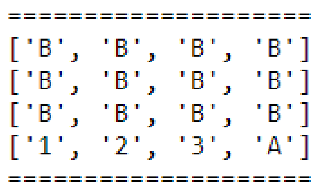
Coursework

Pornphapat Innwon 31378218

November 2019

1 Approach

For the programming language, the program to solve a puzzle problem is written in Python3.7. The puzzle size is 4*4 grid board. The board or a state is defined as the figure shown below, where:



```
====================
['B', 'B', 'B', 'B']
['B', 'B', 'B', 'B']
['B', 'B', 'B', 'B']
['1', '2', '3', 'A']
====================
```

Figure 1: The state of the board

- `A` as tile where an agent located
- `1, 2, 3` as tile where objective boxes located
- `B` as blank or empty tiles

For the implementation design, there is a main class called 'node' in the program that can generate a state of all search methods. The attributes of this class are:

- `state` as the state of the board.
- `agent_pos` as the position of the agent in the board.
- `predecessor` as the parent of the node.
- `depth` as the depth of the node.
- `total_cost` as the total cost or path cost of the node.
- `hcost` as the heuristic cost of the node.

This class also have functions which are:

- `create_successor()` to generate all possible actions on that state. The actions are always random so that it will solve loop problem in depth first search
- `move()` to swap the agent and a block tile.
- `copy_state()` to clone the state.
- `get_solution()` to show path from the initial state leading to goal state.
- `find_hcost()` to calculate heuristic value. It is only used in A* heuristic search.
- `find_fcost()` to calculate function value.

In general, each search method will generate a number of the nodes until they found a goal state. For the tree structure, there is a list named fringe that stores new nodes or successors depending on search method and always removes the first node in the list to find a solution. Furthermore, the actions of successors are always random as explained in the class function. Most importantly, all programs will be counted as fail if the node expansion is exceed 100,000 due to the massive computing time.

Each algorithm also needs to find the coordinator of the agent first before creating a root node. The reason for this is to able to change the difficulty of the puzzle. In other words, an agent location at start state will be not in the same place. Some algorithm is adjusted as described below.

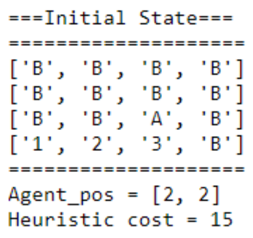
A* Heuristic Search

Concerning the heuristic value, The algorithm uses a sum of the Manhattan distance multiplied by three because it finds a solution significantly faster than only the total of the Manhattan distance.

2 Evidence

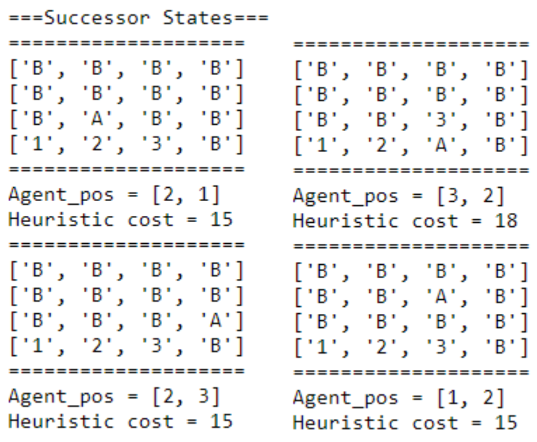
The class node

These figure below show the result of the computing successor nodes, agent position and heuristic value.



```
===Initial State===
====================
['B', 'B', 'B', 'B']
['B', 'B', 'B', 'B']
['B', 'B', 'A', 'B']
['1', '2', '3', 'B']
====================
Agent_pos = [2, 2]
Heuristic cost = 15
```

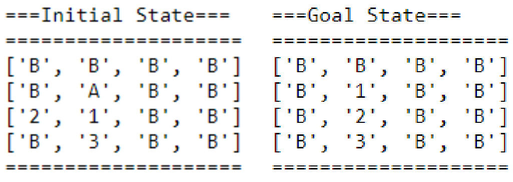
Figure 2: Initial state



```
===Successor States===
====================    ====================
['B', 'B', 'B', 'B']    ['B', 'B', 'B', 'B']
['B', 'B', 'B', 'B']    ['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']    ['B', 'B', '3', 'B']
['1', '2', '3', 'B']    ['1', '2', 'A', 'B']
====================    ====================
Agent_pos = [2, 1]      Agent_pos = [3, 2]
Heuristic cost = 15     Heuristic cost = 18
====================    ====================
['B', 'B', 'B', 'B']    ['B', 'B', 'B', 'B']
['B', 'B', 'B', 'B']    ['B', 'B', 'A', 'B']
['B', 'B', 'B', 'A']    ['B', 'B', 'B', 'B']
['1', '2', '3', 'B']    ['1', '2', '3', 'B']
====================    ====================
Agent_pos = [2, 3]      Agent_pos = [1, 2]
Heuristic cost = 15     Heuristic cost = 15
```

Figure 3: Successor states

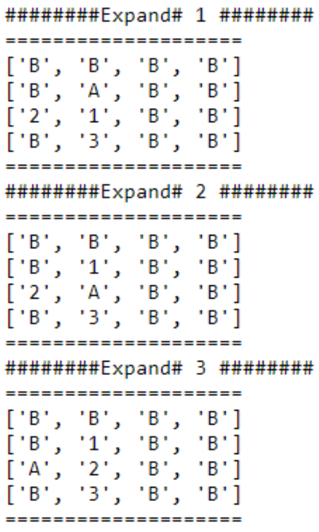
These figures below show how each methods works and the output of a solution. The depth difficulty of these methods is two. That is, it is the minimum step cost that is needed to go to the goal state.



```
===Initial State===    ===Goal State===
====================   ====================
['B', 'B', 'B', 'B']   ['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']   ['B', '1', 'B', 'B']
['2', '1', 'B', 'B']   ['B', '2', 'B', 'B']
['B', '3', 'B', 'B']   ['B', '3', 'B', 'B']
====================   ====================
```

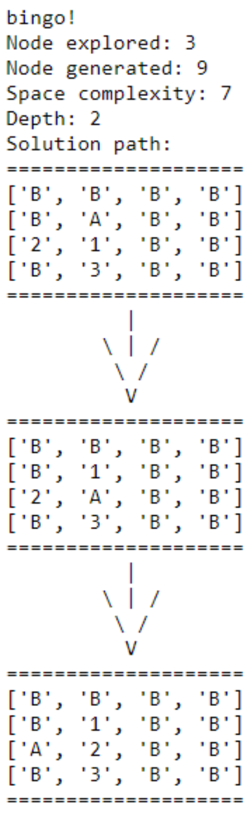
Figure 4: Initial state and goal state

Depth first search



```
########Expand# 1 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 2 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 3 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(a) Node expansion

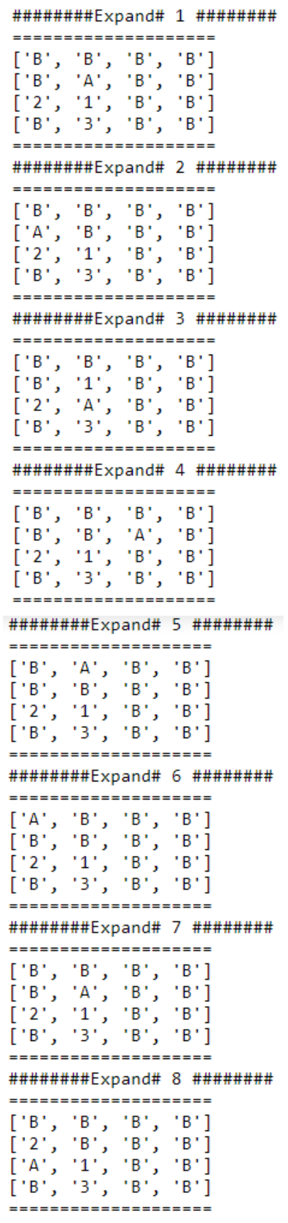


```
bingo!
Node explored: 3
Node generated: 9
Space complexity: 7
Depth: 2
Solution path:
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(b) A Solution

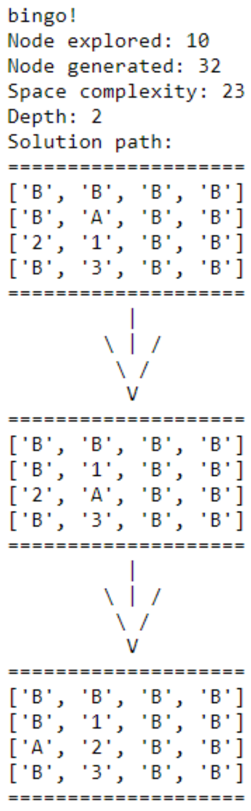
Figure 5: The expansion of the depth first search and the solution

Breadth first search



```
########Expand# 1 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 2 ########
====================
['B', 'B', 'B', 'B']
['A', 'B', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 3 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 4 ########
====================
['B', 'B', 'B', 'B']
['B', 'B', 'A', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 5 ########
====================
['B', 'A', 'B', 'B']
['B', 'B', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 6 ########
====================
['A', 'B', 'B', 'B']
['B', 'B', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 7 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 8 ########
====================
['B', 'B', 'B', 'B']
['2', 'B', 'B', 'B']
['A', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(a) Some process of the node expansion

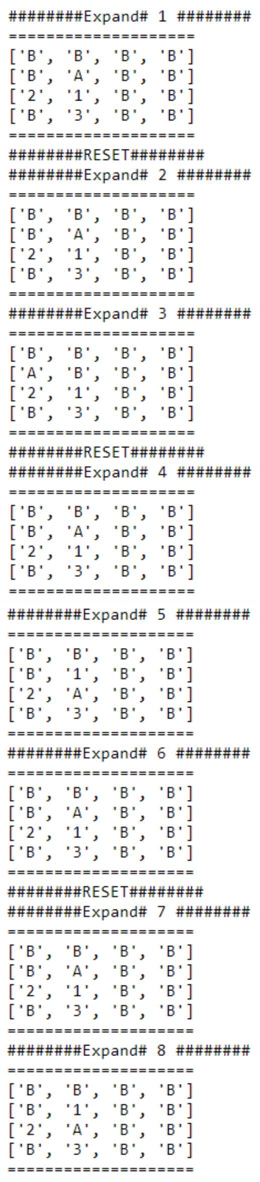


```
bingo!
Node explored: 10
Node generated: 32
Space complexity: 23
Depth: 2
Solution path:
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(b) A Solution

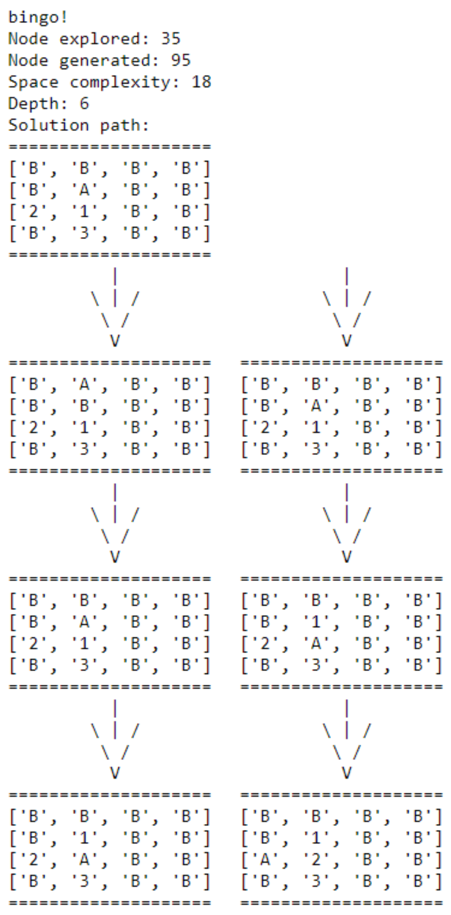
Figure 6: The expansion of the breadth first search and the solution

Iterative deepening search



```
########Expand# 1 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########RESET########
########Expand# 2 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 3 ########
====================
['B', 'B', 'B', 'B']
['A', 'B', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########RESET########
########Expand# 4 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 5 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 6 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########RESET########
########Expand# 7 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 8 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(a) Some process of the node expansion

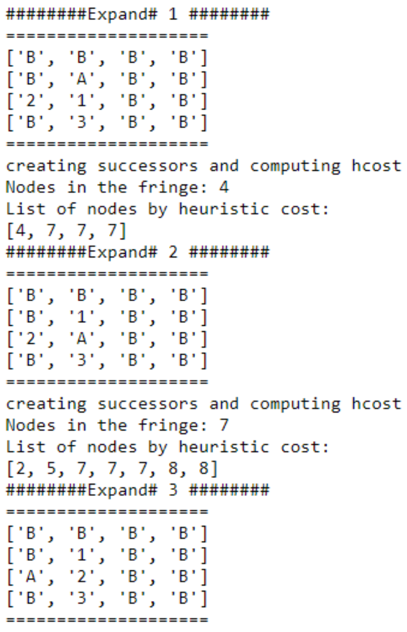


```
bingo!
Node explored: 35
Node generated: 95
Space complexity: 18
Depth: 6
Solution path:
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |                     |
        \ | /                 \ | /
         \ /                   \ /
          V                     V
====================  ====================
['B', 'A', 'B', 'B']  ['B', 'B', 'B', 'B']
['B', 'B', 'B', 'B']  ['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']  ['2', '1', 'B', 'B']
['B', '3', 'B', 'B']  ['B', '3', 'B', 'B']
====================  ====================
          |                     |
        \ | /                 \ | /
         \ /                   \ /
          V                     V
====================  ====================
['B', 'B', 'B', 'B']  ['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']  ['B', '1', 'B', 'B']
['2', '1', 'B', 'B']  ['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']  ['B', '3', 'B', 'B']
====================  ====================
          |                     |
        \ | /                 \ | /
         \ /                   \ /
          V                     V
====================  ====================
['B', 'B', 'B', 'B']  ['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']  ['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']  ['A', '2', 'B', 'B']
['B', '3', 'B', 'B']  ['B', '3', 'B', 'B']
====================  ====================
```

(b) A Solution

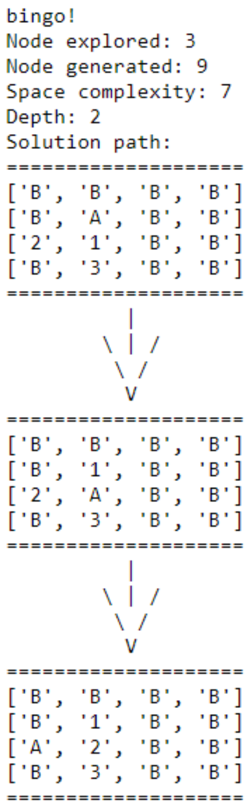
Figure 7: The expansion of the iterative deepening search and the solution

A* Heuristic Search



```
########Expand# 1 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
creating successors and computing hcost
Nodes in the fringe: 4
List of nodes by heuristic cost:
[4, 7, 7, 7]
########Expand# 2 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
creating successors and computing hcost
Nodes in the fringe: 7
List of nodes by heuristic cost:
[2, 5, 7, 7, 7, 8, 8]
########Expand# 3 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(a) Node expansion



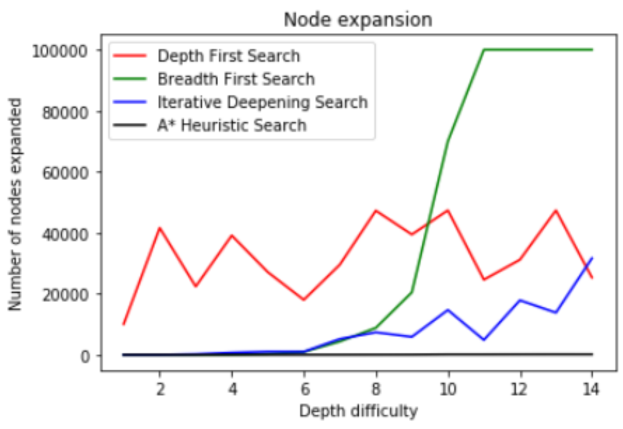
```
bingo!
Node explored: 3
Node generated: 9
Space complexity: 7
Depth: 2
Solution path:
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(b) A Solution

Figure 8: The expansion of the A* heuristic search and the solution

3 Scalability

The graphs displayed below describe the average value of node expansion and time complexity for all algorithm. The depth difficult in the graph is lowest steps used to find a solution.



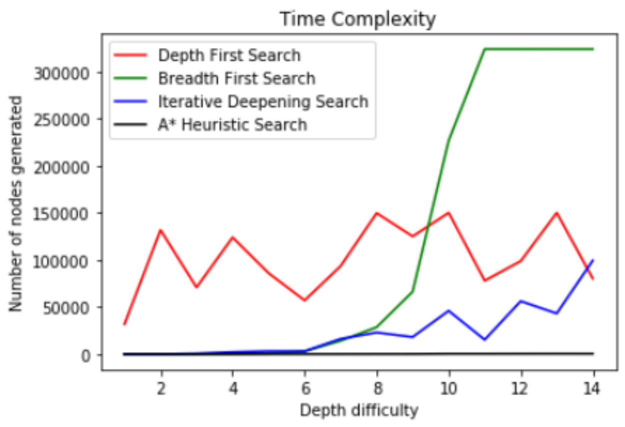


Figure 9: The node expansion and time complexity graph

From an observation of these two graphs, they demonstrate that the breadth-first search method fails once the depth difficulty reaches eleven. In other words, it has the highest average time complexity. The reason for this is because the algorithm expands all shallowest nodes while the goal node is among deeper nodes. Each time, the search goes down, the time complexity exponentially increases. Assuming that the average of successors is three, it means that the method will need 3^{11} or 177,147 expanded nodes to find a solution in depth 11 and 3^{14} or 4,782,969 expanded nodes in depth 14. Nevertheless, the method is the second-lowest time complexity when a solution is in the shallow node.

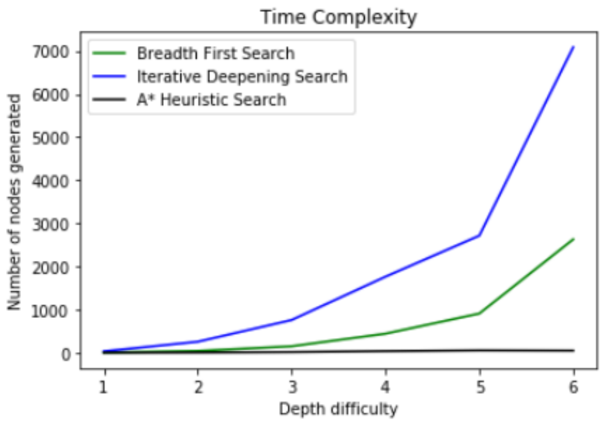


Figure 10: The time complexity during the first half

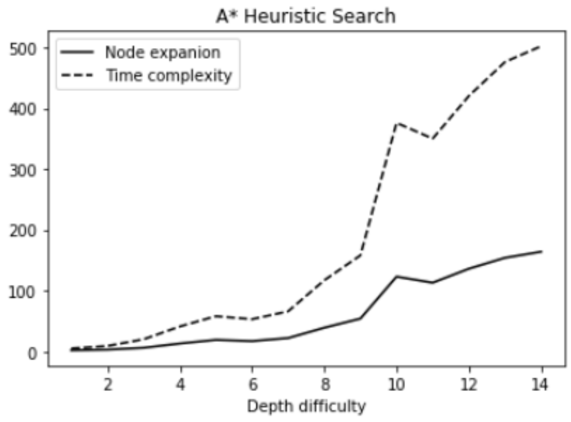


Figure 11: The A* heuristic search

On the contrary, the heuristic A* search has the lowest average of node expansion and time complexity due to heuristic leading the search to a solution. Regarding iterative deepening search and depth-first search, values in the graph are random because the actions or successors are random. Even so, iterative deepening algorithm is better on both terms of the number of nodes expanded and generated. The cause is the search limit the depth to find the shallowest solution while the depth-first search did not.

In summary, A* heuristic search is the best in term of time complexity. On the hand, for uninformed search, the tree search methods for time complexity are breadth-first search if a solution is near the root node and iterative deepening search if a goal state is very far from the initial state.

4 Extras and limitations

For the extras, the space complexity graphs for tree search is displayed below.

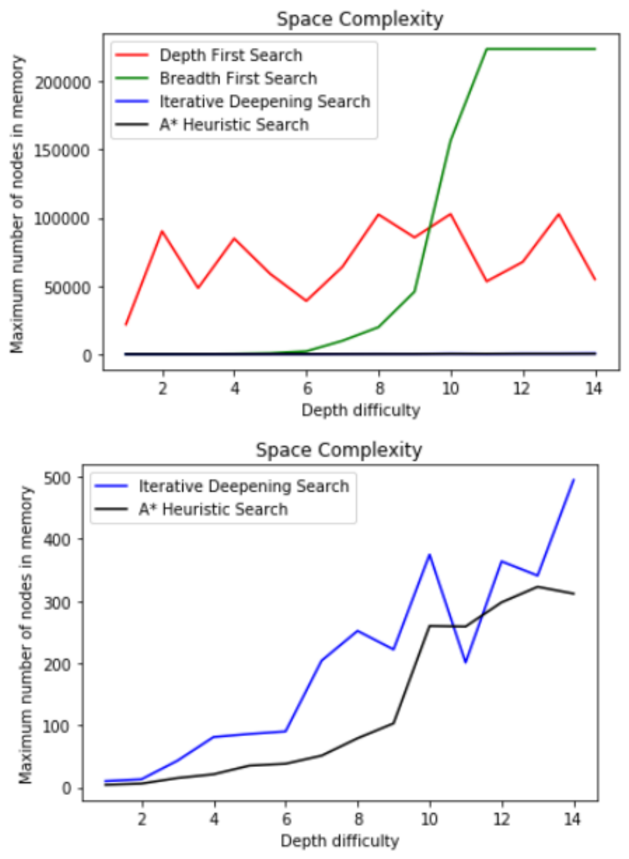
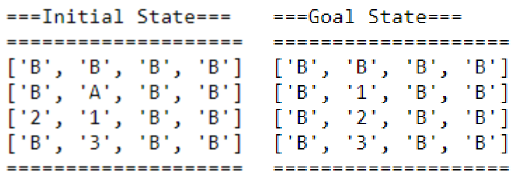


Figure 12: Space Complexity

While the trend is similar to the time complexity graph, it is easy to find that iterative deepening search has a lower value.

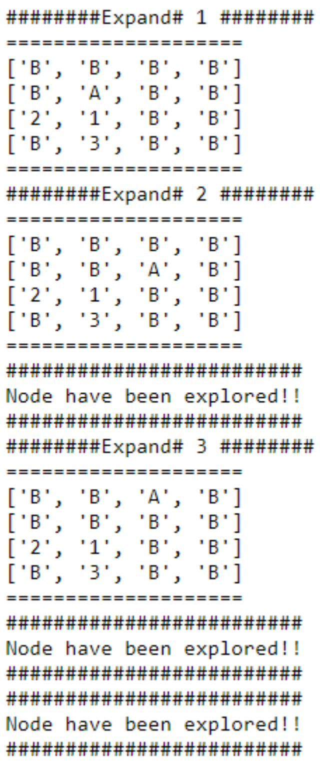
Additionally, the graph search methods are implemented. The difference can be seen as shown below.



```
===Initial State===    ===Goal State===
====================   ====================
['B', 'B', 'B', 'B']   ['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']   ['B', '1', 'B', 'B']
['2', '1', 'B', 'B']   ['B', '2', 'B', 'B']
['B', '3', 'B', 'B']   ['B', '3', 'B', 'B']
====================   ====================
```

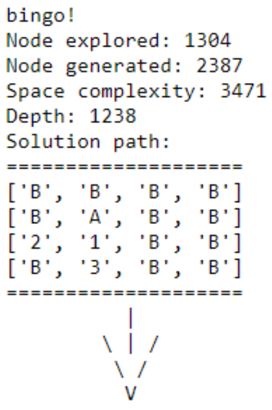
Figure 13: Initial state and goal state

Depth first search



```
########Expand# 1 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 2 ########
====================
['B', 'B', 'B', 'B']
['B', 'B', 'A', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
#########################
Node have been explored!!
#########################
########Expand# 3 ########
====================
['B', 'B', 'A', 'B']
['B', 'B', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
#########################
Node have been explored!!
#########################
#########################
Node have been explored!!
#########################
```

(a) Some process of node expansion

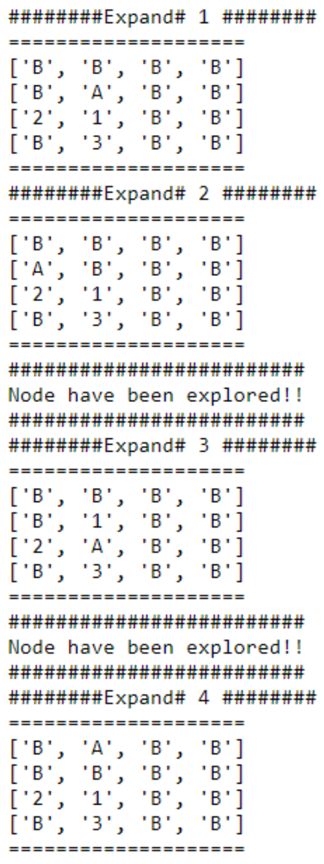


```
bingo!
Node explored: 1304
Node generated: 2387
Space complexity: 3471
Depth: 1238
Solution path:
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
```

(b) A part of the Solution

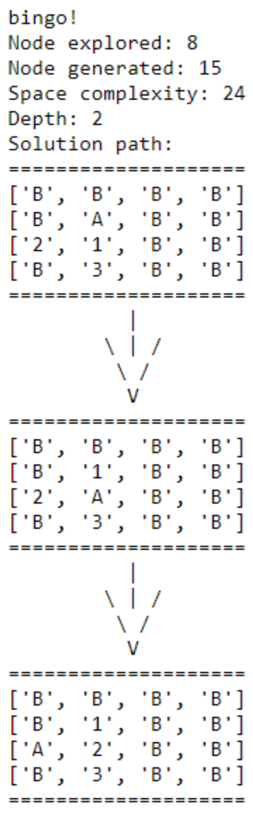
Figure 14: The expansion of the depth first search and the solution

Breadth first search



```
########Expand# 1 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 2 ########
====================
['B', 'B', 'B', 'B']
['A', 'B', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
#########################
Node have been explored!!
#########################
########Expand# 3 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
#########################
Node have been explored!!
#########################
########Expand# 4 ########
====================
['B', 'A', 'B', 'B']
['B', 'B', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(a) Some process of the node expansion

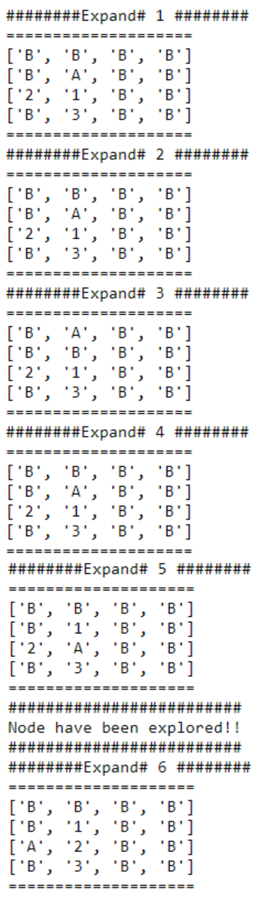


```
bingo!
Node explored: 8
Node generated: 15
Space complexity: 24
Depth: 2
Solution path:
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(b) A Solution

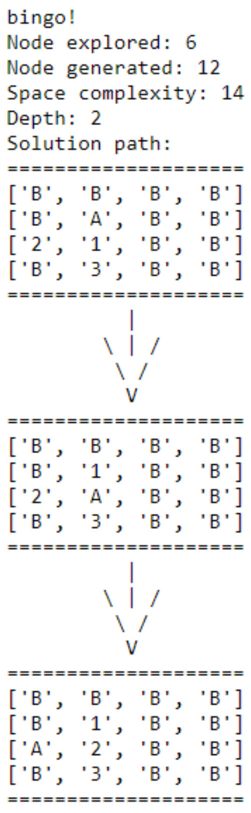
Figure 15: The expansion of the breadth first search and the solution

Iterative deepening search



```
########Expand# 1 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 2 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 3 ########
====================
['B', 'A', 'B', 'B']
['B', 'B', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 4 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 5 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
#########################
Node have been explored!!
#########################
########Expand# 6 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(a) Node expansion

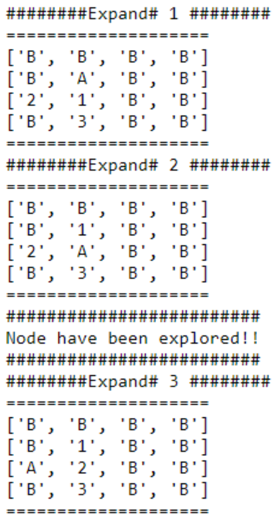


```
bingo!
Node explored: 6
Node generated: 12
Space complexity: 14
Depth: 2
Solution path:
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
         |
       \ | /
        \ /
         V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
         |
       \ | /
        \ /
         V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(b) A Solution

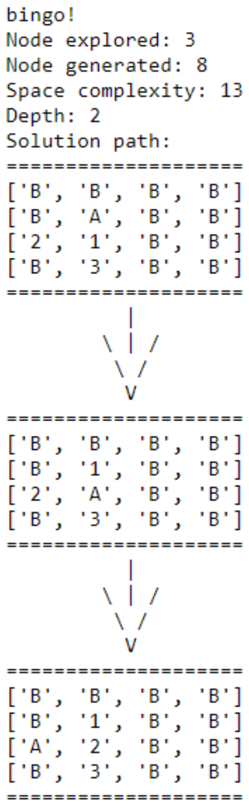
Figure 16: The expansion of the iterative deepening search and the solution

A* Heuristic Search



```
########Expand# 1 ########
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
########Expand# 2 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
#########################
Node have been explored!!
#########################
########Expand# 3 ########
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(a) Node expansion



```
bingo!
Node explored: 3
Node generated: 8
Space complexity: 13
Depth: 2
Solution path:
====================
['B', 'B', 'B', 'B']
['B', 'A', 'B', 'B']
['2', '1', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['2', 'A', 'B', 'B']
['B', '3', 'B', 'B']
====================
          |
        \ | /
         \ /
          V
====================
['B', 'B', 'B', 'B']
['B', '1', 'B', 'B']
['A', '2', 'B', 'B']
['B', '3', 'B', 'B']
====================
```

(b) A Solution

Figure 17: The expansion of the A* heuristic search and the solution

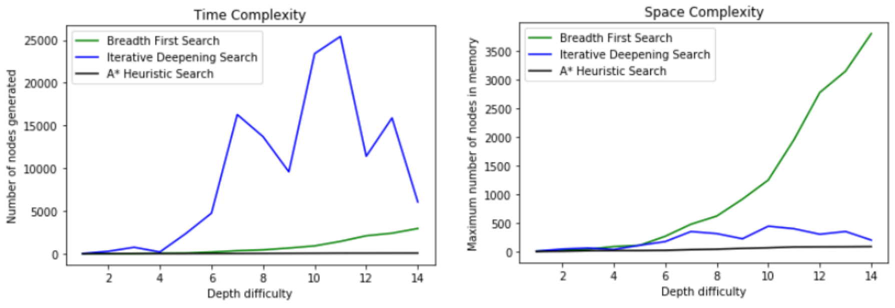


Figure 18: Time and space complexity graph

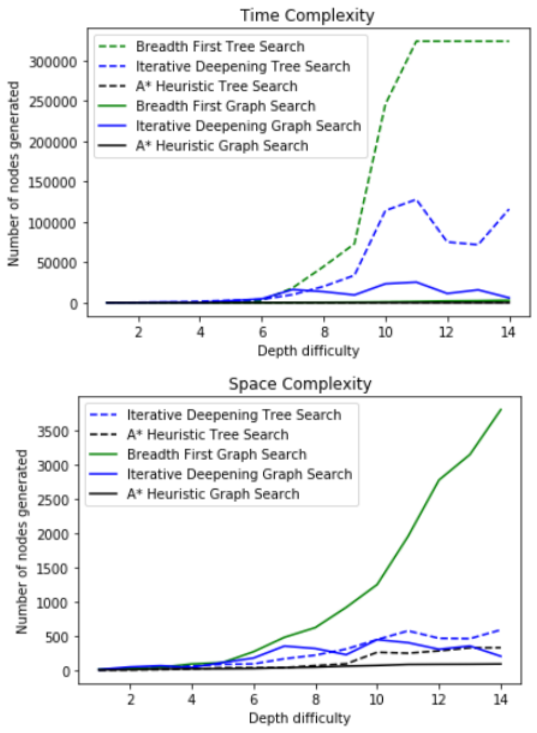


Figure 19: Comparing Time and space complexity of tree and graph search

From the graphs above, the tree search uses less memory than graph search but worse in term of time complexity. The depth-first graph search can not compute due to massive space complexity. The value can be seen in figure 14.

5 Code

The class node

```
#print methods
def print_state(state):
    print("====================")
    for i in state:
        print(i)
    print("====================")

def print_solution(solution):
    print("Solution path:")
    for state in solution:
        print_state(state)
        if(state != solution[-1]):
            print("         |          ")
            print("       \ | /        ")
            print("        \ /         ")
            print("         V          ")

class Node:
    def __init__(self,state,agent_pos,predecessor,depth,total_cost,hcost):
        self.state = state
        self.agent_pos = agent_pos
        self.predecessor = predecessor
        self.depth = depth
        self.total_cost = total_cost
        self.hcost = hcost

    def create_successor(self):
        x,y = self.agent_pos
        action_list = []
        #Can go right?
        if x+1 < len(self.state[0]) :
            action_list.append([x+1,y])
        #Can go left?
        if x-1 >= 0:
            action_list.append([x-1,y])
        #Can go up?
        if y-1 >= 0:
            action_list.append([x,y-1])
        #Can go down?
        if y+1 < len(self.state[:][0]) :
            action_list.append([x,y+1])
        #random all possible actions
        np.random.shuffle( action_list )

        successors = []
        for action in action_list :
            new_agent_pos = [action[0],action [1]]
            new_state = self.move(self. state ,x,y,action [0], action [1])
            successor_node = Node(new_state,new_agent_pos,self,self.depth
                +1,self. total_cost +1,0)
            successors.append(successor_node)
        return successors

    def move(self, old_state ,x1,y1,x2,y2):
        new_state = self.copy_state( old_state )
        new_state[x1][y1] = new_state[x2][y2]
        new_state[x2][y2] = "A"
        return new_state

    def copy_state( self , state ):
        copy_state = []
        for row in state:
            copy_row = []
            for value in row:
                copy_row.append(value)
            copy_state.append(copy_row)
        return copy_state

    def get_solution ( self ):
        solution = []
        cur_node = self
        while cur_node is not None:
            solution.append(cur_node.state)
            cur_node = cur_node.predecessor
        solution.reverse()
        return solution

    #find h
    def find_hcost( self ,goal_index):
        cur_index = {}
        cost = 0

        # find position of each required blocks ###need position of each
            blocks###
        for i in range(len(self.state [:][0]) ):
            for j in range(len(self.state [0]) ):
                if self.state[i][j] != "A" and self.state[i][j] != "B":
                    cur_index[ self.state[i][j]] = [i,j]
        for value in cur_index:
                cost = cost + abs(cur_index[value][0] - goal_index[value
                    ][0]) + abs(cur_index[value][1] - goal_index[value ][1])
        # increase the cost -> improve algorithm
        return cost*3

    def find_fcost ( self ):
        return self.hcost + self. total_cost
```

Tree search

- Depth first search

```
    def DFS(start_state,goal_state):
        fringe = []
        explored = 0
        generated = 0
        space = 0

        #find a agent
        for i in range(len(start_state [:][0]) ):
            for j in range(len(start_state[0])):
                if start_state [i][j] == "A":
                    agent_pos = [i,j]

        #create a node
        root = Node(start_state,agent_pos,None,0,0,0)
        generated = generated + 1
        fringe.append(root)
        space = 1

        while len(fringe)!=0:
            cur_node = fringe.pop(0)
            explored = explored + 1

            #check goal state
            check_state = cur_node.copy_state(cur_node.state)
            check_state[cur_node.agent_pos [0]][cur_node.agent_pos[1]]
                = "B"
            if(check_state == goal_state):
#                   print("bingo!")
#                   print("Node explored:", explored)
#                   print("Node generated:", generated)
#                   print("Space complexity:", space)
#                   print("Depth:",cur_node.depth)
                solution = cur_node.get_solution()
#                   print_solution(solution)
                return solution, explored, generated, space, cur_node.
                    depth

            #### cannot find solution ####
            if explored > 100000:
                print("More than 100000 explored! Cannot find a
                    solution")
                print("Node explored:", explored)
                print("Node generated:", generated)
            print("Space complexity:", space)
            print("Depth:",cur_node.depth)
            return None,explored, generated, space, cur_node.depth
        ################################

        #create successors
        for successor in cur_node.create_successor():
            fringe.insert(0,successor)
            generated = generated + 1

        #compute space complexity
        if(space < len(fringe)):
            space = len(fringe)
    # Can't find a solution
    print("Error! Cannot find a solution")
    return None, explored, generated, space, cur_node.depth

initial_state = [["B","B","B","B"],["B","B","B","B"],["B","B","
    B","B"],["1","2","3","A"]]
goal_state = [["B","B","B","B"],["B","1","B","B"],["B","2","B","
    B"],["B","3","B","B"]]
dfs_solution, dfs_explored, dfs_generated, dfs_space, dfs_depth =
    DFS(initial_state, goal_state)
```

- Breadth first search

```
def BFS(start_state, goal_state):
    fringe = []
    explored = 0
    generated = 0
    space = 0

    #find a agent
    for i in range(len(start_state [:][0]) ):
        for j in range(len(start_state [0]) ):
            if start_state [ i ][ j ] == "A":
                agent_pos = [i , j ]

    #create a node
    root = Node(start_state,agent_pos,None,0,0,0)
    generated = generated + 1
    fringe .append(root)
    space = 1

    while len(fringe)!=0:
        cur_node = fringe.pop(0)
        explored = explored + 1

        #check goal state
        check_state = cur_node.copy_state(cur_node.state)
        check_state [cur_node.agent_pos [0]][ cur_node.agent_pos [1]]
            = "B"
        if(check_state == goal_state):
#               print("bingo!")
#               print("Node explored:", explored)
#               print("Node generated:", generated)
#               print("Space complexity:", space)
#               print("Depth:",cur_node.depth)
            solution = cur_node.get_solution()
#               print_solution (solution)
            return solution, explored, generated, space, cur_node.
                depth

        #### cannot find solution ####
        if explored > 100000:
#               print("More than 100000 explored! Cannot find a
    solution")
#               print("Node explored:", explored)
#               print("Node generated:", generated)
#               print("Space complexity:", space)
#                 print("Depth:",cur_node.depth)
                return None,explored, generated, space, cur_node.depth
            ##############################

            #create successors
            for successor in cur_node.create_successor():
                fringe.append(successor)
                generated = generated + 1

            #compute space complexity
            if(space < len(fringe)):
                space = len(fringe)
        # Can't find a solution
        print("Error! Cannot find a solution")
        return None, explored, generated, space, cur_node.depth

 initial_state  = [["B","B","B","B"],["B","B","B","B"],["B","B","
     B","B"],["1","2","3","A"]]
goal_state = [["B","B","B","B"],["B","1","B","B"],["B","2","B","
     B"],["B","3","B","B"]]
bfs_solution, bfs_explored, bfs_generated, bfs_space, bfs_depth =
     BFS(initial_state, goal_state)
```

- Iterative deepening search

```
def IDS(start_state, goal_state):
    fringe = []
    set_depth = 0
    explored = 0
    generated = 0
    space = 0

    #find a agent
    for i in range(len(start_state [:][0]) ):
        for j in range(len(start_state[0])):
            if start_state [i][j] == "A":
                agent_pos = [i,j]

    #create a node
    root = Node(start_state,agent_pos,None,0,0,0)
    generated = generated + 1
    space = 1

    #set depth from 0 to infinity
    while set_depth > -1:
        #delete all previous data
        del fringe [:]
        fringe.append(root)

        while len(fringe)!=0:
            cur_node = fringe.pop(0)
            explored = explored + 1

            #check goal state
            check_state = cur_node.copy_state(cur_node.state)
            check_state[cur_node.agent_pos [0]][ cur_node.agent_pos
                [1]] = "B"
            if(check_state == goal_state):
#                print("bingo!")
#                print("Node explored:", explored)
#                print("Node generated:", generated)
#                print("Space complexity:", space)
#                print("Depth:",cur_node.depth)
                solution = cur_node.get_solution()
#                print_solution (solution)

                return solution, explored, generated, space,
                    cur_node.depth
            #### cannot find solution ####
            if explored > 100000:
#                   print("More than 100000 explored! Cannot find a
    solution")
#                   print("Node explored:", explored)
#                   print("Node generated:", generated)
#                   print("Space complexity:", space)
#                   print("Depth:",cur_node.depth)
                return None,explored, generated, space, cur_node.
                    depth
            ##############################

            #check if node depth equal to set_depth
            if(set_depth == cur_node.depth):
                # increase depth
                set_depth = set_depth + 1
                break
            else:
                #create successors
                for successor in cur_node.create_successor():
                    fringe.insert(0,successor)
                    generated = generated + 1

            #compute space complexity
            if(space < len(fringe)):
                space = len(fringe)
    # Can't find a solution
    print("Error! Cannot find a solution")
    return None, explored, generated, space, cur_node.depth

initial_state = [["B","B","B","B"],["B","B","B","B"],["B","B","
    B","B"],["1","2","3","A"]]
goal_state = [["B","B","B","B"],["B","1","B","B"],["B","2","B","
    B"],["B","3","B","B"]]
ids_solution, ids_explored, ids_generated, ids_space, ids_depth =
    IDS(initial_state, goal_state)
```

- A* heuristic search

```
def AHS(start_state,goal_state):
    fringe = []
    explored = 0
    generated = 0
    space = 0
    ####
    #cost = []
    ####

    #find a agent
    for i in range(len(start_state [:][0]) ):
        for j in range(len(start_state[0])):
            if start_state[i][j] == "A":
                agent_pos = [i,j]

    #get goal blocks index (for finding hcost)
    goal_index = {}
    for i in range(len(goal_state [:][0]) ):
            for j in range(len(goal_state[0])):
                if goal_state[i][j] != "A" and goal_state[i][j] !=
                    "B":
                    goal_index[goal_state[i][j]] = [i,j]

    #create a node
    root = Node(start_state,agent_pos,None,0,0,0)
    generated = generated + 1
    fringe.append(root)
    space = 1

    while len(fringe)!=0:
        cur_node = fringe.pop(0)
        #if(len(cost)!=0):
        #    cost.pop(0)
        explored = explored + 1

        #check goal state
        check_state = cur_node.copy_state(cur_node.state)
        check_state[cur_node.agent_pos[0]][cur_node.agent_pos[1]]
            = "B"
        if(check_state == goal_state):
            print("bingo!")
            print("Node explored:", explored)
            print("Node generated:", generated)
            print("Space complexity:", space)
        print("Depth:",cur_node.depth)
        solution = cur_node.get_solution()
        print_solution (solution)

        return solution, explored, generated, space, cur_node.
            depth

    #### cannot find solution ####
    if explored > 500000:
        print("More than 500000 explored! Cannot find a
            solution")
        print("Node explored:", explored)
        print("Node generated:", generated)
        print("Space complexity:", space)
        print("Depth:",cur_node.depth)
        return None,explored, generated, space, cur_node.depth
    ##############################

    #create successors
    for successor in cur_node.create_successor():
        #calculate cost so far to reach a goal
        successor.hcost = successor.find_hcost(goal_index)
        successor_fcost = successor.find_fcost()
        #sort queue on fringe based on total cost
        if(len(fringe) != 0):
            #if last node in fringe cost less than successor
                cost
            if( successor_fcost >= fringe[-1].find_fcost()):
                #cost.append(successor_fcost)
                fringe.append(successor)
            #if first node in fringe cost less than successor
                cost
            elif( successor_fcost < fringe[0]. find_fcost()):
                #cost.insert(0, successor_fcost)
                fringe.insert(0,successor)
            #if successor cost is near in the end of the fringe
            elif(( successor_fcost - fringe[0]. find_fcost()) >
                (fringe[-1]. find_fcost() - successor_fcost)):
                for i in range(len(fringe)-1, 0, -1):
                    node = fringe[i]
                    if node.find_fcost() <= successor_fcost:
                        #cost.insert(i+1,successor_fcost)
                        fringe.insert(i+1,successor)
                        break
            #if successor cost is near in the beginning of the
                fringe
                else:
                    for i in range(len(fringe)):
                        node = fringe[i]
                        if node.find_fcost () > successor_fcost:
                            #cost.insert(i, successor_fcost )
                            fringe . insert (i ,successor)
                            break
            #if there is none in fringe
            else:
                #cost.append(successor_fcost)
                fringe .append(successor)
            generated = generated + 1
            #print(cost)

        #compute space complexity
        if(space < len(fringe)):
            space = len(fringe)
    # Can't find a solution
    print("Error! Cannot find a solution")
    return None, explored, generated, space, cur_node.depth

initial_state = [["B","B","B","B"],["B","B","B","B"],["B","B","
    B","B"],["1","2","3","A"]]
goal_state = [["B","B","B","B"],["B","1","B","B"],["B","2","B","
    B"],["B","3","B","B"]]
ahs_solution, ahs_explored, ahs_generated, ahs_space, ahs_depth =
    AHS(initial_state,goal_state)
```

Graph search

- Depth first search

```
def GDFS(start_state,goal_state):
    fringe = []
    mem_state = []
    explored = 0
    generated = 0
    space = 0

    #find a agent
    for i in range(len(start_state [:][0]) ):
        for j in range(len(start_state[0])):
            if start_state [i][j] == "A":
                agent_pos = [i,j]

    #create a node
    root = Node(start_state,agent_pos,None,0,0,0)
    generated = generated + 1
    fringe.append(root)
    mem_state.append(root.state)
    space = 2
    while len(fringe)!=0:
        cur_node = fringe.pop(0)
        explored = explored + 1

        #check goal state
        check_state = cur_node.copy_state(cur_node.state)
        check_state[cur_node.agent_pos [0]][ cur_node.agent_pos[1]]
            = "B"
        if(check_state == goal_state):
            print("bingo!")
            print("Node explored:", explored)
            print("Node generated:", generated)
            print("Space complexity:", space)
            print("Depth:",cur_node.depth)
            solution = cur_node.get_solution()
            print_solution (solution)

            return solution, explored, generated, space, cur_node.
                depth

        #create successors
        for successor in cur_node.create_successor():
            if successor.state not in mem_state:
                mem_state.append(successor.state)
```

```
                    fringe . insert (0, successor)
                    generated = generated + 1

            #compute space complexity
            if(space < len(fringe)+len(mem_state)):
                space = len(fringe)+len(mem_state)
        # Can't find a solution
        print("Error! Cannot find a solution")
        return None, explored, generated, space, cur_node.depth

 initial_state  = [["B","B","B","B"],["B","B","B","B"],["B","B","
     B","B"],["1","2","3","A"]]
goal_state = [["B","B","B","B"],["B","1","B","B"],["B","2","B","
     B"],["B","3","B","B"]]
gdfs_solution , gdfs_explored, gdfs_generated, gdfs_space,
     gdfs_depth = GDFS(initial_state,goal_state)
```

- Breadth first search

```
def GBFS(start_state,goal_state):
    fringe = []
    mem_state = []
    explored = 0
    generated = 0
    space = 0

    #find a agent
    for i in range(len(start_state [:][0]) ):
        for j in range(len(start_state[0])):
            if start_state [ i ][ j ] == "A":
                agent_pos = [i , j ]

    #create a node
    root = Node(start_state,agent_pos,None,0,0,0)
    generated = generated + 1
    fringe .append(root)
    mem_state.append(root.state)
    space = 2
    while len(fringe)!=0:
        cur_node = fringe.pop(0)
        explored = explored + 1

        #check goal state
        check_state = cur_node.copy_state(cur_node.state)
        check_state[cur_node.agent_pos [0]][ cur_node.agent_pos[1]]
            = "B"
        if(check_state == goal_state):
            print("bingo!")
            print("Node explored:", explored)
            print("Node generated:", generated)
            print("Space complexity:", space)
            print("Depth:",cur_node.depth)
            solution = cur_node.get_solution()
            print_solution (solution)

            return solution, explored, generated, space, cur_node.
                depth

        #create successors
        for successor in cur_node. create_successor ():
            if successor . state not in mem_state:
                mem_state.append(successor.state)
                fringe .append(successor)
                generated = generated + 1

        #compute space complexity
        if(space < len(fringe)+len(mem_state)):
            space = len(fringe)+len(mem_state)
    # Can't find a solution
    print("Error! Cannot find a solution")
    return None, explored, generated, space, cur_node.depth

initial_state  = [["B","B","B","B"],["B","B","B","B"],["B","B","
    B","B"],["1","2","3","A"]]
goal_state  = [["B","B","B","B"],["B","1","B","B"],["B","2","B","
    B"],["B","3","B","B"]]
gbfs_solution , gbfs_explored, gbfs_generated, gbfs_space,
    gbfs_depth = GBFS(initial_state,goal_state)
```

- Iterative deepening search

```
def GIDS(start_state,goal_state):
    fringe = []
    set_depth = 0
    mem_state = []
    explored = 0
    generated = 0
    space = 0

    #find a agent
    for i in range(len(start_state [:][0]) ):
        for j in range(len(start_state[0])):
            if start_state[i][j] == "A":
                agent_pos = [i,j]

    #create a node
    root = Node(start_state,agent_pos,None,0,0,0)
    generated = generated + 1
    space = 2

    #set depth from 0 to infinity
    while set_depth > -1:
        #delete all previous data
        del fringe[:]
        del mem_state[:]
        fringe.append(root)

        mem_state.append(root.state)
        while len(fringe)!=0:
            cur_node = fringe.pop(0)
            explored = explored + 1

            #check goal state
            check_state = cur_node.copy_state(cur_node.state)
            check_state[cur_node.agent_pos [0]][cur_node.agent_pos
                [1]] = "B"
            if(check_state == goal_state):
                print("bingo!")
                print("Node explored:", explored)
                print("Node generated:", generated)
                print("Space complexity:", space)
                print("Depth:",cur_node.depth)
                solution = cur_node.get_solution()
                print_solution(solution)
                return solution, explored, generated, space,
                    cur_node.depth

            #check if node depth equal to  set_depth
            if(set_depth == cur_node.depth):
                # increase depth
                set_depth = set_depth + 1
                break
            else:
                #create successors
                for successor in cur_node.create_successor():
                    if successor.state not in mem_state:
                        mem_state.append(successor.state)
                        fringe.insert(0,successor)
                        generated = generated + 1

            #compute space complexity
            if(space < len(fringe)+len(mem_state)):
                space = len(fringe)+len(mem_state)
    # Can't find a solution
    print("Error! Cannot find a solution")
    return None, explored, generated, space, cur_node.depth

initial_state  = [["B","B","B","B"],["B","B","B","B"],["B","B","
    B","B"],["1","2","3","A"]]
goal_state = [["B","B","B","B"],["B","1","B","B"],["B","2","B","
    B"],["B","3","B","B"]]
gids_solution, gids_explored, gids_generated, gids_space,
    gids_depth = GIDS(initial_state, goal_state)
```

- A* heuristic search

```
def GAHS(start_state,goal_state):
    fringe  = []
    mem_state = []
    explored = 0
    generated = 0
    space = 0

    #find a agent
    for i  in range(len(start_state [:][0]) ):
        for j  in range(len(start_state[0]) ):
            if  start_state [ i ][ j ]  == "A":
                agent_pos = [i , j ]

    #get goal blocks index (for finding hcost)
    goal_index = {}
    for i  in range(len(goal_state [:][0]) ):
            for j  in range(len(goal_state[0])) ):
                if  goal_state [ i ][ j ]  != "A" and goal_state[i][ j ]  !=
                    "B":
                    goal_index[ goal_state [ i ][ j ]]  = [i , j ]

    #create a node
    root  = Node(start_state,agent_pos,None,0,0,0)
    generated = generated + 1
    fringe .append(root)
    mem_state.append(root.state)
    space = 2
    while len(fringe)!=0:
        cur_node = fringe.pop(0)
        explored = explored + 1

        #check goal state
        check_state  = cur_node.copy_state(cur_node.state)
        check_state [cur_node.agent_pos [0]][ cur_node.agent_pos [1]]
            = "B"
        if( check_state  == goal_state):
            print("bingo!")
            print("Node explored:", explored)
            print("Node generated:", generated)
            print("Space complexity:", space)
            print("Depth:",cur_node.depth)
            solution  = cur_node.get_solution()
            print_solution ( solution )
    return solution, explored, generated, space, cur_node.
        depth

#create successors
for successor in cur_node.create_successor():
    if successor.state not in mem_state:
        mem_state.append(successor.state)
        #calculate cost so far to reach a goal
        successor.hcost = successor.find_hcost(goal_index)
        successor_fcost = successor.find_fcost()
        #sort queue on fringe based on total cost
        if(len(fringe) != 0):
            #if last node in fringe cost less than
                successor cost
            if(successor_fcost >= fringe[-1].find_fcost()):
                fringe.append(successor)
            #if first node in fringe cost less than
                successor cost
            elif(successor_fcost < fringe[0].find_fcost()):
                fringe.insert(0,successor)
            #if successor cost is near in the end of the
                fringe
            elif((successor_fcost - fringe[0].find_fcost())
                 > (fringe[-1].find_fcost() -
                successor_fcost)):
                for i in range(len(fringe)-1, 0, -1):
                    node = fringe[i]
                    if node.find_fcost() <=
                        successor_fcost:
                        fringe.insert(i+1,successor)
                        break
            #if successor cost is near in the beginning of
                the fringe
            else:
                for i in range(len(fringe)):
                    node = fringe[i]
                    if node.find_fcost() > successor_fcost:
                        fringe.insert(i,successor)
                        break
        #if there is none in fringe
        else:
            fringe.append(successor)
        generated = generated + 1

#compute space complexity
if(space < len(fringe)+len(mem_state)):
```

```
            space = len(fringe)+len(mem_state)

    # Can't find a solution
    print("Error! Cannot find a solution")
    return None, explored, generated, space, cur_node.depth

initial_state  = [["B","B","B","B"],["B","B","B","B"],["B","B","
    B","B"],["1","2","3","A"]]
goal_state = [["B","B","B","B"],["B","1","B","B"],["B","2","B","
    B"],["B","3","B","B"]]
gahs_solution, gahs_explored, gahs_generated, gahs_space,
    gahs_depth = GAHS(initial_state,goal_state)
```