# CS 205: Artificial Intelligence
# Assignment 1

**Name** : Yash Aggarwal
**SID** : 862333037
**Email** : yagga004@ucr.edu
**NetID** : yagga004
**Date** : 14/May/2023

In completing this assignment I consulted:

1. The Project Statement Handout provided.
    a. https://d1u36hdvoy9y69.cloudfront.net/cs-205-ai/Project_1_The_Eight_Puzzle_CS_205.pdf
2. Depth 31 Puzzles were taken from
    a. https://www.researchgate.net/figure/8-Puzzle-problem-instances_tbl1_280545587
3. To check if a puzzle is solvable or not in constant time and by not exploring the entire search space. This was used only for generating puzzle problems and was not used in original search code.
    a. https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

All the code is original **Except**

- Using the math module to calculate the square root of a number
- Using the copy module to make deep copies of a node
- Using the time module to keep track of time
- Using the numpy module to generate random numbers
- Using the matplotlib module to plot graphs and figures
- To check if a puzzle is solvable or not in constant time and by not exploring the entire search space. This was used only for generating puzzle problems and was not used in original search code.
    - https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

**Link to Code**

- The code is available on github → https://github.com/yashUcr773/CS_205_AI/tree/main/Projects/Project%201
- The code can also be run on google colab → **https://colab.research.google.com/drive/18yNb0bLmRX-0jsQMP5Q_JEtD-tOi4NMS**

**Outline of this report**

**CS 205: Artificial Intelligence**
**Project 1: The 8-Puzzle**
**Project Report**

**Name** : Yash Aggarwal
**SID**: 862333037

## 1. Introduction

A sliding tile puzzle, as seen in Figure 1, is a combinatorial puzzle that consists of numbers from 1 to N where N is a perfect square (4,9,16,25…) and the numbers are arranged in a N x N grid. The last tile is removed from the puzzle leaving 1 spot open so total tiles are from 1 to N-1. The goal of the puzzle is to move the tiles in such a way that all the numbers are arranged in ascending order and the empty spot is the last tile. The 8 puzzle is a sub-problem of the sliding puzzle with N set as 9 and tiles are numbered from 1 to 8.

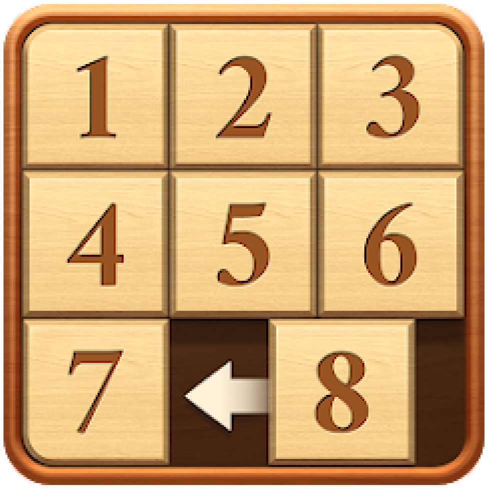


Figure 1: An 8 puzzle
Source: https://play.google.com/store/apps/details?id=com.gsr.npuzzle

This report details the findings for the 8-puzzle solved using different algorithms and a comparison between the algorithms. However, the code attached is generic and and be used to solve any NxN puzzle with any goal state.
In this project we attempt to solve the 8-puzzle using

1) Uniform Cost Search Algorithm
2) A* Algorithm with Misplaced Tile Heuristic
3) A* Algorithm with Manhattan Distance Heuristic

The project and report are a requirement for completion of the course CS 205: Artificial Intelligence taken under professor Dr. Eamonn Keogh in the Spring Quarter of 2023 at the University of California, Riverside.

The Language of choice is Python (version 3.9.X) and additional imports include (matplotlib, numpy, time, math, copy and os). The Report also includes the original source code and link to run it.

Within this report the word puzzle and problem are used interchangeably and denote the state of the puzzle board provided by the user.
The Puzzles in this report are classified as Easy, Medium and Hard.

- Puzzles with depth <=9 are considered to be Easy.
- Puzzles with depth >= 10 and depth <= 19 are considered medium.
- Puzzles with depth >= 20 are considered Hard.

This is an informal metric for puzzle classification used in this report.

# 2. Algorithms

The algorithms used to solve the 8-puzzle are

1) Uniform Cost Search Algorithm
2) A* Algorithm with Misplaced Tile Heuristic
3) A* Algorithm with Manhattan Distance Heuristic

## 2.1 Uniform Cost Search Algorithm

Uniform Cost search, also known as the Uninformed search, is a search algorithm that assigns a uniform cost (say 0) as a cost of expansion of every node. This means that the total cost to expand a node is only its depth. This also means that the cost of expanding all the nodes at any given depth d is the same for all the nodes.

When compared to A* Search and while implementing the algorithm in the project,
g(n) = depth of the node
h(n) = 0
Total cost to expand a node = h(n) + g(n) = g(n)

## 2.2 A* Search Algorithm

A* search, also known as the Informed search, is a search algorithm that uses a heuristic or a cost function for each node to measure how farther or closer the expansion will take us to the solution. It is referred to as Informed search as we make informed decisions based on cost function for which nodes need to be expanded next. The node with the smallest cost is selected.
This means all the nodes have different cost of expansion that depends on their depth and how close to the final state we are.

For this project we have chosen 2 Heuristic functions.

1) Misplaced Tile Heuristic
2) Manhattan Distance Heuristic

### 2.2.1 Misplaced Tile Heuristic

The Misplaced tile heuristic or the hamming distance heuristic counts the total number of tiles that are not in their correct position when compared to the goal state. While calculating the heuristic, we do not consider the placeholder (blank, _, 0) in the calculations.
In Figure 2, we can see there are 5 tiles namely (1,2,5,6,8) that are in incorrect positions and only 3 tiles (3,4,7) that are in correct positions. Therefore the misplaced tile heuristic or the hamming heuristic for the problem is 5.

Let us say this node is at depth d.
The total cost of expansion of this node would be g(n) + h(n) = d + 5.
Instead of just d in case of uniform search.
With this heuristic cost function we could expand the nodes with the smallest cost and thus expand cheaper nodes first, leading to goal nodes in fewer expansions.

### 2.2.2 Manhattan Distance Heuristic

The Manhattan distance heuristic counts the minimum number of moves that would be required to move a tile to its correct place assuming there are no other tiles on the board. While calculating the heuristic, we do not consider the placeholder (blank, _, 0) in the calculations.
In Figure 2, we can see there are 5 tiles namely (1,2,5,6,8) that are in incorrect positions and it would take (1,2,2,2,3) moves respectively for each tile to be in the correct position. Therefore the Manhattan heuristic value is 10.

Let us say this node is at depth d.
The total cost of expansion of this node would be g(n) + h(n) = d + 10.
Instead of just d in case of uniform search.
With this heuristic cost function we could expand the nodes with the smallest cost and thus expand cheaper nodes first, leading to goal nodes in fewer expansions.

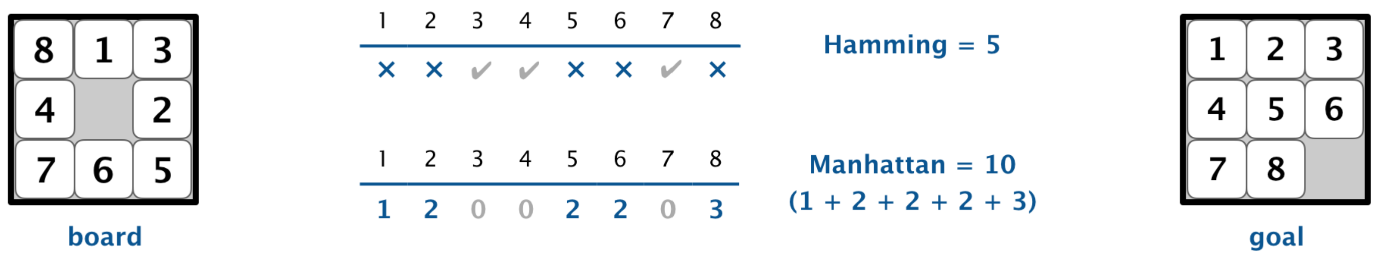
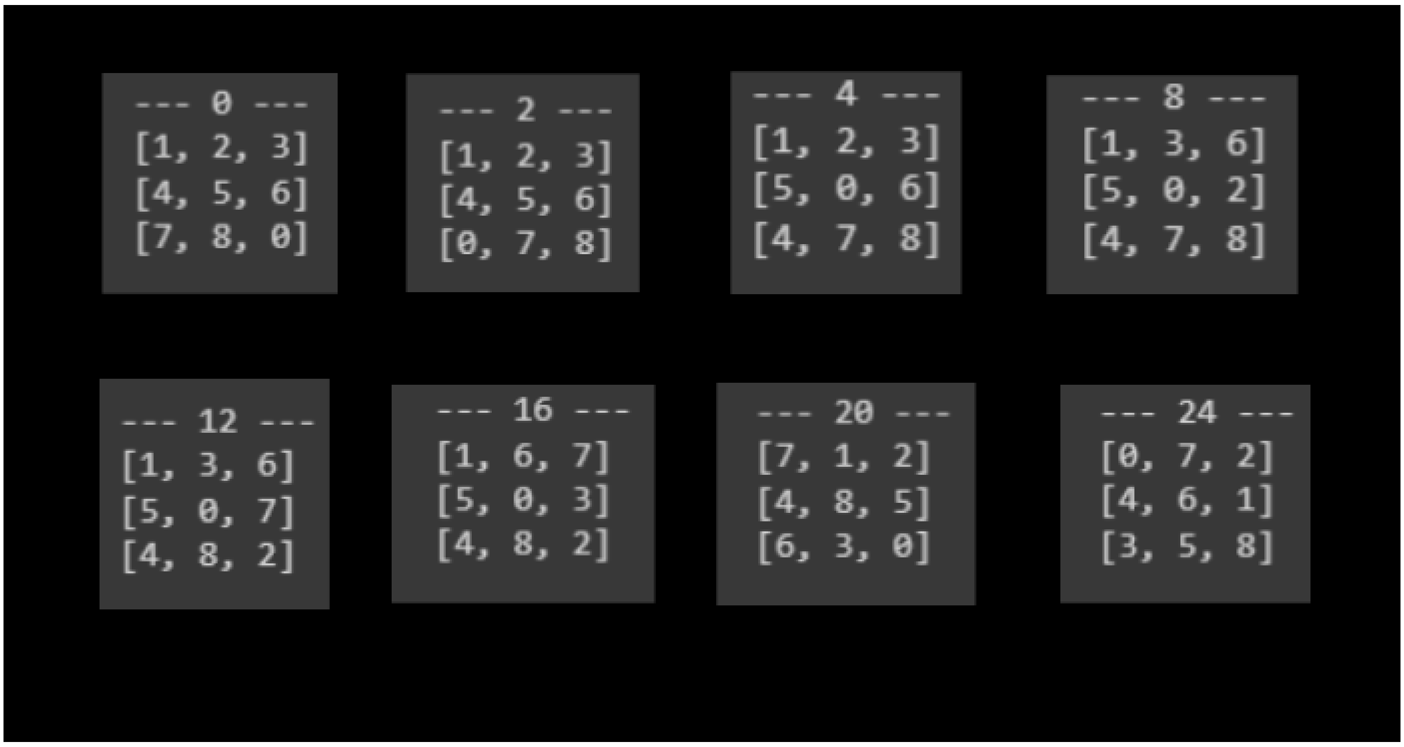


Figure 2: Misplaced Tile (Hamming) Heuristic and Manhattan Heuristic
Source: [https://coursera.cs.princeton.edu/algs4/assignments/8puzzle/hamming-manhattan.png](https://coursera.cs.princeton.edu/algs4/assignments/8puzzle/hamming-manhattan.png)

## 3. Comparison of Algorithms

We will now be running our program on various 8-puzzles of varied depths and visualizing the results.

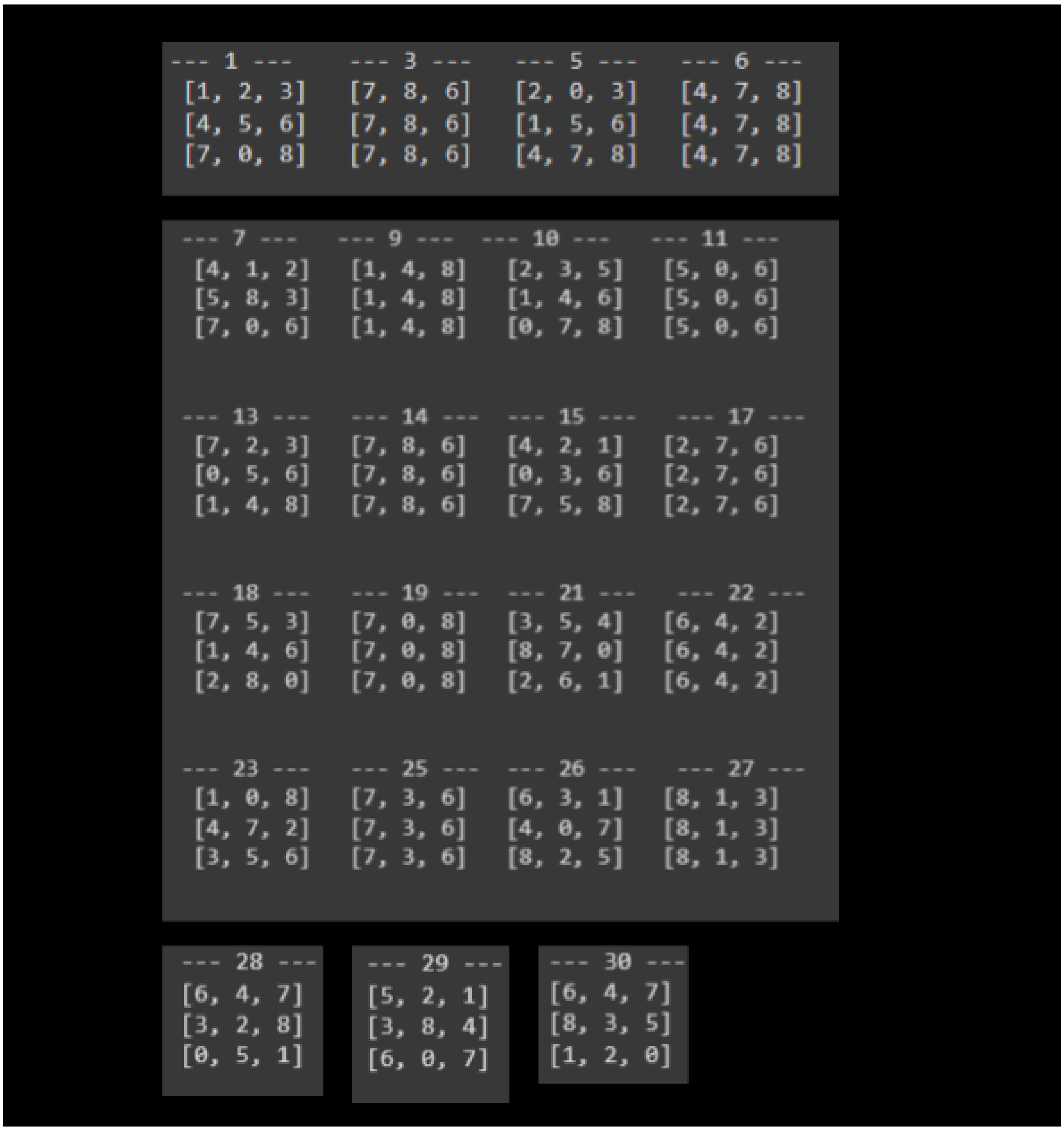
Figure 3 Denotes the list of puzzles used in testing. These were a good starting point to test the algorithms in early development stages.



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

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

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

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

--- 12 ---
[1, 3, 6]
[5, 0, 7]
[4, 8, 2]

--- 16 ---
[1, 6, 7]
[5, 0, 3]
[4, 8, 2]

--- 20 ---
[7, 1, 2]
[4, 8, 5]
[6, 3, 0]

--- 24 ---
[0, 7, 2]
[4, 6, 1]
[3, 5, 8]
```

Figure 3: List of Problem States provided by Professor Keogh

Figure 4 Denotes some of the puzzles that I created for other depths. These were generated by me using the code that is attached with this report. These were helpful in generating the reports for analysis.



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

```
 --- 7 ---    --- 9 ---   --- 10 ---   --- 11 ---
  [4, 1, 2]   [1, 4, 8]   [2, 3, 5]    [5, 0, 6]
  [5, 8, 3]   [1, 4, 8]   [1, 4, 6]    [5, 0, 6]
  [7, 0, 6]   [1, 4, 8]   [0, 7, 8]    [5, 0, 6]


 --- 13 ---   --- 14 ---  --- 15 ---    --- 17 ---
  [7, 2, 3]   [7, 8, 6]   [4, 2, 1]    [2, 7, 6]
  [0, 5, 6]   [7, 8, 6]   [0, 3, 6]    [2, 7, 6]
  [1, 4, 8]   [7, 8, 6]   [7, 5, 8]    [2, 7, 6]


 --- 18 ---   --- 19 ---  --- 21 ---    --- 22 ---
  [7, 5, 3]   [7, 0, 8]   [3, 5, 4]    [6, 4, 2]
  [1, 4, 6]   [7, 0, 8]   [8, 7, 0]    [6, 4, 2]
  [2, 8, 0]   [7, 0, 8]   [2, 6, 1]    [6, 4, 2]


 --- 23 ---   --- 25 ---  --- 26 ---    --- 27 ---
  [1, 0, 8]   [7, 3, 6]   [6, 3, 1]    [8, 1, 3]
  [4, 7, 2]   [7, 3, 6]   [4, 0, 7]    [8, 1, 3]
  [3, 5, 6]   [7, 3, 6]   [8, 2, 5]    [8, 1, 3]
```

```
 --- 28 ---    --- 29 ---    --- 30 ---
 [6, 4, 7]     [5, 2, 1]     [6, 4, 7]
 [3, 2, 8]     [3, 8, 4]     [8, 3, 5]
 [0, 5, 1]     [6, 0, 7]     [1, 2, 0]
```

Figure 4: List of Problem States generated by me. These are used to test the algorithms and run analysis.

In Figure 5, we can see the comparison of algorithms based on time taken to solve the puzzles at various depths. A* Search with Manhattan distance heuristic seems to be performing the best among the three algorithms. Even on the hardest puzzle with depth 31, the algorithm seems to take merely 20 seconds to search for the answer.

The second best algorithm seems to be A* search with Misplaced Tile heuristic. For easy to medium puzzles, The Misplaced Tile Heuristic takes little to no time. However as we move towards harder problems, we can see the sharp increase in time taken to run the algorithm with the hardest taking close to 750 seconds to run a problem with depth 31.

Finally the Uniform Search performed the worst among the three algorithms. Half way through the medium puzzles we can see it taking longer to search for the goal state with the algorithm taking around 700 seconds for the depth 31 puzzle. However, at the highest depths, we can see that Misplaced Tile heuristic takes more time than Uniform Cost Search.

The better performance of A* algorithms can be attributed to the cost function heuristic that helps the algorithm expand only the best tile closest to solution as opposed to uniform search that uses no cost function.

As for the manhattan and misplaced tile heuristics, the former runs better than the latter as it better estimates the cost function and thus expands the nodes closest to goal.

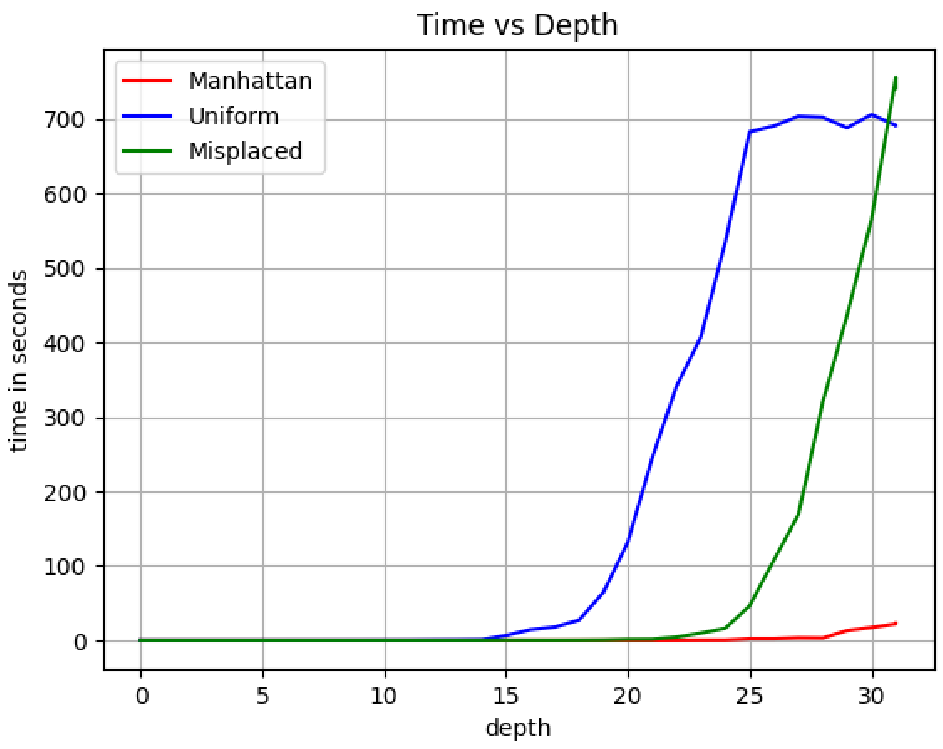


Figure 5: Comparison of all three algorithms based on time taken to reach goal state vs the depth of solution

In Figure 6, we can see the comparison of algorithms based on the number of nodes expanded to solve the puzzles of various depths. Similar results can be drawn here as Figure 5.

A* Search with Manhattan distance heuristic seems to be performing the best among the three algorithms. Even on the hardest puzzle with depth 31, the algorithm expands around 24,000 nodes to reach the goal state.

The second best algorithm seems to be A* search with Misplaced Tile heuristic. For easy to medium puzzles, The Misplaced Tile Heuristic expanded a low amount of nodes. However as we move towards harder problems, we can see the sharp increase in total nodes expanded with the hardest taking close to 150,000 nodes for a problem with depth 31

Finally the Uniform Search performed the worst among the three algorithms. Half way through the medium puzzles we can see it expanding a large number of nodes to search for the goal state with a maximum of 180,000 at depth 31

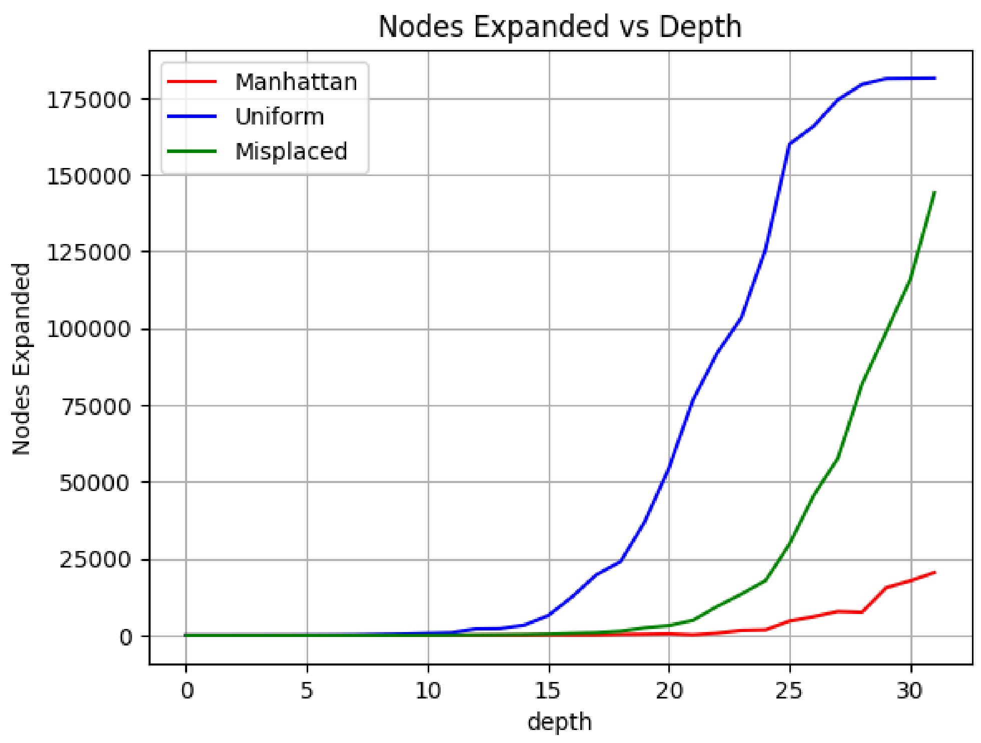


Figure 6: Comparison of all three algorithms based on number of nodes expanded to reach goal state vs the depth of solution

In Figure 7, we can see the comparison of algorithms based on the maximum length of the frontier queue at various solution depths. Similar results can be drawn here as Figure 5 and 6.

A* Search with Manhattan distance heuristic seems to be performing the best among the three algorithms. Even on the hardest puzzle with depth 31, the algorithm had a maximum of around 9,000 nodes in its queue.

The second best algorithm seems to be A* search with Misplaced Tile heuristic. For easy to medium puzzles, The Misplaced Tile Heuristic had a low amount of nodes in its queue. However as we move towards harder problems, we can see a sharp increase in queue size with the largest queue size of approximately 25,000 for a problem with depth 31

Finally the Uniform Search performed the worst among the three algorithms. At the start of medium puzzles, we can see it has a very large queue size and the trend continues as we move towards the hard puzzles with the hardest expanding close to 28000 nodes. At the very high depth (28, 29, 30, 31) we can see that Uniform search and Misplaced search have a similar max queue size.

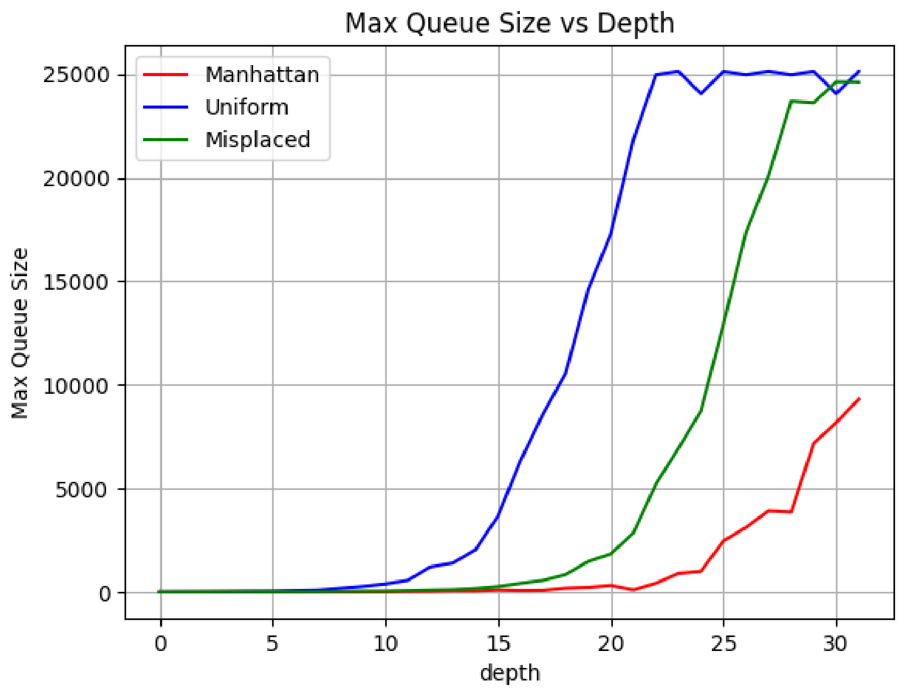


Figure 7: Comparison of all three algorithms based on maximum length of frontier queue vs the depth of solution

## 4. Conclusions

After testing the puzzles on all the algorithms, it was found that.

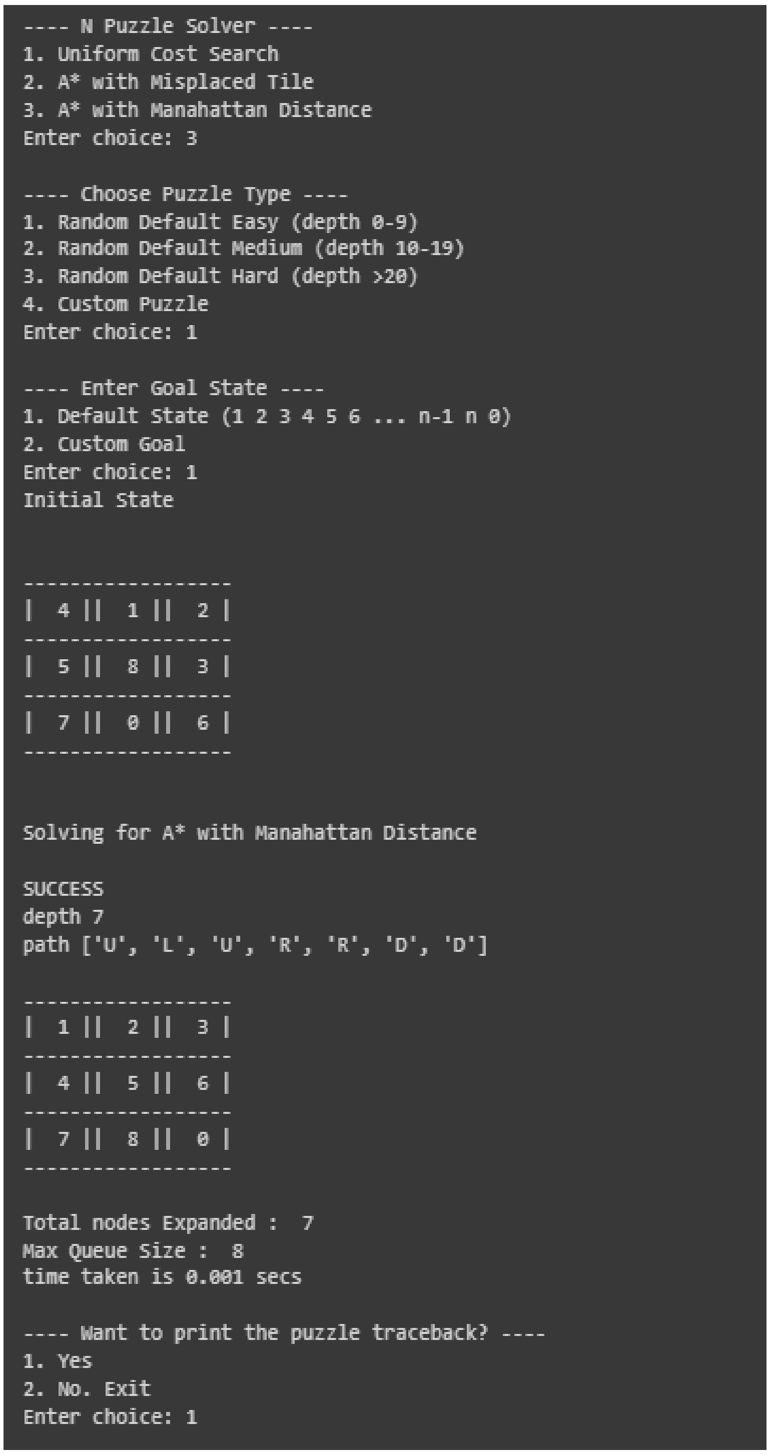
1. Clearly The A* Algorithm with Manhattan Distance Heuristic is the best algorithms from the three algorithms.
2. All the algorithms did similar in terms of time, nodes expanded and max length of queue for easier puzzles (depth < 10)
3. For the medium puzzles (depth >= 10 and depth <20), we see that uniform search takes a lot more time whereas Manhattan and Misplaced heuristics are still comparable. Same is true for total nodes expanded and Max nodes in queue.

4. For the Hard Puzzles (depth >= 20), we can see that uniform search seems to take a considerable amount of time with Misplaced Heuristic not far behind but Manhattan Heuristic still seems to be very fast taking less than 20 seconds for the hardest puzzle.
5. Manhattan Distance Heuristics outperforms Misplaced Tile and Uniform cost search by a large margin and Misplaced tile heuristic performs considerably better than Uniform Cost search
6. For the hardest Puzzles at depths 29, 30 and 31, we can see that Misplaced Tile search is a little slower than the Uniform cost search and  takes almost the same amount of memory. The only benefit at greater depths of using Misplaced Tile search instead of Uniform cost search is that it expands fewer nodes than Uniform Cost Search.

## 5. Traceback of easy puzzle

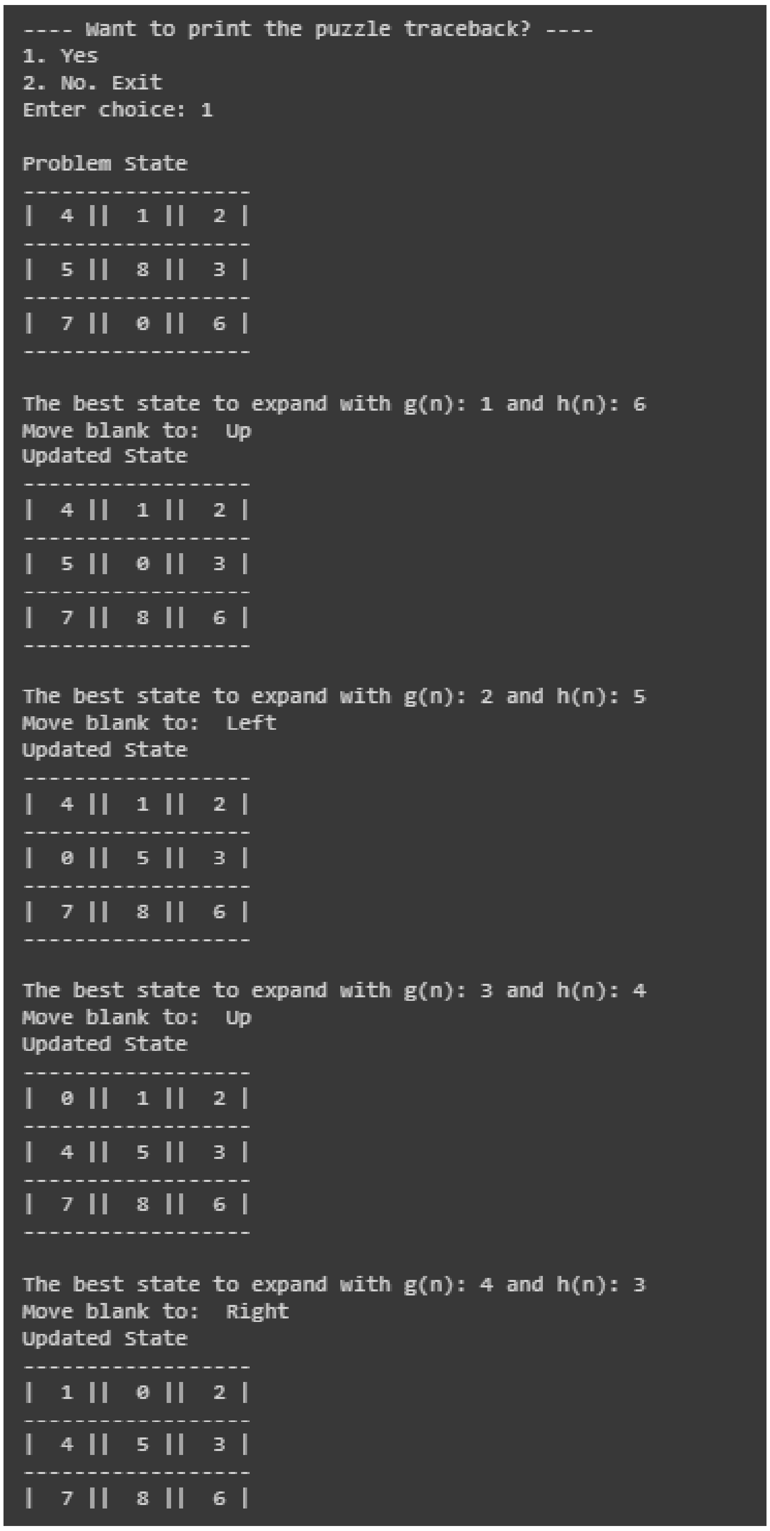
Figures 8, 9 and 10 show the code running for an easy problem and its traceback.

In Figure 8, we can see that by choosing A* Search with Manhattan distance heuristic on an easy puzzle of depth 7, we can find the solution in 0.001 secs with expanding only 7 nodes and having a max frontier size of 8



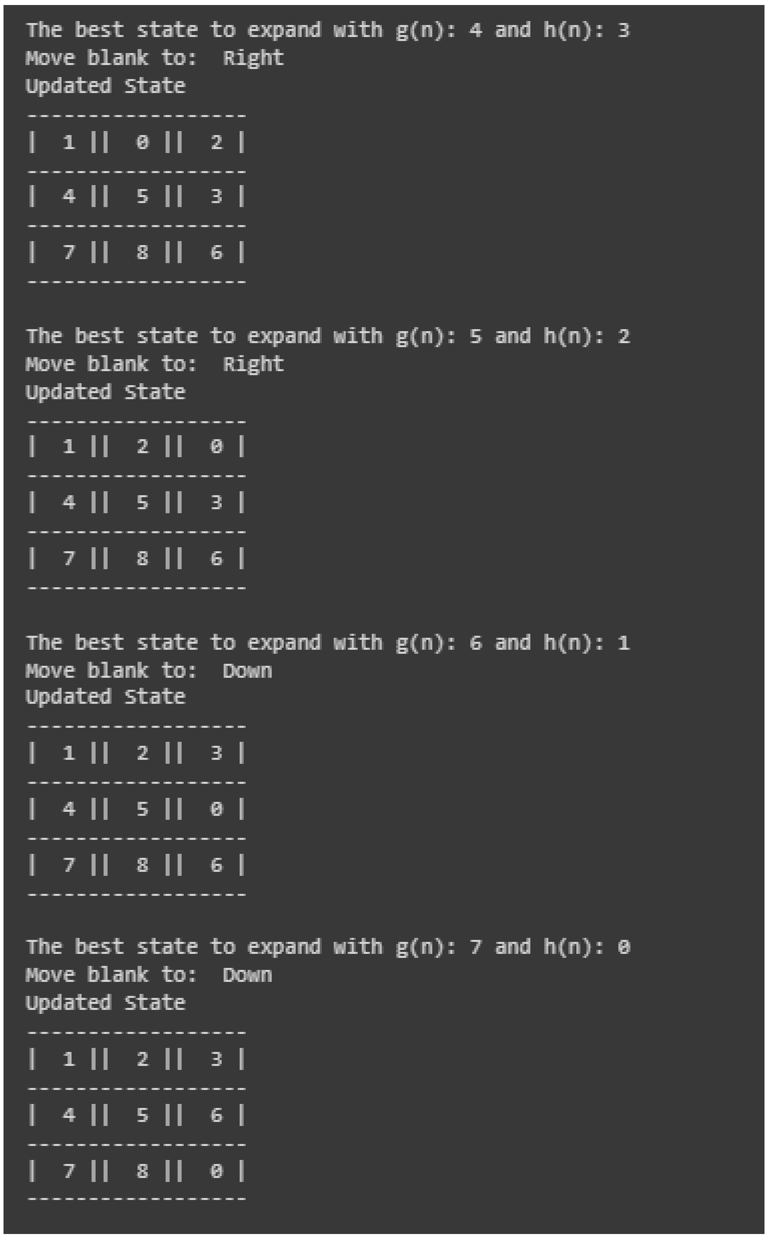
```
---- N Puzzle Solver ----
1. Uniform Cost Search
2. A* with Misplaced Tile
3. A* with Manahattan Distance
Enter choice: 3

---- Choose Puzzle Type ----
1. Random Default Easy (depth 0-9)
2. Random Default Medium (depth 10-19)
3. Random Default Hard (depth >20)
4. Custom Puzzle
Enter choice: 1

---- Enter Goal State ----
1. Default State (1 2 3 4 5 6 ... n-1 n 0)
2. Custom Goal
Enter choice: 1
Initial State


------------------
|  4 ||  1 ||  2 |
------------------
|  5 ||  8 ||  3 |
------------------
|  7 ||  0 ||  6 |
------------------


Solving for A* with Manahattan Distance

SUCCESS
depth 7
path ['U', 'L', 'U', 'R', 'R', 'D', 'D']

------------------
|  1 ||  2 ||  3 |
------------------
|  4 ||  5 ||  6 |
------------------
|  7 ||  8 ||  0 |
------------------

Total nodes Expanded :  7
Max Queue Size :  8
time taken is 0.001 secs

---- Want to print the puzzle traceback? ----
1. Yes
2. No. Exit
Enter choice: 1
```

Figure 8: Take Input from user, search to find the solution. Display the result. (Easy)



```
---- Want to print the puzzle traceback? ----
1. Yes
2. No. Exit
Enter choice: 1

Problem State
------------------
|  4 ||  1 ||  2 |
------------------
|  5 ||  8 ||  3 |
------------------
|  7 ||  0 ||  6 |
------------------

The best state to expand with g(n): 1 and h(n): 6
Move blank to:  Up
Updated State
------------------
|  4 ||  1 ||  2 |
------------------
|  5 ||  0 ||  3 |
------------------
|  7 ||  8 ||  6 |
------------------

The best state to expand with g(n): 2 and h(n): 5
Move blank to:  Left
Updated State
------------------
|  4 ||  1 ||  2 |
------------------
|  0 ||  5 ||  3 |
------------------
|  7 ||  8 ||  6 |
------------------

The best state to expand with g(n): 3 and h(n): 4
Move blank to:  Up
Updated State
------------------
|  0 ||  1 ||  2 |
------------------
|  4 ||  5 ||  3 |
------------------
|  7 ||  8 ||  6 |
------------------

The best state to expand with g(n): 4 and h(n): 3
Move blank to:  Right
Updated State
------------------
|  1 ||  0 ||  2 |
------------------
|  4 ||  5 ||  3 |
------------------
|  7 ||  8 ||  6 |
```

Figure 9: Print Traceback with best move and its cost for the problem in figure 8



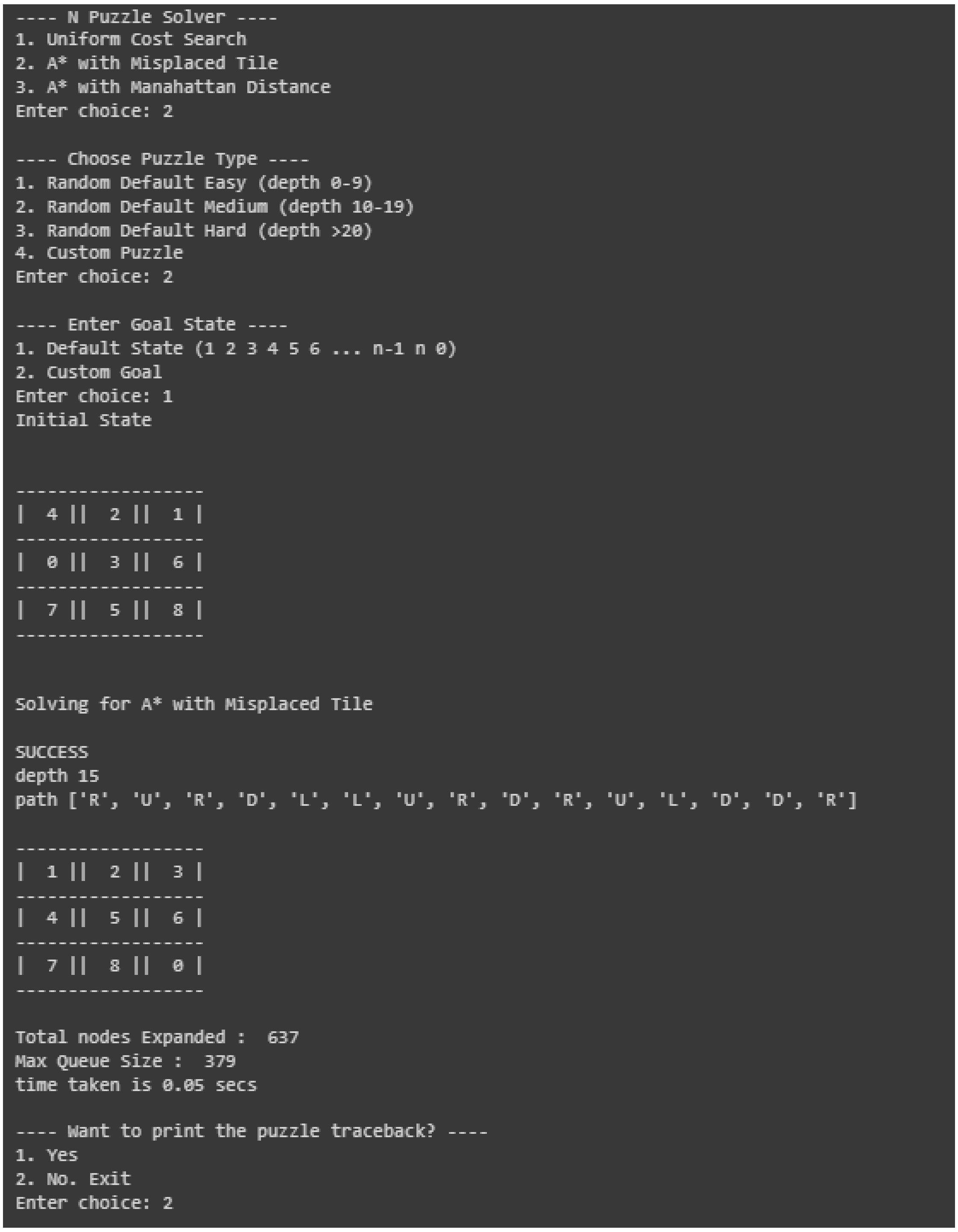
```
The best state to expand with g(n): 4 and h(n): 3
Move blank to:  Right
Updated State
-----------------
|  1 ||  0 ||  2 |
-----------------
|  4 ||  5 ||  3 |
-----------------
|  7 ||  8 ||  6 |
-----------------

The best state to expand with g(n): 5 and h(n): 2
Move blank to:  Right
Updated State
-----------------
|  1 ||  2 ||  0 |
-----------------
|  4 ||  5 ||  3 |
-----------------
|  7 ||  8 ||  6 |
-----------------

The best state to expand with g(n): 6 and h(n): 1
Move blank to:  Down
Updated State
-----------------
|  1 ||  2 ||  3 |
-----------------
|  4 ||  5 ||  0 |
-----------------
|  7 ||  8 ||  6 |
-----------------

The best state to expand with g(n): 7 and h(n): 0
Move blank to:  Down
Updated State
-----------------
|  1 ||  2 ||  3 |
-----------------
|  4 ||  5 ||  6 |
-----------------
|  7 ||  8 ||  0 |
-----------------
```

Figure 10: Continued Printing of Traceback from figure 9

# 6. Solution of a Medium Puzzle without Traceback

Figure 11 shows the code running for a medium problem without its traceback.

In Figure 11, we can see that by choosing A* Search with Misplaced tile distance heuristic on a medium puzzle of depth 15, we can find the solution in 0.05 secs with expanding 637 nodes and having a max frontier size of 379



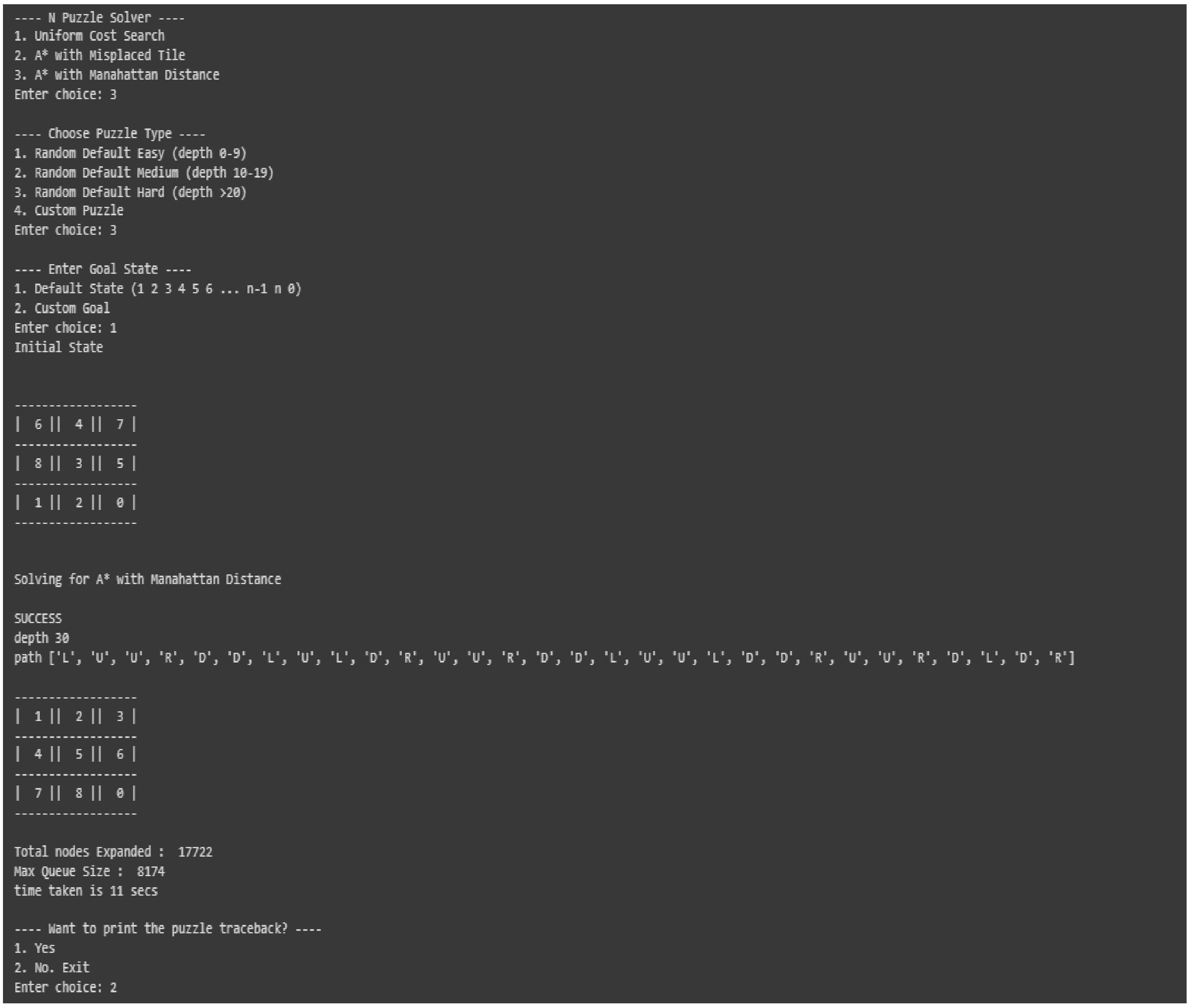
```
---- N Puzzle Solver ----
1. Uniform Cost Search
2. A* with Misplaced Tile
3. A* with Manahattan Distance
Enter choice: 2

---- Choose Puzzle Type ----
1. Random Default Easy (depth 0-9)
2. Random Default Medium (depth 10-19)
3. Random Default Hard (depth >20)
4. Custom Puzzle
Enter choice: 2

---- Enter Goal State ----
1. Default State (1 2 3 4 5 6 ... n-1 n 0)
2. Custom Goal
Enter choice: 1
Initial State


------------------
|  4 ||  2 ||  1 |
------------------
|  0 ||  3 ||  6 |
------------------
|  7 ||  5 ||  8 |
------------------


Solving for A* with Misplaced Tile

SUCCESS
depth 15
path ['R', 'U', 'R', 'D', 'L', 'L', 'U', 'R', 'D', 'R', 'U', 'L', 'D', 'D', 'R']

------------------
|  1 ||  2 ||  3 |
------------------
|  4 ||  5 ||  6 |
------------------
|  7 ||  8 ||  0 |
------------------

Total nodes Expanded :  637
Max Queue Size :  379
time taken is 0.05 secs

---- Want to print the puzzle traceback? ----
1. Yes
2. No. Exit
Enter choice: 2
```

Figure 11: Take Input from user, search to find the solution. Display the result. (Medium)

# 7. Solution of a Hard Puzzle without Traceback

Figure 12 shows the code running for a Hard problem without its traceback.

In Figure 12, we can see that by choosing A* Search with Manhattan distance heuristic on a hard puzzle of depth 30, we can find the solution in 11 secs with expanding 17722 nodes and having a max frontier size of 8174



```
---- N Puzzle Solver ----
1. Uniform Cost Search
2. A* with Misplaced Tile
3. A* with Manahattan Distance
Enter choice: 3

---- Choose Puzzle Type ----
1. Random Default Easy (depth 0-9)
2. Random Default Medium (depth 10-19)
3. Random Default Hard (depth >20)
4. Custom Puzzle
Enter choice: 3

---- Enter Goal State ----
1. Default State (1 2 3 4 5 6 ... n-1 n 0)
2. Custom Goal
Enter choice: 1
Initial State


-----------------
|  6 ||  4 ||  7 |
-----------------
|  8 ||  3 ||  5 |
-----------------
|  1 ||  2 ||  0 |
-----------------


Solving for A* with Manahattan Distance

SUCCESS
depth 30
path ['L', 'U', 'U', 'R', 'D', 'D', 'L', 'U', 'L', 'D', 'R', 'U', 'U', 'R', 'D', 'D', 'L', 'U', 'U', 'L', 'D', 'D', 'R', 'U', 'U', 'R', 'D', 'L', 'D', 'R']

-----------------
|  1 ||  2 ||  3 |
-----------------
|  4 ||  5 ||  6 |
-----------------
|  7 ||  8 ||  0 |
-----------------

Total nodes Expanded :  17722
Max Queue Size :  8174
time taken is 11 secs

---- Want to print the puzzle traceback? ----
1. Yes
2. No. Exit
Enter choice: 2
```

Figure 12: Take Input from user, search to find the solution. Display the result. (Hard)

# 8. Original Code

The original code can be found at

- The code is available on github → https://github.com/yashUcr773/CS_205_AI/tree/main/Projects/Project%201
- The code can also be run on google colab → **https://colab.research.google.com/drive/18yNb0bLmRX-0jsQMP5Q_JEtD-tOi4NMS**

Initial.py

```
'''
Solve 8 puzzle with
- Uniform cost search
- A* with misplaced tile
- A* with manhattan distance
extensible code
- can work for any N puzzle.
- can update heuristic function to use any custom function
'''

##################################################################
########           LIBRARY IMPORTS           ########
##################################################################

# for square root of numbers
import math
# clear output screen
import os
# to make deep copies of nodes' states
import copy as cpy
# to track the time used for execution
import time
# for creating random states
import numpy as np
# for plotting results and graphs
import matplotlib.pyplot as plt

##################################################################
########              CONSTANTS              ########
##################################################################
UNIFORM = 'Uniform'
MISPLACED = 'Misplaced'
MANHATTAN = 'Manhattan'

COLOR_MAP = {
    MANHATTAN: 'red',
    MISPLACED: 'green',
    UNIFORM: 'blue',
}

DIRECTIONS_MAP = {
    'R': 'Right',
    'L': 'Left',
    'U': 'Up',
    'D': 'Down'
}

#################################################################
########              LIST OF PUZZLES              ########
#################################################################
# Stores some randomly generated puzzles with puzzle state and true depth

list_of_easy_puzzles = [
    ([1, 2, 3, 4, 5, 6, 7, 8, 0], 0),
    ([1, 2, 3, 4, 5, 6, 7, 0, 8], 1),
    ([1, 2, 3, 4, 5, 6, 0, 7, 8], 2),
    ([1, 0, 3, 4, 2, 5, 7, 8, 6], 3),
    ([1, 2, 3, 5, 0, 6, 4, 7, 8], 4),
    ([2, 0, 3, 1, 5, 6, 4, 7, 8], 5),
    ([2, 5, 3, 1, 0, 6, 4, 7, 8], 6),
    ([4, 1, 2, 5, 8, 3, 7, 0, 6], 7),
    ([1, 3, 6, 5, 0, 2, 4, 7, 8], 8),
    ([2, 5, 3, 0, 7, 6, 1, 4, 8], 9),
]

list_of_medium_puzzles = [
    ([2, 3, 5, 1, 4, 6, 0, 7, 8], 10),
    ([1, 4, 2, 7, 8, 3, 5, 0, 6], 11),
    ([1, 3, 6, 5, 0, 7, 4, 8, 2], 12),
    ([7, 2, 3, 0, 5, 6, 1, 4, 8], 13),
    ([1, 5, 0, 3, 2, 4, 7, 8, 6], 14),
    ([4, 2, 1, 0, 3, 6, 7, 5, 8], 15),
    ([1, 6, 7, 5, 0, 3, 4, 8, 2], 16),
    ([4, 8, 1, 0, 3, 5, 2, 7, 6], 17),
    ([7, 5, 3, 1, 4, 6, 2, 8, 0], 18),
    ([5, 4, 6, 3, 1, 2, 7, 0, 8], 19),
]

list_of_hard_puzzles = [
    ([7, 1, 2, 4, 8, 5, 6, 3, 0], 20),
    ([3, 5, 4, 8, 7, 0, 2, 6, 1], 21),
    ([7, 1, 8, 5, 0, 3, 6, 4, 2], 22),
    ([1, 0, 8, 4, 7, 2, 3, 5, 6], 23),
    ([0, 7, 2, 4, 6, 1, 3, 5, 8], 24),
    ([5, 2, 1, 0, 8, 4, 7, 3, 6], 25),
    ([6, 3, 1, 4, 0, 7, 8, 2, 5], 26),
    ([4, 0, 7, 2, 6, 5, 8, 1, 3], 27),
    ([8, 6, 4, 2, 0, 7, 3, 1, 5], 28),
    ([5, 2, 1, 3, 8, 4, 6, 0, 7], 29),
    ([6, 4, 7, 8, 3, 5, 1, 2, 0], 30),
]

#################################################################
```

```
########            Utility Functions            ########
################################################################


def generate_random_states(state_len):
    '''
    generate random states for evaulating and testing
    '''
    arr = [i for i in range(state_len)]
    np.random.shuffle(arr)
    return arr


def validate_state(problem_state: list) -> bool:
    '''
    check if the state passed is valid or not.
    checks that state length is a perfect sqaure.
    checks that all numbers are unique.
    checks that numbers are in range 0-len(state)
    '''

    # get length of puzzle
    # is it 8 puzzle, 15 puzzle etc
    puzzle_size = len(problem_state)

    # check that length is perfect square
    sqrt = int(math.sqrt(puzzle_size))
    if(sqrt*sqrt) != puzzle_size:
        return False

    # generate valid inputs.
    # valid inputs range from 0 - n for n puzzle
    valid_inputs = [i for i in range(puzzle_size)]

    # put the problem state in a set to remove duplicates.
    problem_state_set = set(problem_state)

    for i in problem_state_set:
        if i not in valid_inputs:
            return False

    return True


def print_formatted_time(time_input):
    '''
    funtion to take in seconds as input and
    print in Hours, minutes, and seconds
    '''
    hrs = int(time_input // 3600)
    mins = int((time_input % 3600) // 60)
    secs = int((time_input % 3600) % 60)
    if hrs:
        print(f'time taken is {hrs} hrs, {mins} mins and {secs} secs')
    elif mins:
        print(f'time taken is {mins} mins and {secs} secs')
    else:
        print(f'time taken is {secs} secs')


def print_time(time_input):
    '''
    function to print the time with appropritate precision if between 0 and 1
    else print in HH, MM, SS format
    '''
    if time_input <= 1e-5:
        print(f'time taken is {time_input:.6f} secs')
    elif time_input <= 1e-4:
        print(f'time taken is {time_input:.5f} secs')
    elif time_input <= 1e-3:
        print(f'time taken is {time_input:.4f} secs')
    elif time_input <= 1e-2:
        print(f'time taken is {time_input:.3f} secs')
    elif time_input <= 1e-1:
        print(f'time taken is {time_input:.2f} secs')
    elif time_input >= 0 and time_input <= 1:
        print(f'time taken is {time_input} secs')
    else:
        print_formatted_time(time_input)


def print_trace(node, goal_state):
    '''
    take in any node and print the trace on how to reach this node from the parent node
    '''
    if node is None:
        return

    print_trace(node.parent, goal_state)
    node.print_trace_info(goal_state)

##############################################################
########    NODE CLASS AND CORRESPONDING FUNCTION    ########
##############################################################


class Node():

    '''
    create a node class for the states.
    stores path to current node from parent, depth etc and other properties.
    has uitlity methods.
    '''
    # to store the states that have already been generated.
    # prevents exploring repeating states.
    # If a state is present here, then it has already been generated at higher depth
    global_states_manager = {}

    # initialize the node
    # depth of node.
    # path stores the path to be taken to reach till current node.
    # state of node
    # link to parent node
    def __init__(self, depth, path, state, parent):
        self.depth = depth
        self.path = path
        self.state = state
        self.parent = parent

        # puzzle length 8/15/24
        self.state_length = len(state)
        # length per row
        self.row_length = int(math.sqrt(self.state_length))
        # length per column
        self.col_length = int(math.sqrt(self.state_length))

        # in case the current node is parent node, empty the globally stored states.
        if self.parent is None:
            Node.global_states_manager = {}
            Node.global_states_manager[self._get_state_string(
                self.state)] = self.depth

    def spawn_children(self):
        '''
        create child nodes after making valid moves on parent node.
        children are not generated is their states are already present in global states manager
        '''

        # get index of blank tile
        blank_idx = self.state.index(0)

        children_list = []

        for move in self.get_valid_moves():
            state_copy = cpy.deepcopy(self.state)
            path = cpy.deepcopy(self.path)

            if move == 'U':
                path.append('U')
                state_copy[blank_idx], state_copy[blank_idx -
                                                  self.row_length] = state_copy[blank_idx-self.row_length],
state_copy[blank_idx]

            elif move == 'L':
                path.append('L')
                state_copy[blank_idx], state_copy[blank_idx -
                                                1] = state_copy[blank_idx-1], state_copy[blank_idx]

            elif move == 'R':
                path.append('R')
                state_copy[blank_idx], state_copy[blank_idx +
                                                1] = state_copy[blank_idx+1], state_copy[blank_idx]

            elif move == 'D':
                path.append('D')
                state_copy[blank_idx], state_copy[blank_idx +
                                                self.row_length] = state_copy[blank_idx+self.row_length],
state_copy[blank_idx]

            # check if the node is already generated.
            past_depth_if_generated = self._is_state_already_generated(
                state_copy)

            # if the node is never generated or if the node generated earlier has
            # depth greater than current node, then generate another node with lower depth
            if past_depth_if_generated == -1 or past_depth_if_generated > self.depth+1:
                child_node = Node(self.depth+1, path, state_copy, self)
                Node.global_states_manager[self._get_state_string(
                    child_node.state)] = self.depth+1
                children_list.append(child_node)

        return children_list

    def get_valid_moves(self):
        '''
        get list of valid operators for each puzzle state
        checks the position of blank and returns array of valid moves possible
        '''

        # total Valid moves.
        # Move the blank space in following directions
        # Up, Left, Right, Down
        valid = ['U', 'L', 'R', 'D']

        # get index of blank tile
        blank_idx = self.state.index(0)

        # if the blank tile is in first row, cant move up
        if blank_idx >= 0 and blank_idx < self.col_length:
            valid.remove('U')

        # if the blank tile is in last row, cant move down
        if blank_idx >= (self.col_length*self.col_length - self.col_length) and blank_idx <
self.col_length*self.col_length:
            valid.remove('D')

        # if the blank tile is in first column, cant move left
        if blank_idx % self.col_length == 0:
            valid.remove('L')

        # if the blank tile is in last column, cant move right
        if (blank_idx + 1) % self.col_length == 0:
            valid.remove('R')

        return valid

    def manhattan_distance_heuristic(self, goal_state):
        '''
        get value for manhattan distance for a current state and goal state
        it is the shortest distance a tile needs to be moved to get to correct position
        total distance is sum of all individual distances
        does not include blank for calculation
        '''

        total_manhattan_distance = 0

        for value in goal_state:
            if value == 0:
                continue

            goal_state_row, goal_state_colums = self._get_row_col_position(
                goal_state, value)
            random_state_row, random_state_colums = self._get_row_col_position(
                self.state, value)
            total_manhattan_distance += abs(goal_state_colums-random_state_colums)+abs(
                goal_state_row-random_state_row)

        return int(total_manhattan_distance)

    def misplaced_tile_heuristic(self, goal_state):
        '''
        get value of misplaced tile heuristic for a current state and goal state.
        it is the count of all the tiles that are not in correct position
        does not include blank for calculation
        '''

        misplaced_count = 0
        for idx,value in enumerate(goal_state):
            if value == 0:
                continue

            if self.state[idx] != goal_state[idx]:
                misplaced_count += 1

        return misplaced_count

    def get_heuristic_cost(self, heuristic_measure, goal_state):
        '''
        get the heuristic value cost of expanding this node.
```

```
        takes in heuristic measure and goal state.
        return g(n) + h(n).
        g(n) is the depth of node.
        '''

        g_n = self.depth
        h_n = 0

        if heuristic_measure == MANHATTAN:
            h_n = self.manhattan_distance_heuristic(goal_state)
        elif heuristic_measure == MISPLACED:
            h_n = self.misplaced_tile_heuristic(goal_state)
        elif heuristic_measure == UNIFORM:
            h_n = 0
        else:
            h_n = 0

        return g_n + h_n

    def print_state(self, verbose=False):
        '''
        print the current node in puzzle view.
        if verbose is True, also print the path to reach this node
        and its depth
        '''

        if verbose:
            print('depth', self.depth)
            print('path', self.path)

        self._print_horizontal_divider(self.state_length)
        for i in range(self.state_length):
            print(f'| {self.state[i]:2} |', end="")
            if (i+1) % self.row_length == 0:
                self._print_horizontal_divider(self.state_length)
        print()

    def print_trace_info(self, goal_state):
        '''
        print the trace of the current node.
        start from the parent and print out the heuristic code and cumulative cost.
        print the path to take from parent to reach this node.
        '''

        if len(self.path):
            print(
                f'The best state to expand with g(n): {self.depth} and h(n):
{self.manhattan_distance_heuristic(goal_state)}')
            print('Move blank to: ', DIRECTIONS_MAP[self.path[-1]])
            print('Updated State', end='')
        else:
            print('\nProblem State', end='')
        self._print_horizontal_divider(self.state_length)
        for i in range(self.state_length):
            print(f'| {self.state[i]:2} |', end="")
            if (i+1) % self.row_length == 0:
                self._print_horizontal_divider(self.state_length)
        print()

    def _get_row_col_position(self, state, element):
        '''
        get row and column positions for a given element in a given state
        used for manhattan distance
        '''

        idx = state.index(element)
        column_val = int(idx % self.row_length)
        r_val = int(idx // self.row_length)
        return r_val, column_val

    def _print_horizontal_divider(self, size=8):
        '''
        print horizontal dividers after each row for better UI
        '''
        if size == 9:
            print('\n------------------')
        elif size == 16:
            print('\n------------------------')

    def _is_state_already_generated(self, state):
        '''
        check if the potential child state is already generated
        return the depth if state exists, else return -1
        '''
        state_string = "".join([str(i) for i in state])
        return Node.global_states_manager[state_string] if state_string in Node.global_states_manager else
-1

    def _get_state_string(self, state):
        '''
        convert the child node to a string for unique and easy representation
        '''
        state_string = "".join([str(i) for i in state])
        return state_string

############################################################
########               GENERAL SEARCH               ########
############################################################


def make_queue(node: Node, goal_state: list, heuristic_measure: str):
    '''
    Initialize an empty queue.
    take in a node and add it to the queue.
    return the queue
    '''
    return [(node.get_heuristic_cost(heuristic_measure, goal_state), node)]


def is_queue_empty(queue: list):
    '''
    take in queue and check if the queue is empty
    '''
    return False if len(queue) > 0 else True


def remove_front(queue: list):
    '''
    take in queue and sort it.
    remove the first node.
    return the first removed node.
    '''

    queue = sorted(queue, key=lambda x: x[0])
    node = queue.pop(0)[1]
    return queue, node


def expand_nodes(node: Node):
    '''
    takes in node and operators and expands the node based on operators.
    returns a list of nodes
    '''

    children = node.spawn_children()
    return children


def make_node_from_state(state: list):
    '''
    call in the Node class to create Parent Node
    '''
    parent_node = Node(0, [], state, None)
    return parent_node


def queueing_function(queue, children, heuristic_measure, goal_state):
    '''
    take in queue,
    take in children
    put children in priority queue based on heuristic value
    sort the queue
    return the queue
    '''
    child_queue = []
    for child in children:
        child_queue.append((child.get_heuristic_cost(
            heuristic_measure, goal_state), child))

    queue = queue + child_queue
    queue = sorted(queue, key=lambda x: x[0])

    return queue


def general_search(initial_state, goal_state, queueing_function, heuristic_measure, verbose=False):
    '''
    # general search function
    # refered from the problem statement doc provided.
    # link to doc
    # https://d1u36hdvoy9y69.cloudfront.net/cs-205-ai/Project_1_The_Eight_Puzzle_CS_205.pdf
    # takes in problem state, goal state, queueing function and heuristic measure
    # solves the problem using the queueing function
    # returns final node, total nodes expanded, max queue size
    '''

    # create parent node
    nodes = make_queue(make_node_from_state(initial_state),
                       goal_state, heuristic_measure)

    # store total nodes expanded count and max size of queue
    total_nodes_expanded = 0
    max_queue_size = 0

    while True:
        max_queue_size = max(max_queue_size, len(nodes))

        if is_queue_empty(nodes):
            print("FAILURE")
            return -1, total_nodes_expanded, max_queue_size
        else:
            nodes, node = remove_front(nodes)

            if goal_state == node.state:
                if verbose:
                    print("SUCCESS")
                    node.print_state(True)
                    print('Total nodes Expanded : ', total_nodes_expanded)
                    print('Max Queue Size : ', max_queue_size)
                return node, max_queue_size, total_nodes_expanded
            else:
                total_nodes_expanded += 1
                nodes = queueing_function(nodes, expand_nodes(
                    node), heuristic_measure, goal_state)
```

```
# ###############################################################
# ########  UI LANDING PAGE AND INPUT VALIDATION      ########
# ###############################################################


def main_block(clear_previous=True):
    '''
    Print out the Landing Page.
    Get Algo Choice From user
    Get Puzzle Choice From user
    Get Goal State From user
    Search the goal state
    Print Traceback
    '''

    ############### Clear screen for First View ###############
    if clear_previous:
        os.system('cls')


    #################     GET ALGO CHOICE     #################
    print('---- N Puzzle Solver ----')
    print('1. Uniform Cost Search')
    print('2. A* with Misplaced Tile')
    print('3. A* with Manahattan Distance')
    algo_choice = int(input('Enter choice: '))

    if algo_choice not in [1, 2, 3]:
        os.system('cls')
        print('Please enter correct choice.\n')
        main_block(clear_previous=False)
        return

    ################     GET PUZZLE CHOICE     ################
    print('\n---- Choose Puzzle Type ----')
    print('1. Random Default Easy (depth 0-9)')
    print('2. Random Default Medium (depth 10-19)')
    print('3. Random Default Hard (depth >20)')
    print('4. Custom Puzzle')
    puzzle_choice = int(input('Enter choice: '))

    if puzzle_choice not in [1, 2, 3, 4]:
        os.system('cls')
        print('Please enter correct choice.\n')
        main_block(clear_previous=False)
        return

    ################     INPUT CUSTOM PUZZLE     ################
    problem_state = []
    if puzzle_choice == 1:
        problem_state = list_of_easy_puzzles[np.random.choice(len(list_of_easy_puzzles))][0]
    elif puzzle_choice == 2:
```

```
    problem_state = list_of_medium_puzzles[np.random.choice(len(list_of_medium_puzzles))][0]
elif puzzle_choice == 3:
    problem_state = list_of_hard_puzzles[np.random.choice(len(list_of_hard_puzzles))][0]
elif puzzle_choice == 4:
    print('\nEnter the numbers in puzzle as space seperated list.')
    print('Represent blank with 0')
    print('For Example: 1 2 3 4 0 5 6 7 8\n')
    problem_input = input('Numbers: ')
    problem_state = problem_input.split(' ')

    # convert string to integers
    problem_state = [int(i) for i in problem_state]

    if not validate_state(problem_state):
        os.system('cls')
        print('Pleae enter valid input state.\n')
        main_block(clear_previous=False)
        return

################    GET GOAL STATE    ################
print('\n---- Enter Goal State ----')
print('1. Default State (1 2 3 4 5 6 ... n-1 n 0)')
print('2. Custom Goal')
puzzle_choice = int(input('Enter choice: '))

if puzzle_choice not in [1, 2]:
    os.system('cls')
    print('Please enter correct choice.\n')
    main_block(clear_previous=False)
    return

################    GET GOAL STATE    ################
goal_state = []
if puzzle_choice == 1:
    goal_state = list(range(1, len(problem_state)))
    goal_state.append(0)

elif puzzle_choice == 2:
    print('\nEnter the numbers in goal as space seperated list.')
    print('Represent blank with 0')
    print('For Example: 1 2 3 4 0 5 6 7 8\n')
    problem_input = input('Numbers: ')
    goal_state = problem_input.split(' ')

    # convert string to integers
    goal_state = [int(i) for i in goal_state]

    if not validate_state(goal_state):
        os.system('cls')
        print('Pleae enter valid input state.\n')
        main_block(clear_previous=False)
        return
    ################      FIND GOAL STATE USING SEARCH      ################
    os.system('cls')
    print('Initial State\n')
    parent_node = Node(0, [], problem_state, None)
    parent_node.print_state()

    if algo_choice == 1:
        print('\nSolving for Uniform cost\n')
        time_before = time.time()
        final_node, _, _ = general_search(problem_state, goal_state, queueing_function, UNIFORM,
verbose=True)
        time_after = time.time()
        total_time = time_after - time_before
        print_time(total_time)

    elif algo_choice == 2:
        print('\nSolving for A* with Misplaced Tile\n')
        time_before = time.time()
        final_node, _, _ = general_search(problem_state, goal_state, queueing_function, MISPLACED,
verbose=True)
        time_after = time.time()
        total_time = time_after - time_before
        print_time(total_time)


    elif algo_choice == 3:
        print('\nSolving for A* with Manahattan Distance\n')
        time_before = time.time()
        final_node, _, _ = general_search(problem_state, goal_state, queueing_function, MANHATTAN,
verbose=True)
        time_after = time.time()
        total_time = time_after - time_before
        print_time(total_time)

    ################      PRINT TRACEBACK      ################

    print('\n---- Want to print the puzzle traceback? ----')
    print('1. Yes')
    print('2. No. Exit')
    traceback_choice = int(input('Enter choice: '))

    if traceback_choice not in [1, 2]:
        os.system('cls')
        print('Please enter correct choice.\n')
        main_block(clear_previous=False)
        return

    if traceback_choice == 1:
        print_trace(final_node, goal_state)
    return

main_block()

# ###############################################################
# ########     Multiple TESTS and Result Analysis      ########
# ###############################################################
# code to run all the puzzles and generate graphs


def run_analysis(combined_puzzles_list):
    '''
    runs analysis on all list of puzzles provided.
    solves the puzzles using all three algorithms
    stores the time taken per algorithm per puzzle
    stores the max queue size
    stores the total nodes expanded
    Plots the results in graphs
    '''

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

    time_collection = {}
    queue_collection = {}
    nodes_collection = {}

    for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:
        time_collection[heuristic] = []
        queue_collection[heuristic] = []
        nodes_collection[heuristic] = []


    for puzzle, true_depth in combined_puzzles_list:
        for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:
            print (heuristic, puzzle, true_depth)
            time_before = time.time()
            final_node, max_queue, total_nodes = general_search(puzzle, goal_state, queueing_function,
heuristic, verbose=False)
            time_after = time.time()
            total_time = time_after - time_before
            print (final_node.depth)

            time_collection[heuristic].append((true_depth, total_time))
            queue_collection[heuristic].append((true_depth, max_queue))
            nodes_collection[heuristic].append((true_depth, total_nodes))
        print ()

    plt.figure(1)
    for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:
        temp_arr = np.array(time_collection[heuristic])
        plt.plot(temp_arr[:,0], temp_arr[:,1], color = COLOR_MAP[heuristic], label=heuristic)
    plt.title('Time vs Depth')
    plt.xlabel('depth')
    plt.ylabel('time in seconds')
    plt.grid()
    plt.legend()
    plt.show()

    plt.figure(2)
    for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:
        temp_arr = np.array(nodes_collection[heuristic])
        plt.plot(temp_arr[:,0], temp_arr[:,1], color = COLOR_MAP[heuristic], label=heuristic)

    plt.title('Nodes Expanded vs Depth')
    plt.xlabel('depth')
    plt.ylabel('Nodes Expanded')
    plt.grid()
    plt.legend()
    plt.show()

    plt.figure(3)
    for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:
        temp_arr = np.array(queue_collection[heuristic])
        plt.plot(temp_arr[:,0], temp_arr[:,1], color = COLOR_MAP[heuristic], label=heuristic)

    plt.title('Max Queue Size vs Depth')
    plt.xlabel('depth')
    plt.ylabel('Max Queue Size')
    plt.grid()
    plt.legend()
    plt.show()

# this will take ~2hrs to run
# run_analysis(list_of_easy_puzzles + list_of_medium_puzzles + list_of_hard_puzzles)


##################################################################
########    Generating puzzles at different depths   ########
##################################################################

# code to generate random puzzles and solving them to find the depth of puzzles.
# used to create multiple puzzles to test the algorithms and for benchmarking
# the original method to let the algorithm run till all nodes are explored to get failure takes a lot of time.
# therefore using a constant time algorithm to generate puzzles.
# code is taken from
# https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

def get_int_count(arr):
    '''
    counts the total number of inversions to be made
    '''
    inv_count = 0
    empty_value = 0
    for i in range(0, 9):
        for j in range(i + 1, 9):
            if arr[j] != empty_value and arr[i] != empty_value and arr[i] > arr[j]:
                inv_count += 1
    return inv_count

def is_solvable(puzzle):
    '''
    checks if the 8 puzzle is solvable or not
    '''
    # Count inversions in given 8 puzzle
    inv_count = get_int_count(puzzle)

    # return true if inversion count is even.
    return (inv_count % 2 == 0)

def create_puzzle_book(n_iters):
    '''
    create a puzzle book that stores a list of problem states are various depths
    '''

    puzzle_book = {}
    for _ in range(n_iters):

        # generate a random 8 puzzle
        random_state = generate_random_states(9)

        # check if the puzzle is solvable
        if (is_solvable(random_state)):

            # if solvable, solve the puzzle to get depth
            goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
            final_node, _, _ = general_search(
                random_state, goal_state, queueing_function, MANHATTAN, verbose=False)

            # store the puzzle at appropriate depth entry
            if final_node.depth not in puzzle_book:
                puzzle_book[final_node.depth] = []

            puzzle_book[final_node.depth].append(random_state)

    return puzzle_book

# print(create_puzzle_book(30))

# ################################################################
# ########  Pretty Print Puzzles to display in Report  ########
# ################################################################

def pretty_print_puzzles(combined_puzzles):
    '''
    prints the puzzles in matrix form so they can be added to report
    '''

    for puzzle, true_depth in combined_puzzles:

        print (f'          --- {true_depth} ---')
        print ('         ', puzzle[:3])
        print ('         ', puzzle[3:6])
        print ('         ', puzzle[6:9])
        print ()
        print ()

# pretty_print_puzzles(list_of_easy_puzzles + list_of_medium_puzzles + list_of_hard_puzzles)
# pretty_print_puzzles(list_of_easy_puzzles)
```