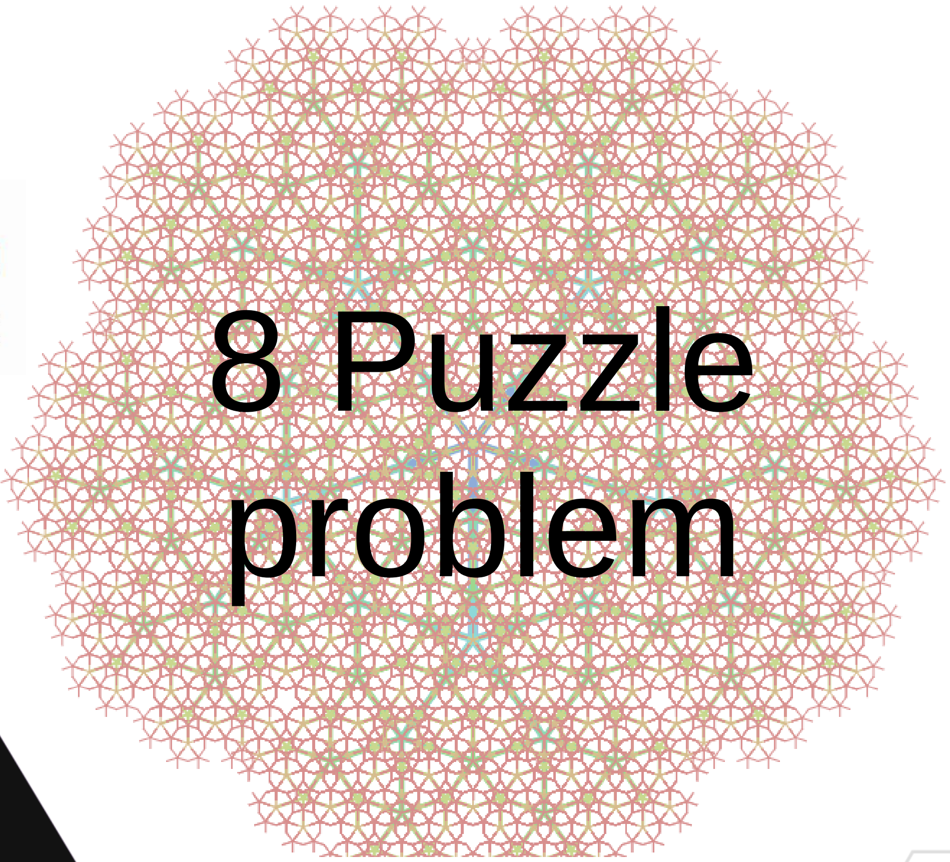


# 8 Puzzle problem

Dedicated to

Prof. Passent Mohamed Mohamed Elkafrawy

Eng. Tasneem Wael Mahmoud

Submitted by

Mohamed Atta

Submitted as a partial requirement for AIS201 assignment 1

## Abstract

In this report, an application of Artificial Intelligence is used to solve a famous problem known as the N-Puzzle. Non-informed search algorithms such as Breadth-First search and Depth-First search are proposed and implemented to solve the 8-Puzzle version. The simulations were carried out on Pycharm IDE and Google Collaboratory. The paper provides a detailed step-by-step discussion and analysis of the approach and the algorithms. Ultimately, it was found that adding costs and depth-limitations would have enhanced the simulation's performance. Also, informed search algorithms are a better approach for such problem.

## Table of Contents

Abstract ........ 2

Introduction ........ 4

Methodology ........ 5

1. Approach ........ 5

1.1. Pseudocode ........ 6

2. Functions description ........ 7

2.1. Puzzle constructor: Randomized generation ........ 7

2.2. Actions function ........ 8

2.3. Transition model ........ 9

2.4. Goal test ........ 9

2.5. Solution ........ 10

Results ........ 13

1. Depth-First search ........ 13

2. Breadth-First search ........ 14

Discussion and Conclusion ........ 15

References ........ 16

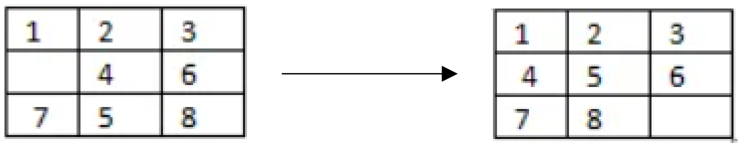
Appendices ........ 17

Appendix A ........ 17

## Introduction

The Eight Puzzle problem is originally known as the N puzzle problem or the sliding puzzle problem. This puzzle is a square that is divided into a grid of N+1 tiles (N tiles with an empty tile). The n puzzle is a well-known problem in the modelling of heuristic algorithms such as the A* algorithm to reach optimal solutions [1]. However, this report will be solving the puzzle with non-informed algorithms to satisfy the assignment's requirements.

This report discusses the case where N = 8 where the square root of (8+1) equals three rows and three columns, so this puzzle can be expressed as a 3x3 matrix. The puzzle is played by moving tiles in the same row or column of the open position horizontally or vertically by sliding them horizontally or vertically. The puzzle's purpose is to arrange the tiles in numerical sequence (see Figure 1). The report will present solving the Eight Puzzle using the non-informed Breadth-First and Depth-First search algorithms. In addition, both searching algorithms will be analyzed and compared.



| | | |
|---|---|---|
| 1 | 2 | 3 |
| | 4 | 6 |
| 7 | 5 | 8 |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

*Figure 1: Initial state and goal state*

The following section will cover the methodology by which the simulation was established, giving a detailed demonstration of the libraries used and the program design.

## Methodology

The idea of the algorithm used to approach the problem was learned and inspired from Harvard CS50's Introduction to Artificial Intelligence course on the website www.edx.org. This method was illustrated and implemented on a maze. Having learned the algorithm, implementing it on a different problem was challenging but possible. Improvements and completely different functions were invented to suit the Eight Puzzle problem. The tools used for this project were Pycharm and Google Collaboratory.

### 1. Approach

To solve the search problem, the following was required:

- An initial state
- A function to determine the valid actions in each state:

  ACTIONS(s) returns the set of actions that can be executed at the state s (aka: state space).

- A transition model to describe what state results from performing any applicable action in any state:

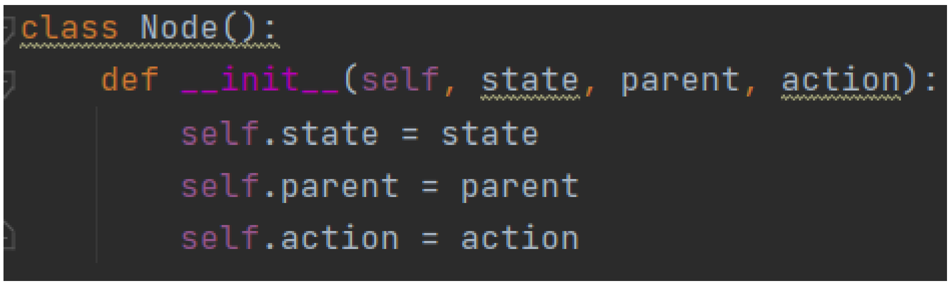
  RESULT(s, a) returns the state resulting from performing action a in state s.

- A goal test to determine whether a given state is a goal state:

  GOAL(s) returns true if the state is the goal state, otherwise false.

After the solution was to be found, we would need to obtain it without including the useless states that were explored in the process of finding the solution. Thus, we implemented a data structure called Node (see Figure 2) that keeps track of:

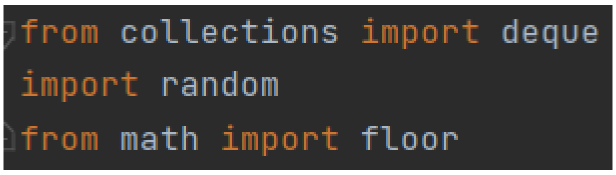
- A state
- A parent (the node that generated this node)
- An action (the action applied to the parent to get this node)



```python
class Node():
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
```

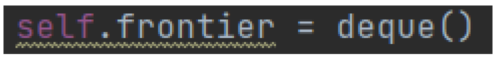
*Figure 2: Node data structure implementation*

Nodes keep the necessary information for the search, but they cannot implement the search by themselves. So, we needed a data structure called “frontier” which is either a stack or a queue depending on the type of search we are implementing. Stack implementation is for Depth-First, while the queue one is for the Breadth-First. For this structure, we imported the dequeue class from collections library (see Figures 3, 4).



```python
from collections import deque
import random
from math import floor
```

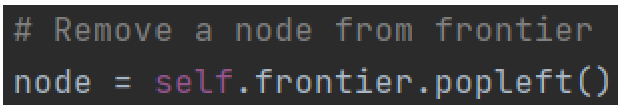
*Figure 3: The Imported libraries*



```python
self.frontier = deque()
```

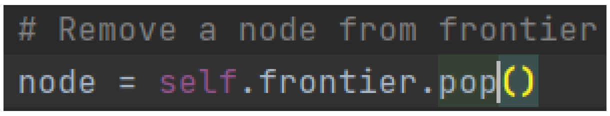
*Figure 4: Frontier initialization*

Note that BFS and DFS have the same exact search algorithm implementation. Changing the node removal method is the only difference between the two search types (see Figures 5 and 6).



```python
# Remove a node from frontier
node = self.frontier.popleft()
```

*Figure 5: Queue*



```python
# Remove a node from frontier
node = self.frontier.pop()
```

*Figure 6: Stack*

## 1.1. Pseudocode

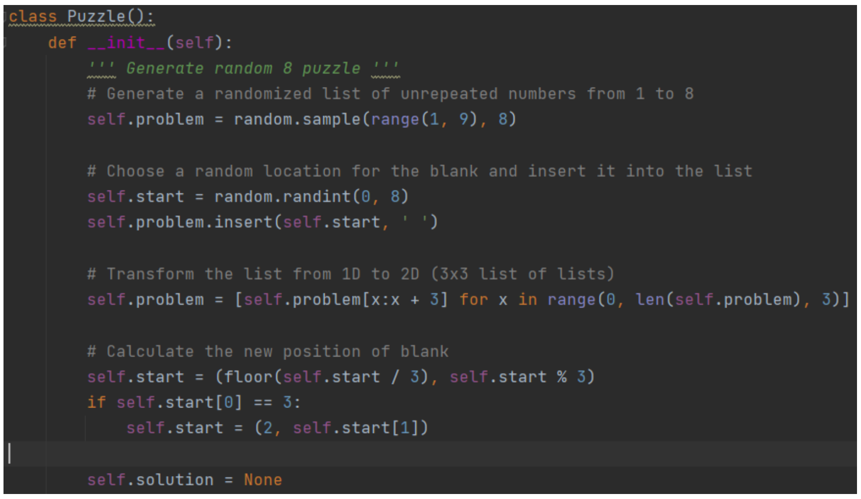
The puzzle implemented as an entire class that contains a constructor along with other essential functions (neighbours, goal_test, transition_model, DFS, BFS). The node structure is implemented as another class (see Appendix A for full code). Here is the pseudocode of the search algorithms:

- Start with a frontier containing the initial state
- Start with an empty explored set
- Repeat:
    - If the frontier is empty
        - Return no solution
    - Remove a node from frontier
    - If the node contains a goal state
        - Return solution
    - Add the node state to explored set
    - Expand the node using the valid actions and transition model functions, add resulting nodes to the frontier if they are not already in the frontier OR the explored set

## 2. Functions description

### 2.1. Puzzle constructor: Randomized generation

A constructor was made to generate random Eight Puzzles in each run (see Figure 7). This implementation used the random library to generate random numbers and lists. Firstly, a random one-dimensional list of unrepeated numbers from 1 to 8 is generated using the function random.sample(). Secondly, a random location is chosen for the empty tile using the function random.randint() to be inserted in the list using the insert function. Then, the list is transformed from 1D to 2D to become a list containing 3 lists using list. The result is the required 3x3 matrix problem. Then the new 2D coordinates of the empty position (the actor) are calculated. The problem’s solution is initialized to none.

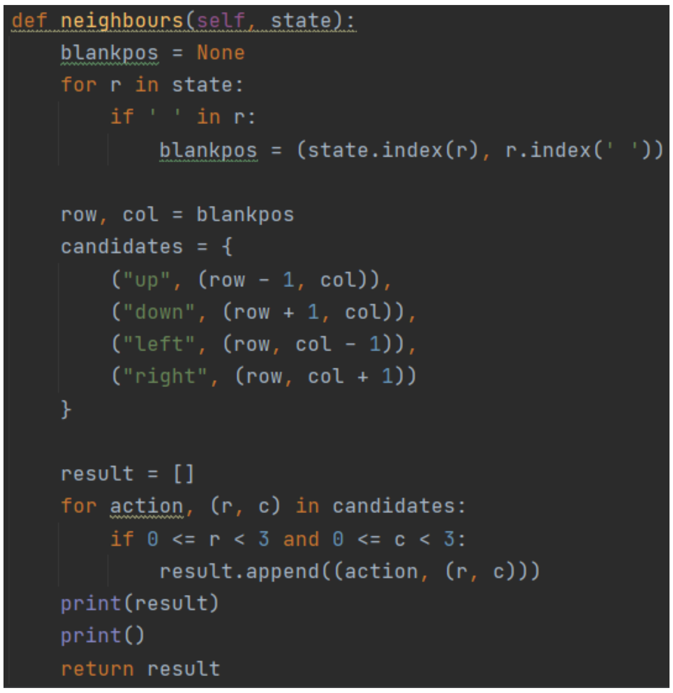


```python
class Puzzle():
    def __init__(self):
        ''' Generate random 8 puzzle '''
        # Generate a randomized list of unrepeated numbers from 1 to 8
        self.problem = random.sample(range(1, 9), 8)

        # Choose a random location for the blank and insert it into the list
        self.start = random.randint(0, 8)
        self.problem.insert(self.start, ' ')

        # Transform the list from 1D to 2D (3x3 list of lists)
        self.problem = [self.problem[x:x + 3] for x in range(0, len(self.problem), 3)]

        # Calculate the new position of blank
        self.start = (floor(self.start / 3), self.start % 3)
        if self.start[0] == 3:
            self.start = (2, self.start[1])

        self.solution = None
```

*Figure 7: Puzzle constructor*

## 2.2. Actions function

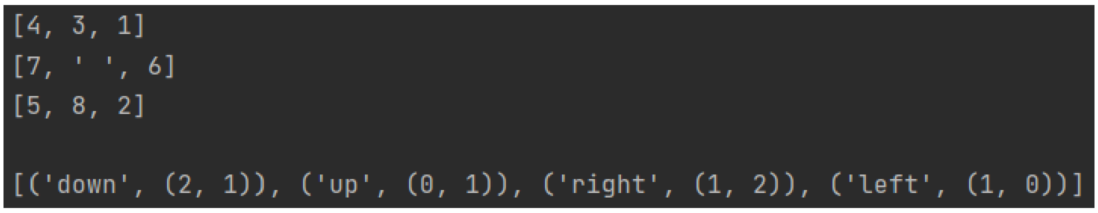
The neighbors(s) function takes the current state s and returns a list of the actions that can be taken in that state (see Figure 8).



```python
def neighbours(self, state):
    blankpos = None
    for r in state:
        if ' ' in r:
            blankpos = (state.index(r), r.index(' '))

    row, col = blankpos
    candidates = {
        ("up", (row - 1, col)),
        ("down", (row + 1, col)),
        ("left", (row, col - 1)),
        ("right", (row, col + 1))
    }

    result = []
    for action, (r, c) in candidates:
        if 0 <= r < 3 and 0 <= c < 3:
            result.append((action, (r, c)))
    print(result)
    print()
    return result
```

*Figure 8: neighbors function*

First, it locates the empty tile position then stores it as a tuple in the variable blankpos. Next, it looks at all the possible moves in the game generally which are (up-down-left-right). Finally, it determines which of these moves are valid for the current position and returns the result as an array of tuples. Each tuple consists of the move and the future index of the blank after implementing this move (see Figure 9).

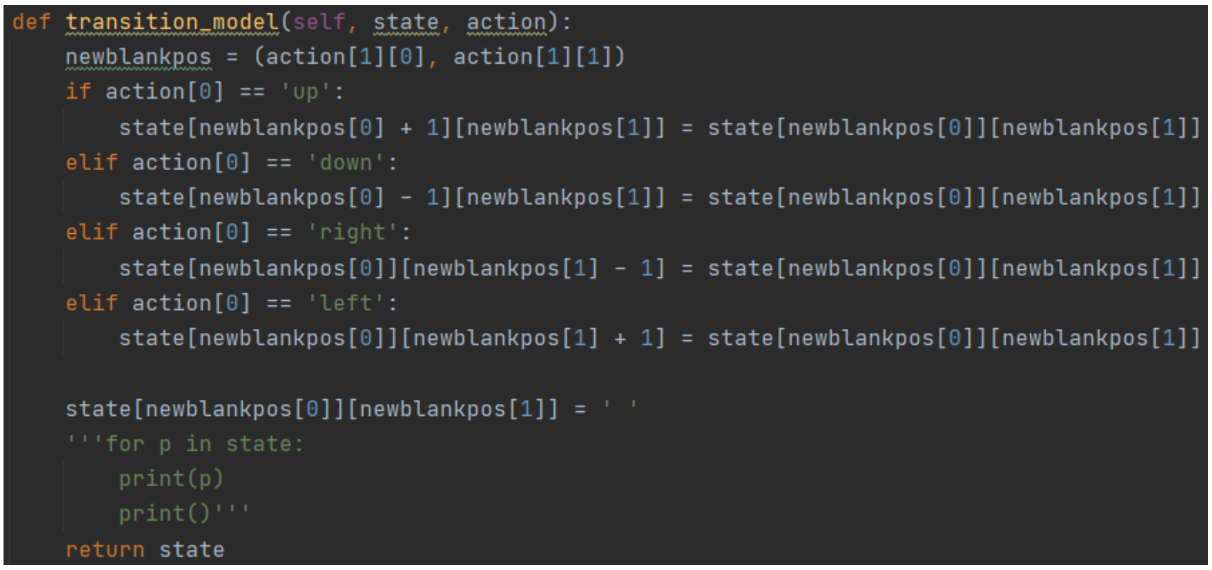


```
[4, 3, 1]
[7, ' ', 6]
[5, 8, 2]

[('down', (2, 1)), ('up', (0, 1)), ('right', (1, 2)), ('left', (1, 0))]
```

*Figure 9: Output of the possible moves on a given state from neighbors function*

## 2.3. Transition model

Implementing the resulting moves from the neighbors function and identifying the relationship between actions and states is done by the transition model. transition_model(s, a) takes a state and an action, applies the action on this state and returns the resulting state (see Figure 10).



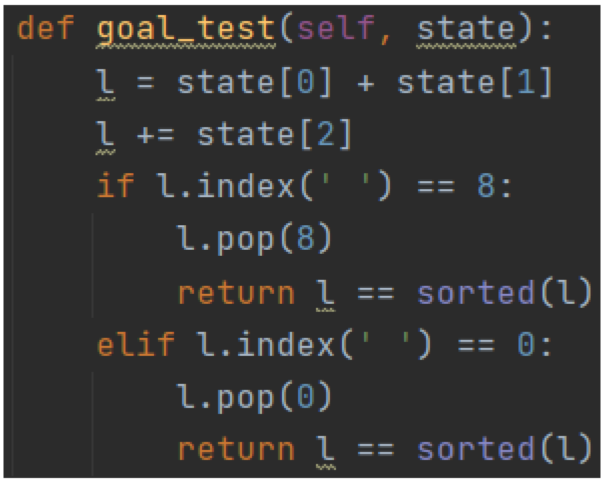
```
def transition_model(self, state, action):
    newblankpos = (action[1][0], action[1][1])
    if action[0] == 'up':
        state[newblankpos[0] + 1][newblankpos[1]] = state[newblankpos[0]][newblankpos[1]]
    elif action[0] == 'down':
        state[newblankpos[0] - 1][newblankpos[1]] = state[newblankpos[0]][newblankpos[1]]
    elif action[0] == 'right':
        state[newblankpos[0]][newblankpos[1] - 1] = state[newblankpos[0]][newblankpos[1]]
    elif action[0] == 'left':
        state[newblankpos[0]][newblankpos[1] + 1] = state[newblankpos[0]][newblankpos[1]]

    state[newblankpos[0]][newblankpos[1]] = ' '
    '''for p in state:
        print(p)
        print()'''
    return state
```

*Figure 10: Transition model*

The function assigns the future blank position into newblankpos then looks at what move is to be performed to make a swap operation between the blank and the tile it will be moving to.

## 2.4. Goal test

Two goal states were given to the model. Both states have the numbers ordered but with the blank either at the start or the end. The function goal_test(s) takes an input state s and returns true if it is a goal state, otherwise false (see figure 11).

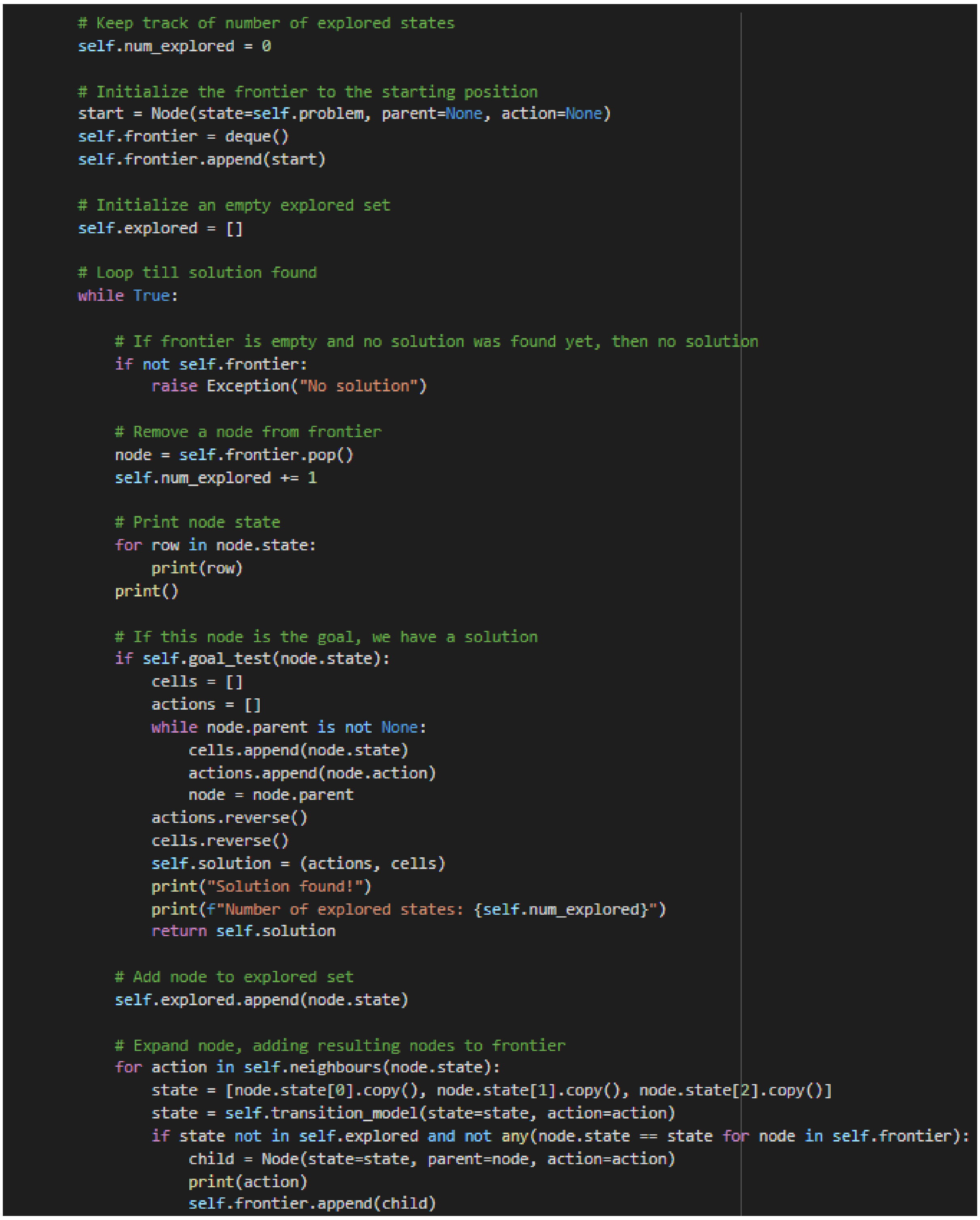


```python
def goal_test(self, state):
    l = state[0] + state[1]
    l += state[2]
    if l.index(' ') == 8:
        l.pop(8)
        return l == sorted(l)
    elif l.index(' ') == 0:
        l.pop(0)
        return l == sorted(l)
```

*Figure 11: Goal test*

## 2.5. Solution

Now that all the essential functions to the solving algorithm were discussed, the solving algorithm (which was illustrated in the pseudocode) can be examined (see Figure 12).



```
        # Keep track of number of explored states
        self.num_explored = 0

        # Initialize the frontier to the starting position
        start = Node(state=self.problem, parent=None, action=None)
        self.frontier = deque()
        self.frontier.append(start)

        # Initialize an empty explored set
        self.explored = []

        # Loop till solution found
        while True:

            # If frontier is empty and no solution was found yet, then no solution
            if not self.frontier:
                raise Exception("No solution")

            # Remove a node from frontier
            node = self.frontier.pop()
            self.num_explored += 1

            # Print node state
            for row in node.state:
                print(row)
            print()

            # If this node is the goal, we have a solution
            if self.goal_test(node.state):
                cells = []
                actions = []
                while node.parent is not None:
                    cells.append(node.state)
                    actions.append(node.action)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                print("Solution found!")
                print(f"Number of explored states: {self.num_explored}")
                return self.solution

            # Add node to explored set
            self.explored.append(node.state)

            # Expand node, adding resulting nodes to frontier
            for action in self.neighbours(node.state):
                state = [node.state[0].copy(), node.state[1].copy(), node.state[2].copy()]
                state = self.transition_model(state=state, action=action)
                if state not in self.explored and not any(node.state == state for node in self.frontier):
                    child = Node(state=state, parent=node, action=action)
                    print(action)
                    self.frontier.append(child)
```

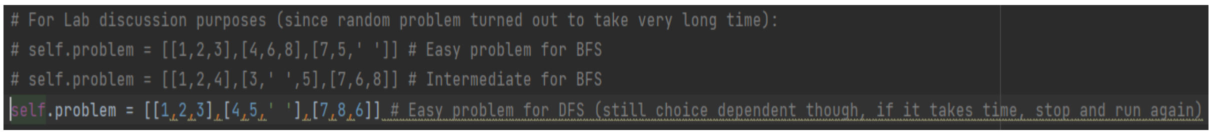
*Figure 12: Code implementation of the solving algorithm*

The frontier is initialized to the initial state (the randomly generated problem from the constructor) and an empty set to keep the explored states is also initialized. All the rest of the code is looped over till either finding a solution or no solution. In case that the frontier becomes empty before ever returning a solution, an exception is raised to indicate that there is no solution to the problem. Otherwise, the algorithm continues to remove a node from the frontier and putting it in the variable “node”. Notice that the number of explored states is constantly being tracked. Next, we look at this node whether it contains the goal state. If so, two empty lists are initialized to contain all the states and actions that originally led to this solution, discarding any unnecessarily explored states in the way of finding the solution. A back-tracing operation begins by appending the node’s state and action to these lists then assigning the node its predecessor each time. Then, the two lists are reversed to be in the logical order from the start of the problem to its end. These two lists are returned in a tuple together to be printed. Coming back to the case that the state in the node was not a goal state, this state is added to the explored set. After that, the node is expanded by looking in the valid actions that can be performed in its current state and by using the transition model. If the resulting nodes are not already in the explored set or in the frontier, they are added to the frontier.

The following section will present the results of the simulation.

## Results

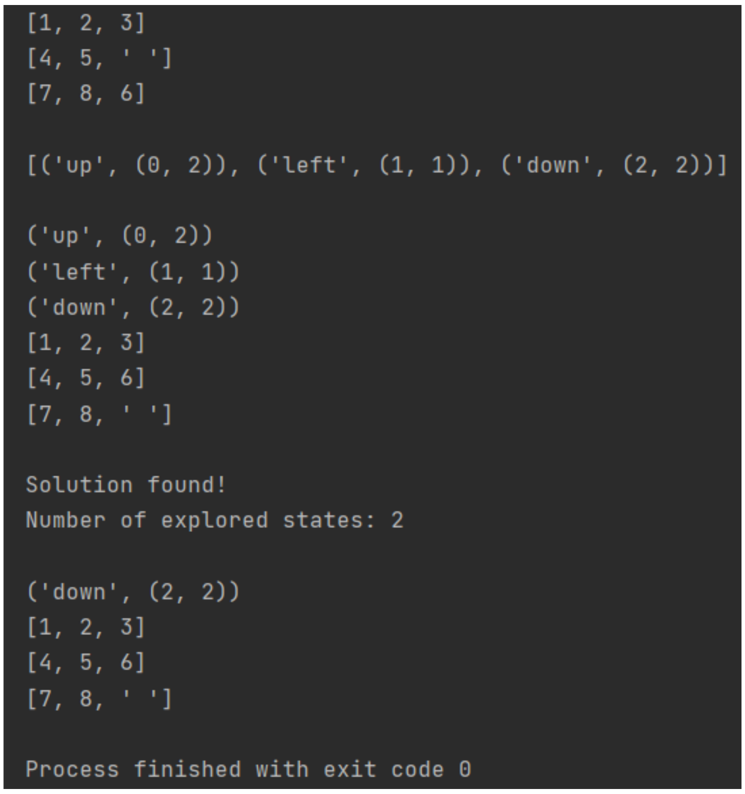
In this section, the simulation’s results will be discussed. For this section’s purpose, we used easy, close to goal tests to give a clearer illustration. This was done by overriding the randomly generated problem with an assignment at the end of the constructor (see Figure 13)



```
# For Lab discussion purposes (since random problem turned out to take very long time):
# self.problem = [[1,2,3],[4,6,8],[7,5,' ']] # Easy problem for BFS
# self.problem = [[1,2,4],[3,' ',5],[7,6,8]] # Intermediate for BFS
self.problem = [[1,2,3],[4,5,' '],[7,8,6]] # Easy problem for DFS (still choice dependent though, if it takes time, stop and run again)
```

*Figure 13: Problem overriding*

### 1. Depth-First search

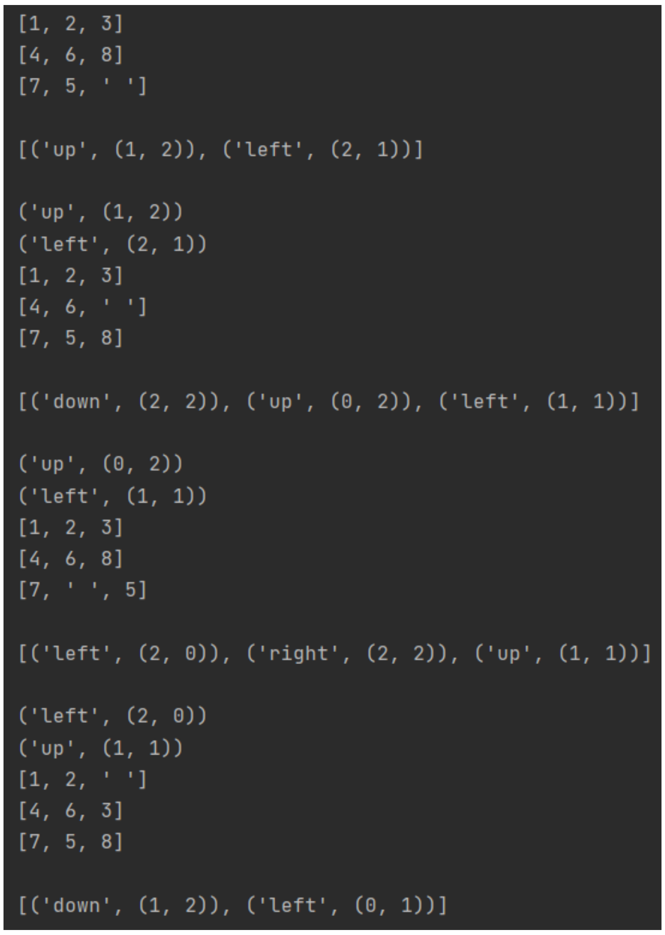


```
[1, 2, 3]
[4, 5, ' ']
[7, 8, 6]

[('up', (0, 2)), ('left', (1, 1)), ('down', (2, 2))]

('up', (0, 2))
('left', (1, 1))
('down', (2, 2))
[1, 2, 3]
[4, 5, 6]
[7, 8, ' ']

Solution found!
Number of explored states: 2

('down', (2, 2))
[1, 2, 3]
[4, 5, 6]
[7, 8, ' ']

Process finished with exit code 0
```

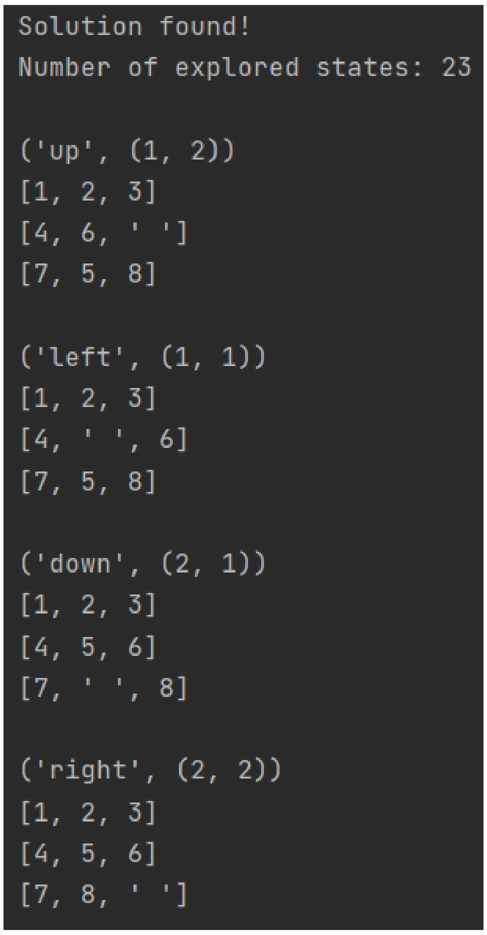
*Figure 14: DFS output*

In figure 14, the first matrix is the initial state. Below it is the list returned by the neighbors(s) function. The actions which did not lead to a previously explored are then added to the frontier and displayed in order. So, the resulting state from the down action is at the top of the stack. Directly below these actions is the following state after popping the stack. The blank is indeed moved downwards from the position (1, 2) to the position (2, 2). Since this resulting state is a goal state, therefore the program outputs that a solution was found, the number of explored states, the actions and the states that led to the solution.

## 2. Breadth-First search



```
[1, 2, 3]
[4, 6, 8]
[7, 5, ' ']

[('up', (1, 2)), ('left', (2, 1))]

('up', (1, 2))
('left', (2, 1))
[1, 2, 3]
[4, 6, ' ']
[7, 5, 8]

[('down', (2, 2)), ('up', (0, 2)), ('left', (1, 1))]

('up', (0, 2))
('left', (1, 1))
[1, 2, 3]
[4, 6, 8]
[7, ' ', 5]

[('left', (2, 0)), ('right', (2, 2)), ('up', (1, 1))]

('left', (2, 0))
('up', (1, 1))
[1, 2, ' ']
[4, 6, 3]
[7, 5, 8]

[('down', (1, 2)), ('left', (0, 1))]
```



```
Solution found!
Number of explored states: 23

('up', (1, 2))
[1, 2, 3]
[4, 6, ' ']
[7, 5, 8]

('left', (1, 1))
[1, 2, 3]
[4, ' ', 6]
[7, 5, 8]

('down', (2, 1))
[1, 2, 3]
[4, 5, 6]
[7, ' ', 8]

('right', (2, 2))
[1, 2, 3]
[4, 5, 6]
[7, 8, ' ']
```

*Figure 15: BFS output*

Figure 15 (left) shows the first part of the output (as it passed over many states to get the solution). At the first state on the top left, the available actions are up and left. Up and left are inserted into the queue respectively. Since Breadth-First explores the shallowest nodes first, both directions will be explored in order. Breadth-First explores level by level in the search tree. This is shown in the second and third matrices in the left part of the figure. The tuples that are printed after the valid actions list are not connected to the following state directly, but they are the actions that will be implemented from these points when going in their deeper level. The fourth matrix is the result of performing up action on the (initial state + up action). Analyzing the output shows that indeed, nodes are being explored level by level.

After the solution was found, the number of explored states was displayed, and the sequence of states and actions that did lead to the solution.

## Discussion and Conclusion

Depth-First and Breadth-First search algorithms were implemented on the Eight Puzzle. There are some cases where the Eight Puzzle has no solution when the tiles are placed in certain configurations. Random puzzle generation can include these cases, which will result in returning no solution after a long run.

After running the search algorithms, the following points were realized:

- DFS may (but not always) take much longer time and space than BFS and this is especially shown in the case where the initial state is very close to the goal. This is logical due to the nature of DFS where it explores the deepest node, so it continues all the way down one path to its end. Meanwhile, BFS explores each level of the search tree (the shallowest node), so it may find the close solution much faster than DFS if DFS chooses to follow different paths from the ones leading to the goal.
- It was also noticed that DFS may not find the optimal solution. This was realized from one of the runs when the final solution returned was the result of a very long path taken with more than 5000 moves after an approximate 31 minutes of execution time on Google.
- Both BFS and DFS take a very long time in solving the problem. In this implementation, depth limits and costs would have greatly addressed this issue if they were added. Additionally, this also proves that informed search algorithms are better for this type of problems such as A*. With a good heuristic, A* can reach the optimal solutions [1].

## References

[1] R. E. Korf, “Recent progress in the design and analysis of admissible heuristic functions,” *Lecture Notes in Computer Science*, pp. 45–55, 2000.

# Appendices

## Appendix A

Here is the entire code for the simulation:

```python
from collections import deque
import random
from math import floor

class Node():
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action

class Puzzle():
    def __init__(self):
        ''' Generate random 8 puzzle '''
        # Generate a randomized list of unrepeated numbers from 1 to 8
        self.problem = random.sample(range(1, 9), 8)

        # Choose a random location for the blank and insert it into the
list
        self.start = random.randint(0, 8)
        self.problem.insert(self.start, ' ')

        # Transform the list from 1D to 2D (3x3 list of lists)
        self.problem = [self.problem[x:x + 3] for x in range(0,
len(self.problem), 3)]

        # Calculate the new position of blank
        self.start = (floor(self.start / 3), self.start % 3)
        if self.start[0] == 3:
            self.start = (2, self.start[1])

        self.solution = None

        # For Lab discussion purposes (since random problem turned out to
take very long time):
        self.problem = [[1,2,3],[4,6,8],[7,5,' ']] # Easy problem for BFS
        # self.problem = [[1,2,4],[3,' ',5],[7,6,8]] # Intermediate for BFS
        # self.problem = [[1,2,3],[4,5,' '],[7,8,6]] # Easy problem for DFS
(still choice dependent though, if it takes time, stop and run again)

        #for p in self.problem:
            #print(p)
        #print(self.start)


    def neighbours(self, state):
        blankpos = None
        for r in state:
            if ' ' in r:
                blankpos = (state.index(r), r.index(' '))

        row, col = blankpos
        candidates = {
            ("up", (row - 1, col)),
            ("down", (row + 1, col)),
            ("left", (row, col - 1)),
            ("right", (row, col + 1))
        }

        result = []
        for action, (r, c) in candidates:
            if 0 <= r < 3 and 0 <= c < 3:
                result.append((action, (r, c)))
        print(result)
        print()
        return result

    def transition_model(self, state, action):
        newblankpos = (action[1][0], action[1][1])
        if action[0] == 'up':
            state[newblankpos[0] + 1][newblankpos[1]] =
state[newblankpos[0]][newblankpos[1]]
        elif action[0] == 'down':
            state[newblankpos[0] - 1][newblankpos[1]] =
state[newblankpos[0]][newblankpos[1]]
        elif action[0] == 'right':
            state[newblankpos[0]][newblankpos[1] - 1] =
state[newblankpos[0]][newblankpos[1]]
        elif action[0] == 'left':
            state[newblankpos[0]][newblankpos[1] + 1] =
state[newblankpos[0]][newblankpos[1]]

        state[newblankpos[0]][newblankpos[1]] = ' '
        '''for p in state:
            print(p)
            print()'''
        return state

    def goal_test(self, state):
        l = state[0] + state[1]
        l += state[2]
        if l.index(' ') == 8:
            l.pop(8)
            return l == sorted(l)
        elif l.index(' ') == 0:
            l.pop(0)
            return l == sorted(l)

    def DFS(self):
        # Keep track of number of explored states
        self.num_explored = 0

        # Initialize the frontier to the starting position
        start = Node(state=self.problem, parent=None, action=None)
        self.frontier = deque()
        self.frontier.append(start)

        # Initialize an empty explored set
        self.explored = []

        # Loop till solution found
        while True:

            # If frontier is empty and no solution was found yet, then no
solution
            if not self.frontier:
                raise Exception("No solution")
```

```python
            # Remove a node from frontier
            node = self.frontier.pop()
            self.num_explored += 1

            # Print node state
            for row in node.state:
                print(row)
            print()

            # If this node is the goal, we have a solution
            if self.goal_test(node.state):
                cells = []
                actions = []
                while node.parent is not None:
                    cells.append(node.state)
                    actions.append(node.action)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                print("Solution found!")
                print(f"Number of explored states: {self.num_explored}")
                return self.solution

            # Add node to explored set
            self.explored.append(node.state)

            # Expand node, adding resulting nodes to frontier
            for action in self.neighbours(node.state):
                state = [node.state[0].copy(), node.state[1].copy(),
node.state[2].copy()]
                state = self.transition_model(state=state, action=action)
                if state not in self.explored and not any(node.state ==
state for node in self.frontier):
                    child = Node(state=state, parent=node, action=action)
                    print(action)
                    self.frontier.append(child)

    def BFS(self):
        # Keep track of number of explored states
        self.num_explored = 0

        # Initialize the frontier to the starting position
        start = Node(state=self.problem, parent=None, action=None)
        self.frontier = deque()
        self.frontier.append(start)

        # Initialize an empty explored set
        self.explored = []

        # Loop till solution found
        while True:

            # If frontier is empty and no solution was found yet, then no
solution
            if not self.frontier:
                raise Exception("No solution")

            # Remove a node from frontier
            node = self.frontier.popleft()
            self.num_explored += 1

            # Print node state
            for row in node.state:
                print(row)
            print()

            # If this node is the goal, we have a solution
            if self.goal_test(node.state):
                cells = []
                actions = []
                while node.parent is not None:
                    cells.append(node.state)
                    actions.append(node.action)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                print("Solution found!")
                print(f"Number of explored states: {self.num_explored}")
                return self.solution

            # Add node to explored set
            self.explored.append(node.state)

            # Expand node, adding resulting nodes to frontier
            for action in self.neighbours(node.state):
                state = [node.state[0].copy(), node.state[1].copy(),
node.state[2].copy()]
                state = self.transition_model(state=state, action=action)
                if state not in self.explored and not any(node.state ==
state for node in self.frontier):
                    child = Node(state=state, parent=node, action=action)
                    print(action)
                    self.frontier.append(child)



p = Puzzle()

# sol = p.DFS()
sol = p.BFS()


i = 0
for action in sol[0]:
    print()
    print(action)
    for state in sol[1][i]:
        print(state)
    i += 1
```