8 Puzzle Solver

Your Name	ID
Ahmed Osama	3
Kamal Rashid	47
Mohamed Saad	51

Overview

This problem appeared as a project in the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI). In this assignment an agent will be implemented to solve the 8-puzzle.

An instance of the 8-puzzle game consists of a board holding 8 distinct movable tiles, plus an empty space. For any such board, the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment, the blank space is going to be represented with the number0.

Given an initial state of the board, the search problem is to find a sequence of moves that transitions this state to the goal state; that is, the configuration with all tiles arranged in ascending order 0,1,2,3,4,5,6,7,8.

The search space is the set of all possible states reachable from the initial state. The blank space may be swapped with a component in one of the four directions 'Up', 'Down', 'Left', 'Right', one move at a time. The cost of moving from one configuration of the board to another is the same and equal to one. Thus, the total cost of a path is equal to the number of moves made from the initial state to the goal state.

Goals

1. Implement the 8 puzzle search problem using the three search: BFS ,DFS and A*
2. For the A* (the informed search) use Manhattan heuristic and Euclidean heuristic and compare between number of nodes expanded and output paths, and to report which heuristic is more admissible.

Easy Puzzle Sample Run

```
initialState = [[2, 5, 4], [3, 0, 8], [1, 6, 7]]
goalState = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

2	5	4
3		8
1	6	7

	1	2
3	4	5
6	7	8

1. DFS

```
DFS
path to goal:
root
[2, 5, 4]
[3, 0, 8]
[1, 6, 7]
left
[2, 5, 4]
[0, 3, 8]
[1, 6, 7]
down
[2, 5, 4]
[1, 3, 8]
[0, 6, 7]
right
[2, 5, 4]
[1, 3, 8]
[6, 0, 7]
right
[2, 5, 4]
[1, 3, 8]
[6, 7, 0]
up
[2, 5, 4]
[1, 3, 0]
[6, 7, 8]
left
[2, 5, 4]
[1, 0, 3]
[6, 7, 8]
left
[2, 5, 4]
[0, 1, 3]
[6, 7, 8]
up
[0, 5, 4]
[2, 1, 3]
[6, 7, 8]
right
[5, 0, 4]
[2, 1, 3]
[6, 7, 8]
down
[5, 1, 4]
[2, 0, 3]
[6, 7, 8]
left
[5, 1, 4]
[0, 2, 3]
[6, 7, 8]
down
[5, 1, 4]
[6, 2, 3]
[0, 7, 8]
right
[5, 1, 4]
[6, 2, 3]
[7, 0, 8]
up
[5, 1, 4]
[6, 0, 3]
[7, 2, 8]
right
[5, 1, 4]
[6, 3, 0]
[7, 2, 8]
down
[5, 1, 4]
[6, 3, 8]
[7, 2, 0]
left
[5, 1, 4]
[6, 3, 8]
[7, 0, 2]
left
[5, 1, 4]
[6, 3, 8]
[0, 7, 2]
up
[5, 1, 4]
[0, 3, 8]
[6, 7, 2]
right
[5, 1, 4]
[3, 0, 8]
[6, 7, 2]
right
[5, 1, 4]
[3, 8, 0]
[6, 7, 2]
down
[5, 1, 4]
[3, 8, 2]
[6, 7, 0]
left
[5, 1, 4]
[3, 8, 2]
[6, 0, 7]
left
[5, 1, 4]
[3, 8, 2]
[0, 6, 7]
up
[5, 1, 4]
[0, 8, 2]
[3, 6, 7]
up
[0, 1, 4]
[5, 8, 2]
[3, 6, 7]
right
[1, 0, 4]
[5, 8, 2]
[3, 6, 7]
down
[1, 8, 4]
[5, 0, 2]
[3, 6, 7]
right
[1, 8, 4]
[5, 2, 0]
[3, 6, 7]
up
[1, 8, 0]
[5, 2, 4]
[3, 6, 7]
left
[1, 0, 8]
[5, 2, 4]
[3, 6, 7]
down
[1, 2, 8]
[5, 0, 4]
[3, 6, 7]
right
[1, 2, 8]
[5, 4, 0]
[3, 6, 7]
up
[1, 2, 0]
[5, 4, 8]
[3, 6, 7]
```

```
left
[1, 0, 2]
[5, 4, 8]
[3, 6, 7]
down
[1, 4, 2]
[5, 0, 8]
[3, 6, 7]
left
[1, 4, 2]
[0, 5, 8]
[3, 6, 7]
down
[1, 4, 2]
[3, 5, 8]
[0, 6, 7]
right
[1, 4, 2]
[3, 5, 8]
[6, 0, 7]
right
[1, 4, 2]
[3, 5, 8]
[6, 7, 0]
up
[1, 4, 2]
[3, 5, 0]
[6, 7, 8]
```

```
left
[1, 4, 2]
[3, 0, 5]
[6, 7, 8]
up
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]
left
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
cost of path:  44
nodes expanded:  86719
search depth:  50
running time:  1269.2394526 s
```

2. BFS

```
BFS
path to goal:
root
[2, 5, 4]
[3, 0, 8]
[1, 6, 7]
left
[2, 5, 4]
[0, 3, 8]
[1, 6, 7]
down
[2, 5, 4]
[1, 3, 8]
[0, 6, 7]
right
[2, 5, 4]
[1, 3, 8]
[6, 0, 7]
right
[2, 5, 4]
[1, 3, 8]
[6, 7, 0]
up
[2, 5, 4]
[1, 3, 0]
[6, 7, 8]
up
[2, 5, 0]
[1, 3, 4]
[6, 7, 8]
left
[2, 0, 5]
[1, 3, 4]
[6, 7, 8]
left
[0, 2, 5]
[1, 3, 4]
[6, 7, 8]
down
[1, 2, 5]
[0, 3, 4]
[6, 7, 8]
right
[1, 2, 5]
[3, 0, 4]
[6, 7, 8]
right
[1, 2, 5]
[3, 4, 0]
[6, 7, 8]
up
[1, 2, 0]
[3, 4, 5]
[6, 7, 8]
left
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]
left
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
cost of path:  14
nodes expanded:  5977
search depth:  14
running time:  13.059432 s
```

3. A* Euclidean Distance

```
A* using Euclidean Distance

path to goal:
[2, 5, 4]
[3, 0, 8]
[1, 6, 7]

[2, 5, 4]
[0, 3, 8]
[1, 6, 7]

[2, 5, 4]
[1, 3, 8]
[0, 6, 7]

[2, 5, 4]
[1, 3, 8]
[6, 0, 7]

[2, 5, 4]
[1, 3, 8]
[6, 7, 0]

[2, 5, 4]
[1, 3, 0]
[6, 7, 8]

[2, 5, 0]
[1, 3, 4]
[6, 7, 8]

[2, 0, 5]
[1, 3, 4]
[6, 7, 8]

[0, 2, 5]
[1, 3, 4]
[6, 7, 8]

[1, 2, 5]
[0, 3, 4]
[6, 7, 8]

[1, 2, 5]
[3, 0, 4]
[6, 7, 8]

[1, 2, 5]
[3, 4, 0]
[6, 7, 8]

[1, 2, 0]
[3, 4, 5]
[6, 7, 8]

[1, 0, 2]
[3, 4, 5]
[6, 7, 8]

[0, 1, 2]
[3, 4, 5]
[6, 7, 8]


cost of path =  14.0

there are 49 nodes expanded:

search depth =  14

running time =  0.01013 s
```

4. A* using Manhattan Distance

```
A* using Manhattan Distance

path to goal:
[2, 5, 4]
[3, 0, 8]
[1, 6, 7]

[2, 5, 4]
[0, 3, 8]
[1, 6, 7]

[2, 5, 4]
[1, 3, 8]
[0, 6, 7]

[2, 5, 4]
[1, 3, 8]
[6, 0, 7]

[2, 5, 4]
[1, 3, 8]
[6, 7, 0]
```

```
[2, 5, 4]
[1, 3, 0]
[6, 7, 8]

[2, 5, 0]
[1, 3, 4]
[6, 7, 8]

[2, 0, 5]
[1, 3, 4]
[6, 7, 8]

[0, 2, 5]
[1, 3, 4]
[6, 7, 8]

[1, 2, 5]
[0, 3, 4]
[6, 7, 8]

[1, 2, 5]
[3, 0, 4]
[6, 7, 8]
```

```
[1, 2, 5]
[3, 4, 0]
[6, 7, 8]

[1, 2, 0]
[3, 4, 5]
[6, 7, 8]

[1, 0, 2]
[3, 4, 5]
[6, 7, 8]

[0, 1, 2]
[3, 4, 5]
[6, 7, 8]


cost of path =  14

there are 43 nodes expanded:

search depth =  14

running time =  0.0 s
```

1st hard Puzzle Sample Run for A* Only

```
initialState = [[6, 4, 7], [8, 5, 0], [3, 2, 1]]
goalState = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

6	4	7
8	5	
3	2	1

1	2	3
4	5	6
7	8	

1. A* using Manhattan Distance

```
D:\pythonProject\8Puzzle_AI\Scripts\python.exe D:/pythonProject/main.py
A* using Manhattan Distance

path to goal:
[6, 4, 7]
[8, 5, 0]
[3, 2, 1]

[6, 4, 0]
[8, 5, 7]
[3, 2, 1]

[6, 0, 4]
[8, 5, 7]
[3, 2, 1]

[6, 5, 4]
[8, 0, 7]
[3, 2, 1]

[6, 5, 4]
[0, 8, 7]
[3, 2, 1]

[6, 5, 4]
[3, 8, 7]
[0, 2, 1]
```

```
[6, 5, 4]
[3, 8, 7]
[2, 0, 1]

[6, 5, 4]
[3, 8, 7]
[2, 1, 0]

[6, 5, 4]
[3, 8, 0]
[2, 1, 7]

[6, 5, 4]
[3, 0, 8]
[2, 1, 7]

[6, 0, 4]
[3, 5, 8]
[2, 1, 7]

[0, 6, 4]
[3, 5, 8]
[2, 1, 7]

[3, 6, 4]
[0, 5, 8]
[2, 1, 7]

[3, 6, 4]
[2, 5, 8]
[0, 1, 7]

[3, 6, 4]
[2, 5, 8]
[1, 0, 7]

[3, 6, 4]
[2, 5, 8]
[1, 7, 0]

[3, 6, 4]
[2, 5, 0]
[1, 7, 8]

[3, 6, 0]
[2, 5, 4]
[1, 7, 8]

[3, 0, 6]
[2, 5, 4]
[1, 7, 8]

[0, 3, 6]
[2, 5, 4]
[1, 7, 8]

[2, 3, 6]
[0, 5, 4]
[1, 7, 8]

[2, 3, 6]
[1, 5, 4]
[0, 7, 8]

[2, 3, 6]
[1, 5, 4]
[7, 0, 8]

[2, 3, 6]
[1, 0, 4]
[7, 5, 8]

[2, 3, 6]
[1, 4, 0]
[7, 5, 8]

[2, 3, 0]
[1, 4, 6]
[7, 5, 8]

[2, 0, 3]
[1, 4, 6]
[7, 5, 8]

[0, 2, 3]
[1, 4, 6]
[7, 5, 8]

[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


cost of path =  31

there are 6262 nodes expanded:

search depth =  31

running time =  41569396000 ns
```

2. A* Euclidean Distance

```
D:\pythonProject\8Puzzle_AI\Scripts\python.exe D:/pythonProject/main.py
A* using Euclidean Distance

path to goal:
[6, 4, 7]
[8, 5, 0]
[3, 2, 1]

[6, 4, 7]
[8, 5, 1]
[3, 2, 0]

[6, 4, 7]
[8, 5, 1]
[3, 0, 2]

[6, 4, 7]
[8, 5, 1]
[0, 3, 2]

[6, 4, 7]
[0, 5, 1]
[8, 3, 2]

[0, 4, 7]
[6, 5, 1]
[8, 3, 2]
```

```
[4, 0, 7]
[6, 5, 1]
[8, 3, 2]

[4, 7, 0]
[6, 5, 1]
[8, 3, 2]

[4, 7, 1]
[6, 5, 0]
[8, 3, 2]

[4, 7, 1]
[6, 0, 5]
[8, 3, 2]

[4, 0, 1]
[6, 7, 5]
[8, 3, 2]

[4, 1, 0]
[6, 7, 5]
[8, 3, 2]

[4, 1, 5]
[6, 7, 0]
[8, 3, 2]
```

```
[4, 1, 5]
[6, 7, 2]
[8, 3, 0]

[4, 1, 5]
[6, 7, 2]
[8, 0, 3]

[4, 1, 5]
[6, 0, 2]
[8, 7, 3]

[4, 1, 5]
[0, 6, 2]
[8, 7, 3]

[4, 1, 5]
[8, 6, 2]
[0, 7, 3]

[4, 1, 5]
[8, 6, 2]
[7, 0, 3]

[4, 1, 5]
[8, 0, 2]
[7, 6, 3]
```

```
[4, 1, 5]
[0, 8, 2]
[7, 6, 3]

[0, 1, 5]
[4, 8, 2]
[7, 6, 3]

[1, 0, 5]
[4, 8, 2]
[7, 6, 3]

[1, 5, 0]
[4, 8, 2]
[7, 6, 3]

[1, 5, 2]
[4, 8, 0]
[7, 6, 3]

[1, 5, 2]
[4, 8, 3]
[7, 6, 0]

[1, 5, 2]
[4, 8, 3]
[7, 0, 6]
```

```
[1, 5, 2]
[4, 0, 3]
[7, 8, 6]

[1, 0, 2]
[4, 5, 3]
[7, 8, 6]

[1, 2, 0]
[4, 5, 3]
[7, 8, 6]

[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


cost of path =  31.0

there are 38314 nodes expanded:

search depth =  31

running time =  2014501274200 ns
```

2nd Hard Puzzle Sample Run for A* Only

```
initialState = [[8, 6, 7], [2, 5, 4], [3, 0, 1]]
goalState = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

8	6	7
2	5	4
3		1

1	2	3
4	5	6
7	8	

1. A* using Manhattan Distance

```
D:\pythonProject\8Puzzle_AI\Scripts\python.exe D:/pythonProject/main.py
A* using Manhattan Distance

path to goal:
[8, 6, 7]
[2, 5, 4]
[3, 0, 1]

[8, 6, 7]
[2, 5, 4]
[0, 3, 1]

[8, 6, 7]
[0, 5, 4]
[2, 3, 1]

[8, 6, 7]
[5, 0, 4]
[2, 3, 1]

[8, 0, 7]
[5, 6, 4]
[2, 3, 1]

[8, 7, 0]
[5, 6, 4]
[2, 3, 1]
```

```
[8, 7, 4]
[5, 6, 0]
[2, 3, 1]

[8, 7, 4]
[5, 6, 1]
[2, 3, 0]

[8, 7, 4]
[5, 6, 1]
[2, 0, 3]

[8, 7, 4]
[5, 0, 1]
[2, 6, 3]

[8, 7, 4]
[0, 5, 1]
[2, 6, 3]

[0, 7, 4]
[8, 5, 1]
[2, 6, 3]

[7, 0, 4]
[8, 5, 1]
[2, 6, 3]
```

```
[7, 4, 0]
[8, 5, 1]
[2, 6, 3]

[7, 4, 1]
[8, 5, 0]
[2, 6, 3]

[7, 4, 1]
[8, 5, 3]
[2, 6, 0]

[7, 4, 1]
[8, 5, 3]
[2, 0, 6]

[7, 4, 1]
[8, 5, 3]
[0, 2, 6]

[7, 4, 1]
[0, 5, 3]
[8, 2, 6]

[0, 4, 1]
[7, 5, 3]
[8, 2, 6]
```

```
[4, 0, 1]
[7, 5, 3]
[8, 2, 6]

[4, 1, 0]
[7, 5, 3]
[8, 2, 6]

[4, 1, 3]
[7, 5, 0]
[8, 2, 6]

[4, 1, 3]
[7, 0, 5]
[8, 2, 6]

[4, 1, 3]
[7, 2, 5]
[8, 0, 6]

[4, 1, 3]
[7, 2, 5]
[0, 8, 6]

[4, 1, 3]
[0, 2, 5]
[7, 8, 6]
```

```
[0, 1, 3]
[4, 2, 5]
[7, 8, 6]

[1, 0, 3]
[4, 2, 5]
[7, 8, 6]

[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


cost of path =  31

there are 6330 nodes expanded:

search depth =  31

running time =  48988673600 ns
```

2. A* Euclidean Distance

```
D:\pythonProject\8Puzzle_AI\Scripts\python.exe D:/pythonProject/main.py
A* using Euclidean Distance

path to goal:
[8, 6, 7]
[2, 5, 4]
[3, 0, 1]

[8, 6, 7]
[2, 5, 4]
[3, 1, 0]

[8, 6, 7]
[2, 5, 0]
[3, 1, 4]

[8, 6, 0]
[2, 5, 7]
[3, 1, 4]

[8, 0, 6]
[2, 5, 7]
[3, 1, 4]

[0, 8, 6]
[2, 5, 7]
[3, 1, 4]
```

```
[2, 8, 6]
[0, 5, 7]
[3, 1, 4]

[2, 8, 6]
[3, 5, 7]
[0, 1, 4]

[2, 8, 6]
[3, 5, 7]
[1, 0, 4]

[2, 8, 6]
[3, 0, 7]
[1, 5, 4]

[2, 0, 6]
[3, 8, 7]
[1, 5, 4]

[0, 2, 6]
[3, 8, 7]
[1, 5, 4]

[3, 2, 6]
[0, 8, 7]
[1, 5, 4]
```

```
[3, 2, 6]
[1, 8, 7]
[0, 5, 4]

[3, 2, 6]
[1, 8, 7]
[5, 0, 4]

[3, 2, 6]
[1, 8, 7]
[5, 4, 0]

[3, 2, 6]
[1, 8, 0]
[5, 4, 7]

[3, 2, 6]
[1, 0, 8]
[5, 4, 7]

[3, 0, 6]
[1, 2, 8]
[5, 4, 7]

[0, 3, 6]
[1, 2, 8]
[5, 4, 7]
```

```
[1, 3, 6]
[0, 2, 8]
[5, 4, 7]

[1, 3, 6]
[5, 2, 8]
[0, 4, 7]

[1, 3, 6]
[5, 2, 8]
[4, 0, 7]

[1, 3, 6]
[5, 2, 8]
[4, 7, 0]

[1, 3, 6]
[5, 2, 0]
[4, 7, 8]

[1, 3, 0]
[5, 2, 6]
[4, 7, 8]

[1, 0, 3]
[5, 2, 6]
[4, 7, 8]
```

```
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]

[1, 2, 3]
[0, 5, 6]
[4, 7, 8]

[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


cost of path =  31.0

there are 38385 nodes expanded:

search depth =  31

running time =  2212605952100 ns
```

Data Structure

1. List
2. Heapq
3. Tuples

Algorithms

1. Check if puzzle is solvable

```
initialState = [[8, 6, 7], [2, 5, 4], [3, 0, 1]]
goalState = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
initial1D = np.array(initialState).flatten()
initialInversionCount = 0
goal1D = np.array(goalState).flatten()
goalInversionCount = 0
for i in range(9):
    for j in range(i, 9):
        if initial1D[j] != 0 and initial1D[i] != 0 and (initial1D[i] > initial1D[j]):
            initialInversionCount += 1
for i in range(9):
    for j in range(i, 9):
        if goal1D[j] != 0 and goal1D[i] != 0 and (goal1D[i] > goal1D[j]):
            goalInversionCount += 1
if initialInversionCount % 2 != goalInversionCount % 2:
    print("unsolvable!!")
    sys.exit()
```

From this link:
https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

2. Euclidean Distance Calculation

```
def Euclidean_Distance(self, currentState, theGoalState):
    theH = 0
    for ci in range(3):
        for cj in range(3):
            if (currentState[ci][cj] == 0):
                continue
            for gi in range(3):
                for gj in range(3):
                    if (currentState[ci][cj] == theGoalState[gi][gj]):
                        theH = theH + math.sqrt((ci - gi)**2 + (cj - gj)**2)
    return theH
```

3. Manhattan Distance Calculation

```
def Manhattan_Distance(self, currentState, theGoalState):
    theH = 0
    for ci in range(3):
        for cj in range(3):
            if (currentState[ci][cj] == 0):
                continue
            for gi in range(3):
                for gj in range(3):
                    if (currentState[ci][cj] == theGoalState[gi][gj]):
                        theH = theH + abs(ci - gi) + abs(cj - gj)
    return theH
```

4. Getting Children in A*

```python
def getChildren(self, root, theGoalState):
    theChildren = []
    for i in range(3):
        for j in range(3):
            if (root.state[i][j] == 0):
                if (i > 0):
                    child = Node()
                    child.parent = root
                    child.g = root.g + 1
                    child.state = self.equals(root.state)
                    child.state[i][j] = child.state[i - 1][j]
                    child.state[i - 1][j] = 0
                    child.h = self.Manhattan_Distance(child.state, theGoalState)
                    child.f = child.g + child.h
                    theChildren.append(child)
                if (j > 0):
                    child = Node()
                    child.parent = root
                    child.g = root.g + 1
                    child.state = self.equals(root.state)
                    child.state[i][j] = child.state[i][j - 1]
                    child.state[i][j - 1] = 0
                    child.h = self.Manhattan_Distance(child.state, theGoalState)
                    child.f = child.g + child.h
                    theChildren.append(child)
                if (i < 2):
                    child = Node()
                    child.parent = root
                    child.g = root.g + 1
                    child.state = self.equals(root.state)
                    child.state[i][j] = child.state[i + 1][j]
                    child.state[i + 1][j] = 0
                    child.h = self.Manhattan_Distance(child.state, theGoalState)
                    child.f = child.g + child.h
                    theChildren.append(child)
                if (j < 2):
                    child = Node()
                    child.parent = root
                    child.g = root.g + 1
                    child.state = self.equals(root.state)
                    child.state[i][j] = child.state[i][j + 1]
                    child.state[i][j + 1] = 0
                    child.h = self.Manhattan_Distance(child.state, theGoalState)
                    child.f = child.g + child.h
                    theChildren.append(child)
                break
        if (root.state[i][j] == 0):
            break
    return theChildren
```

5. Getting Children in DFS and BFS

```python
def expand(self, node, frontier, explored):
    (row, col) = node.index
    # move up
    if row > 0:
        hold = self.equals(node.state)
        hold[row][col] = hold[row - 1][col]
        hold[row - 1][col] = 0
        new = Node1(state=hold, parent=node, action="up", index=(row - 1, col), depth=node.depth + 1)
        inFrontier = False
        for i in range(len(frontier)):
            if (new.state == frontier[i].state):
                inFrontier = True
                break
        inExplored = False
        for i in range(len(explored)):
            if (new.state == explored[i]):
                inExplored = True
                break
        if not (inExplored or inFrontier):
                frontier.append(new)
```

```python
    # move right
    if col < 2:
        hold = self.equals(node.state)
        hold[row][col] = hold[row][col + 1]
        hold[row][col + 1] = 0
        new = Node1(state=hold, parent=node, action="right", index=(row, col + 1), depth=node.depth + 1)
        inFrontier = False
        for i in range(len(frontier)):
            if (new.state == frontier[i].state):
                inFrontier = True
                break
        inExplored = False
        for i in range(len(explored)):
            if (new.state == explored[i]):
                inExplored = True
                break
        if not (inExplored or inFrontier):
            frontier.append(new)
```

```
# move down
if row < 2:
    hold = self.equals(node.state)
    hold[row][col] = hold[row + 1][col]
    hold[row + 1][col] = 0
    new = Node1(state=hold, parent=node, action="down", index=(row + 1, col), depth=node.depth + 1)
    inFrontier = False
    for i in range(len(frontier)):
        if (new.state == frontier[i].state):
            inFrontier = True
            break
    inExplored = False
    for i in range(len(explored)):
        if (new.state == explored[i]):
            inExplored = True
            break
    if not (inExplored or inFrontier):
        frontier.append(new)
```

```
# move left
if col > 0:
    hold = self.equals(node.state)
    hold[row][col] = hold[row][col - 1]
    hold[row][col - 1] = 0
    new = Node1(state=hold, parent=node, action="left", index=(row, col - 1), depth=node.depth + 1)
    inFrontier = False
    for i in range(len(frontier)):
        if (new.state == frontier[i].state):
            inFrontier = True
            break
    inExplored = False
    for i in range(len(explored)):
        if (new.state == explored[i]):
            inExplored = True
            break
    if not (inExplored or inFrontier):
        frontier.append(new)
return frontier
```

Difference between A* using Manhattan heuristic and Euclidean heuristic

1. Number Of Nodes Expanded

Manhattan < Euclidean

2. Output Path

Manhattan = Euclidean

3. Which Heuristic Is More Admissible

Manhattan