# Introduction

This report mentions my approach to the implementation of four tree searching methods (depth first search, breath first search, iterative deepening search and a\* search) for the “Blocksworld Tile Puzzle”. Evidence will be provided to show that the implementation works as intended. Finally, there is analysis of the scalability of the four methods and the limitations of the implementation.

# Approach

## Depth-First Search

There are two approaches for the depth first search. In the first approach, the agent move randomly to prevent the search from stuck. The agent will move indefinitely until it reaches the goal state. Then, it will return the solution.

In the second approach, a depth limit is defined to prevent the search from going endlessly. First, a node stack will be constructed. After putting the root in the stack, it will pop the node on the top of the stack. If the node is not in the goal state and it has not reached the depth limit, it will be expanded and the expanded nodes will be put at the top of the stack. Because the stack is “last-in, first-out”, the most recently expanded nodes, which are also the deepest nodes will be popped first. If the node has reached the depth limit already, it will not be expanded. It repeats all the steps above until it finds a node in the goal state, or there is no more node in the stack. Then, it will return the solution or none if no solution is found.

## Breath-First Search

First, it constructs a node queue then put the root in the queue. After that, it will visit all the nodes in the queue. If it finds a node in the goal state, it will stop and return the solution. If there is no node in the goal state, pop all the nodes in the queue and expand them. This will continue until all nodes in the queue are popped and expanded. The expanded nodes will then be put in the queue. Since the all the nodes are popped, the expanded nodes will all have the same depth. Therefore, the search will visit all the nodes in the same depth first, then expand all the nodes and then visit the expanded nodes in the deeper depth. It repeats all the steps above until it finds a node in the goal state, or there is no more node in the queue. Then, it will return the solution or none if no solution is found.

## Iterative-Deepening Search

It is similar with the depth first search but the depth limit will increase if no solution is found after all the nodes are popped. After putting the root in the stack, it will pop the node on the top of the stack. If the node is not in the goal state and it has not reached the depth limit, it will be expanded and the expanded nodes will be put at the top of the stack. Because the stack is “last-in, first-out”, the most recently expanded nodes, which are also the deepest nodes will be popped first. If the node has reached the depth limit already, it will not be expanded. If it finds a node in the goal state, it will return the solution. If there is no more node, it will increase the depth limit by 1, put the root in the stack and restart searching.

## A\* Search

For the A\* Search, the heuristic algorithm will calculate the Manhattan distance between the position of A, B, C in the current state and the position of A, B, C in the goal state respectively. After that, it will sum up the Manhattan distance of A, B, and C and the depth of the node as the estimated cost. Like other searching algorithms, it constructs a node queue then put the root in the queue. Then, it will pop the node with the lowest estimated cost. If the popped node is not in the goal state, it will expand the popped node. The expanded node will then be put in to the node queue. It keeps popping and expanding until it finds the solution or there is no more node.

# Evidence

## Solution Check

There is a function to check whether the solution found by the search is correct. The function checks whether it can reach the goal state from the start state by applying the solution.

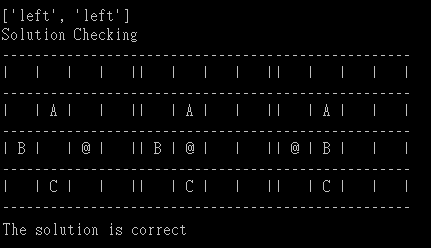


Figure 1. Solution Check Output

## Visiting Order

The visiting order is recorded during the search to make sure it is performing the search wanted. Example output is provided below.

### Randomized Depth-first Search

The search always visits the deepest node first and the moves are selected randomly.

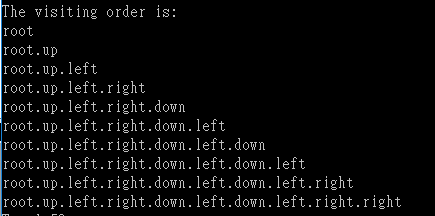


Figure 2. Visiting order of Randomized Depth-first Search

### Depth-first Search with Depth Limit

The search visits the deepest node first. After it reaches the depth limit, it will stop expanding nodes and visit other nodes in the stack.



Figure 3. Visiting order of Depth-first Search with Depth Limit

### Breadth-first search

The search visits all the nodes at the same depth first, then visits the deeper nodes.

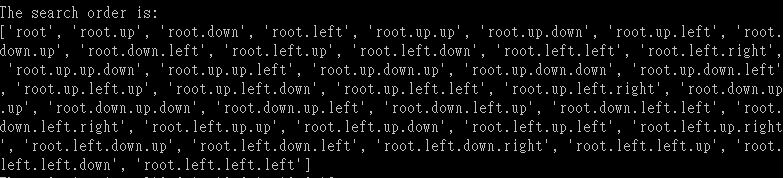


Figure 4. Visiting order of Breadth-first Search

### Iterative Deepening Search

The search visits the deepest node first. If it reaches the depth limit, it will stop expanding nodes. If there is no more node in the stack, it will revisit the root again and the depth limit is also increased by 1.

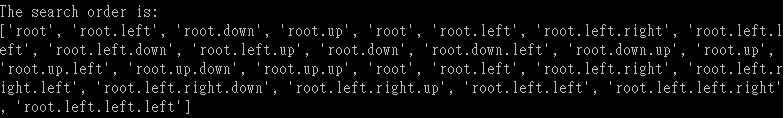


Figure 5. Visiting order of Iterative Deepening Search

### A\* Search

The search visits the node with the least estimated cost first.



Figure 6. Visiting order of A\* Search

## Example Output

See Appendix.

# Scalability study

To measure the scalability of the tree search algorithms, 5 different start states will be used for the search. Each start state has different minimal distance (from 3 to 14) to the goal state. The depth limit for the depth-first search with depth-limit is set to 20.

# Extras and limitations

# References

# Appendix

## Example Output

