % 50000) == 0:
                    print('Time:', computational_time)
    else:
        print("Select an adequate mode: \n")
        print(" 0: Breadth First Graph Search \n 1: Depth First Graph Search or Limited Depth Graph Search \n "
              "2: Iterative Deepening Graph Search \n 3: A Star Graph Search \n")
        return None, None, None, None, None


def bi_search(problem, mode, problem2):
    computational_time = 0
    fringe = deque([MakeNode([problem.w, problem.player], [0], 'Root', 0, 0)])
    goal_fringe = deque([MakeNode([problem2.w, problem2.player], [0], '', 0, 0)])
    visited = [problem.w]
    visited2 = [problem2.w]
    if mode == 0:
        while True:
                node = fringe.popleft()
                node2 = goal_fringe.popleft()
                if node.state[0] == node2.state[0]:
                           a = node2.action.split(',')
                           a.reverse()
                           node2.parent.pop(0)
                           b = [node2.parent[i:i+len(node2.state[0])] for i in range(0, len(node2.parent), len(node2.state[0]))][::-1]
                           return node.path_depth + node2.path_depth, computational_time, node.action+','.join(a), node.parent + \
                                   node.state[0] + sum(b, []), node.estimated_cost + node2.estimated_cost
                problem.w, problem.player = node.state[0], node.state[1]
                computational_time += 1
                new = expand_node(problem, node, 4, 0, visited)
                fringe.extend(new)
                visited.extend([i.state[0] for i in new])
                problem2.w, problem2.player = node2.state[0], node2.state[1]
                new2 = expand_node(problem2, node2, 4, 0, visited2)
                goal_fringe.extend(new2)
                visited2.extend([i.state[0] for i in new2])

                if (computational_time % 50000) == 0:
                    print('Time:', computational_time)


# problem = Space(4, 4, obstacles=True)
# pprint.pprint(problem.w)
```

```
# problem2 = Space2(4, 4, obstacles=True)
# pprint.pprint(problem2.w)

# depth, complexity, moves, history, cost = bi_search(problem, 0, problem2)

problem = Space(4, 4, obstacles=False)
pprint.pprint(problem.w)
pprint.pprint(problem.solution())

depth, complexity, moves, history, cost = search(problem, 2, 30, obstacles=False)

# depth, complexity, moves, history, cost = graph_search(problem, 2, 300,
    obstacles=False)

if depth == 0:
    print('Initial state is equal to goal state')
elif depth == np.inf:
    print('Searched Tree, no possible result')
elif complexity is None:
    print("Please follow the instructions to run the simulation")
else:
    print('Scored Computational TIme:', complexity)
    print('node Depth to reach goal state:', depth)
    print("Estimated Path Cost:", cost)
    print('Moves used to reach goal state (', str(moves).count(','), ') :\n',
    moves)
    print('Graphical representation of moves:')
    pprint.pprint(list(history))
end = time.time()
print("Elapsed Time = %s" % (end - start))
```
Listing 14: Blocksworld_tile_puzzle.py