CS 205: Artificial Intelligence by Dr. Eamonn Keogh

Project 1: 8 Puzzle problem Report

Name: Mohit Porwal

SID: 862325163

UCR mail id: mporw002@ucr.edu

18th May 2022

In completing this assignment, I referred the following resource:

1. The sample report pdf provided by the professor to make this report
2. For conceptual understanding, I referred to the Blind Search 1, Blind Search 2 and the Heuristic Search ppt slides by the professor.
3. For the layout of the program (UI), I referred the one provided in the sample report.
4. To learn about the heapq module, I referred the following YouTube video https://www.youtube.com/watch?v=4hkJBcW5Ruk
5. To learn about the Matplotlib library, I referred the official Matplotlib documentation https://matplotlib.org/stable/index.html
6. The Uniform Cost Search graph and the examples for tiles were taken from professor's slides

The entire code in this document is original, except for some subroutines that I imported from Python module

a. *heapq* module to implement min heap priority queue
b. *sys* to terminate the program
c. *copy* to deepcopy all the object instances
d. *time* to calculate the total time required for the search
e. *itertools* to convert 2D list to 1D list
f. *matplotlib* to plot the graphs

OUTLINE OF THIS REPORT

Introduction

A sliding puzzle or a sliding tile puzzle is a mechanical puzzle consisting of N-1 blocks of tiles where N is the total number of empty spaces in the puzzle. The standard and the most famous category in this are the 15 Puzzle problem where we have a board with 4X4 dimension with 15 blocks of tiles placed on the board. The puzzle is a 2 dimensional board where we leave the last block empty. A smaller version of this puzzle is the 8-puzzle problem where we have a 3X3 board which has 9 spaces and we 8 of those spaces is occupied by the tiles and the last space is kept empty. The two figures, Figure 1 and Figure 2 shown below, illustrate how the puzzle looks in its standard setting or we can say, the goal state.

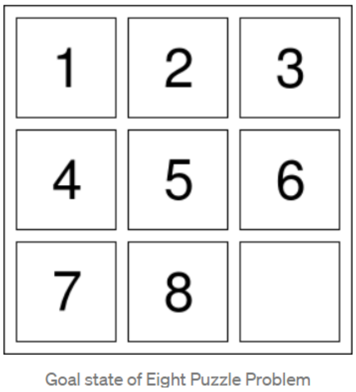


Figure 1: 8 Puzzle Goal State

The area of the board is divided into 4X4 grid where each square of the grid holds a square tile. There are numbers from 1 to N-1 on each tile. For 8 puzzle problem, the tiles are numbered from 1 to 8. There are some rules to solve this puzzle and the aim is to reach the goal state.

In the initial state, the tiles can be arranged in any random order (with one empty space). The aim is to reach the goal state from the initial distorted state. There are four operations allowed on each tile.

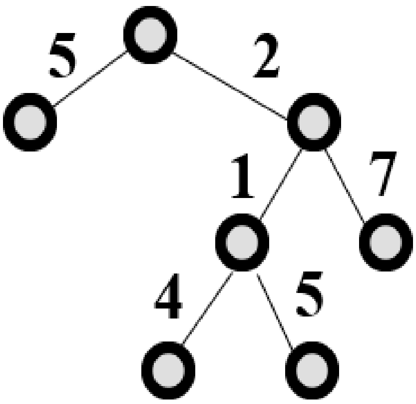
The tile can be slides in four directions which are up, down, right and left. The tile can be moved by only 1 space at a time. Each movement will leave the previous tile with an empty space. In this way, by sliding, we need to reach the goal state.

Figure 2: 15 Puzzle Goal State

The 8-puzzle problem is a part of the course CS205: Artificial Intelligence, by Dr. Eamonn Keogh at University of California, Riverside. Our task in this project is to implement the 8-puzzle problem with 3 algorithms which are, Uniform Cost Search, A* with misplaced tile heuristic and A* with Manhattan distance heuristic. In the upcoming parts of the report, I have compared the performance of these three algorithms and also included my version of the code to implement this using Python programming language. Let us explore each algorithm in brief.

Uniform Cost Search

Uniform Cost Search is a search algorithm which falls in the class of uninformed search algorithms. The algorithm chooses the path or a state with the least cost and then expands that state. So, the expansion of a node happens if it’s the one with the lowest cost when compared to its sibling nodes. We can implement this using a Priority Queue data structure. Figure 3 below shows an illustration of the working of Uniform Cost Search.

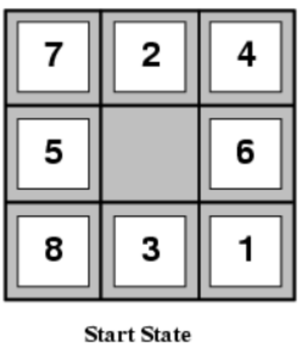


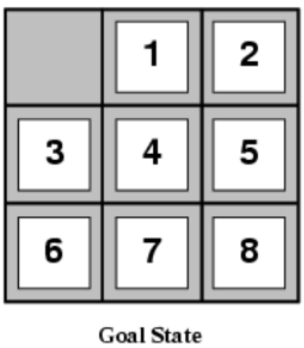
The Uniform Cost Search has no heuristic to assist in its search hence the h(n) value for it will be equal to 0. Therefore, the cost function will be f(n) = g(n), where g(n) is the cost from root node to current node. In this case, it will always be 1, because the cost of each operation in 8 puzzle is just 1. With h(n) as 0, it will act as Breadth-First-Search.

Figure 3: Uniform Cost Search graph

The Misplaced Tile Heuristic

The A* with misplaced tile heuristic calculates the number of misplaced tiles as a heuristic in our algorithm. The expansion of a particular node is decided on the basis of the cost function f(n) = g(n) + h(n) where g(n) is our cost of depth from initial state to current state and the h(n) is the misplaced tile heuristic. The h(n) is calculated by identifying the tiles in the current state which are not in their correct position when compared with the goal state. We sum up the count of all such tiles and that becomes our h(n). We do not consider the blank state when counting the number of misplaced tiles.





In figure 4 and 5 example, the misplaced tiles are 7,4,2,5,6,8,3,1 which is total 8. Note that we are not considering the blank tile in the counting.

The final h(n) cost for this example will become 8.

Fig 4: Misplaced Tile Start **Fig 5: Misplaced Tile Goal**

The Manhattan Distance Heuristic

The A* with Manhattan distance heuristic calculates the cost based on how far the misplaced tiles of the current state are from the goal state. For instance, if there are 3 tiles in the board that are not in their correct places, we can determine the number of steps or the amount of sliding that tile will require in order to reach the goal state. These number of steps are calculated for each misplaced tile and then summed up. The summation will be the final heuristic cost h(n) for that particular state. To illustrate this, we have an example.

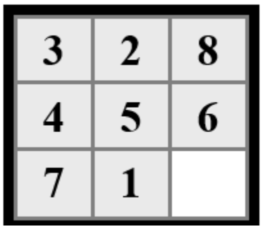


Figure 6:
Manhattan Start State

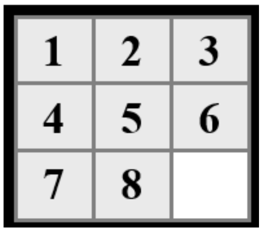


Figure 7:
Manhattan Goal State

If we observe figure 6 which represents the current state and figure 7 which is goal state. The tile 3 is 2 spaces away from its goal state, the tile is 3 spaces away, and the tile 1 is 3 spaces away. Therefore, 3+3+2 = 8, this is our total h(n) for the current state.

Comparison of Algorithms based on Sample Puzzles

Graph color code

Green: Manhattan Distance Algorithm

Orange: Misplaced Tile Algorithm

Blue: Uniform Cost Search Algorithm

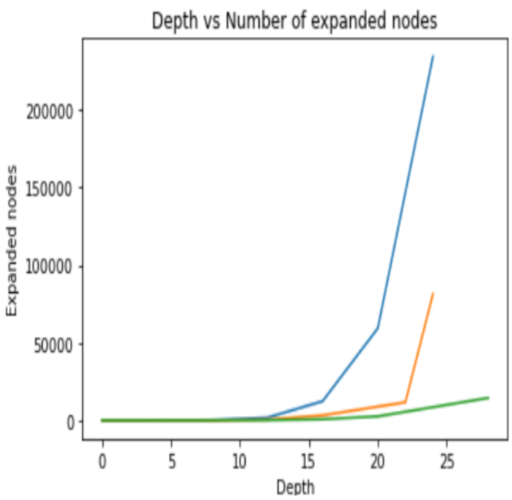
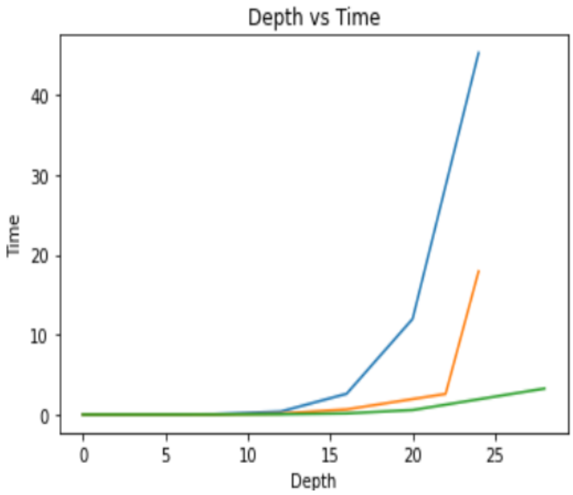


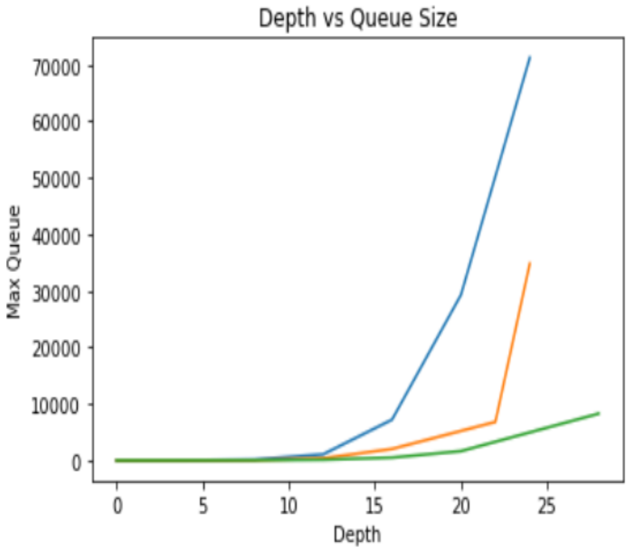
Figure 8: Depth vs Number of nodes expanded

As we can observe, the depth of the search increases with the number of nodes, especially for uniform cost search since it naively expands each and every state, in a breadth first search manner, rather than using some heuristic. The heuristic algorithms Manhattan and Misplaced tile expand far lesser nodes with the increasing depth since they smartly choose the least cost state. From the graph, it can be noted that Uniform Cost Search performs the worst and after depth 20, Manhattan expands far lesser nodes than the misplaced tile.



The depth vs time graph in figure 9, shows that the Uniform Cost Search performs the worst in terms of time as it increases the number of depths to the greatest extent. Misplaced tile's time complexity increases exponentially after node 20 since it has to calculate heuristic cost for a greater number of tiles now which takes time. Similar can be seen and concluded for Manhattan but it seems that the time complexity of Manhattan does not increase exponentially

Figure 9: Depth vs Time



When it comes to the queue size, it increases with depth after node 10 for all the three algorithms. Till node 10, the size of queue remains constant for all three algorithms. Even in this scenario, Manhattan distance performs orders of magnitude better than the other three algorithms.

Figure 10: Depth vs Maximum Queue Size

From the above graphs, it can be concluded that Manhattan Distance gives the best performance most probably because it is more intuitive as compared to the naïve uniform cost and the misplaced tile. The misplaced tile just counts the number of tiles not in their position but the Manhattan distance tells **how far** each tile is from the goal the goal state, thus giving an idea how far the overall state is from the goal state.

CODE (https://github.com/L-E-A-R-N-E-R/8_Puzzle_Problem)

```python
from copy import deepcopy
from itertools import chain
import time
import heapq as heap
import sys
import matplotlib.pyplot as plt

goal8 = [[1,2,3],[4,5,6],[7,8,0]]    #for 8 puzzle
goal15 = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,0]]     #for 15 puzzle
goal24 = [[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15],[16,17,18,19,20],[21,22,23,24,0]]    #for 24 puzzle
mapping = {}                                #to trace the path of parent states
mq_size=0      #maximum size of queue

def main():

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

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

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

    print("8 puzzle solver!\n\n")

    print("The puzzle can be a default and also the one of your choice. Choose 1 for default or choose 2 for your own puzzl

    choice = int(input())

    if(choice==1):

        puzzle = [[1,3,6],[5,0,2],[4,7,8]]
        print("The default puzzle looks something like this\n")
        print(puzzle)

    elif(choice==2):

        print("Enter number of rows\n")
        rows = int(input())
        print("Enter number of columns\n")
        cols = int(input())

        print("Enter the elements one by one. Use zero to represent a blank element in the puzzle\n")

        puzzlein = []
        for i in range(rows):
            inside = []
            for j in range(cols):
                temp = int(input())
                inside.append(temp)
            puzzlein.append(inside)

        print("\n")
        print("The puzzle entered by you looks something like this \n")

        for z1 in range(rows):
            for z2 in range(cols):
                print(puzzlein[z1][z2],end=" ")
            print()

        print("\n")

    print("There are three ways to solve it! Please choose one of the following methods:\n\n")

    print("1. Uniform Cost Search\n2. A* with the Misplaced Tile heuristic\n3. A* with Manhattan Distance heuristic\n")

    method = int(input())

```

```python
    method = int(input())

    if(choice==1):
        expanded_nodes, depth, maximum_size_of_queue, final_time = general_search(puzzle,method)
    else:
        expanded_nodes, depth, maximum_size_of_queue, final_time = general_search(puzzlein,method)

    print("Expanded nodes: " + str(expanded_nodes))
    print("Depth: " + str(depth))
    print("Maximum size of queue:" + str(maximum_size_of_queue))

    print('%.2f' % (final_time))

    print("Choose 1 to plot the graph, 0 to terminate")

    graph = int(input())

    if graph==1:
        plotter(depth_samples,method)
    else:
        sys.exit()

```

```python
def general_search(problem,choice):

    many_nodes = []
    initial_node = (0,0,problem)             #initial node is a tuple where I store the cost, depth and the puzzle state
    heap.heappush(many_nodes,initial_node)
    expanded = 0
    visited_set = set()                      #to keep track of already visited states

    initial_time = time.time()

    global mq_size
    mq_size = 0

    while True:

        if(len(many_nodes)==0):
            print("Failure")
            break

        node = heap.heappop(many_nodes)
        print("The best state to expand with a g(n) = ",node[0],"and h(n) = ",node[1],"is...\n")
        node_content = node[2]
        for insider in range(len(node_content)):
            print(node_content[insider])
        print("\n")
        expanded += 1

        visited_set.add(tuple(map(tuple,node[2])))

        if node[2]==goal8:                   #change goal8 to goal15 for 15 puzzle and goal24 for 24 puzzle
            ending_time = time.time()
            time_diff = ending_time - initial_time
            return expanded,node[1],mq_size,time_diff
```

```python

        many_nodes = queueing_function(many_nodes,expansion(node,choice,visited_set))

def expansion(parent_node,algorithm,seen):

    child_states = []

    cost = parent_node[0]
    depth = parent_node[1]
    matmat = parent_node[2]

    for hail in range(len(matmat)):
        for ucr in range(len(matmat)):
            if(matmat[hail][ucr]==0):

                ele_row = hail
                ele_col = ucr

                if(ele_col>0):
                    left_child = (cost,depth,left(matmat,ele_row,ele_col))
                    temp_tup_left = (tuple(map(tuple,left_child[2])))
                    if (temp_tup_left not in seen):
                        child_states.append(left_child)

                if(ele_col<len(matmat)-1):
                    right_child = (cost,depth,right(matmat,ele_row,ele_col))
                    temp_tup_right = (tuple(map(tuple,right_child[2])))
                    if (temp_tup_right not in seen):
                        child_states.append(right_child)
```

```
            if(ele_row>0):
                up_child = (cost,depth,up(matmat,ele_row,ele_col))
                temp_tup_up = (tuple(map(tuple,up_child[2])))
                if (temp_tup_up not in seen):
                    child_states.append(up_child)

            if(ele_row<len(matmat)-1):
                down_child = (cost,depth,down(matmat,ele_row,ele_col))
                temp_tup_down = (tuple(map(tuple,down_child[2])))
                if (temp_tup_down not in seen):
                    child_states.append(down_child)

    final = []
    for each_child in child_states:

        each_child = list(each_child)
        each_child[1] += 1
        each_child = tuple(each_child)

        if algorithm == 1:
            each_child = list(each_child)
            each_child[0] += 1
            each_child = tuple(each_child)
        if algorithm == 2:
            h_n = misplaced_tiles(each_child[2])
            each_child = list(each_child)
            each_child[0] = each_child[0] + h_n + 1
            each_child = tuple(each_child)
        if algorithm ==3:
            h_n = manhattan(each_child[2])
            each_child = list(each_child)
            each_child[0] = each_child[0] + h_n + 1
```

```
            each_child = list(each_child)
            each_child[0] = each_child[0] + h_n + 1
            each_child = tuple(each_child)

        final.append(each_child)

    return final

def queueing_function(node_list,child_list):
    global mq_size
    for e in child_list:
        heap.heappush(node_list,e)
        if (len(node_list) > mq_size):
            mq_size = len(node_list)
    return node_list

def left(matrix1,p,q):

    xten1 = deepcopy(matrix1)
    tempo1 = xten1[p][q]
    xten1[p][q] = xten1[p][q-1]
    xten1[p][q-1] = tempo1

    fedup1 = tuple((map(tuple,xten1)))

    mapping[fedup1] = tuple((map(tuple,matrix1)))
    return xten1

def right(matrix2,r,s):

    xten2 = deepcopy(matrix2)
    tempo2 = xten2[r][s]
    xten2[r][s] = xten2[r][s+1]
```

```
    xten2[r][s] = xten2[r][s+1]
    xten2[r][s+1] = tempo2

    fedup2 = tuple((map(tuple,xten2)))

    mapping[fedup2] = tuple((map(tuple,matrix2)))
    return xten2

def up(matrix3,t,u):

    xten3 = deepcopy(matrix3)
    tempo3 = xten3[t][u]
    xten3[t][u] = xten3[t-1][u]
    xten3[t-1][u] = tempo3

    fedup3 = tuple((map(tuple,xten3)))

    mapping[fedup3] = tuple((map(tuple,matrix3)))
    return xten3

def down(matrix4,v,w):

    xten4 = deepcopy(matrix4)
    tempo4 = xten4[v][w]
    xten4[v][w] = xten4[v+1][w]
    xten4[v+1][w] = tempo4

    fedup4 = tuple((map(tuple,xten4)))

    mapping[fedup4] = tuple((map(tuple,matrix4)))
    return xten4

```

```
def manhattan(two_d):

    dict = {1:[0,0], 2:[0,1], 3:[0,2], 4:[1,0], 5:[1,1], 6:[1,2], 7:[2,0], 8:[2,1]}  #to create this mapping for puzzle of a
    cost_of_one_state = 0

    for ice in range(len(two_d)):
        for juice in range(len(two_d)):
            if(two_d[ice][juice]!=0):
                r = ice
                c = juice
                linear = dict.get(two_d[ice][juice])
                cost_of_one_state += abs(linear[0]-r) + abs(linear[1]-c)

    return cost_of_one_state

def misplaced_tiles(tile_cost):

    count = 0
    calculation = list(chain.from_iterable(tile_cost))

    for element in calculation:
        if(element!=0):
            position = calculation.index(element)
            if((element-position)!=1):
                count+=1

    return count

def plotter(testers,option):

    exp_node_testers_uniform = []
    depth_testers_uniform = []
    max_queue_testers_uniform = []
    time_testers_uniform = []
```

```
    time_testers_uniform = []

    exp_node_testers_misplaced = []
    depth_testers_misplaced = []
    max_queue_testers_misplaced = []
    time_testers_misplaced = []

    exp_node_testers_manhattan = []
    depth_testers_manhattan = []
    max_queue_testers_manhattan= []
    time_testers_manhattan = []

    for algos in range(1,4):

        for looping in testers:

            exp_node,d,mqsz,t = general_search(looping,algos)

            if algos==1:
                depth_testers_uniform.append(d)
                max_queue_testers_uniform.append(mqsz)
                time_testers_uniform.append(t)
                exp_node_testers_uniform.append(exp_node)

            if algos==2:
                depth_testers_misplaced.append(d)
                max_queue_testers_misplaced.append(mqsz)
                time_testers_misplaced.append(t)
                exp_node_testers_misplaced.append(exp_node)
```

```

            if algos==3:
                depth_testers_manhattan.append(d)
                max_queue_testers_manhattan.append(mqsz)
                time_testers_manhattan.append(t)
                exp_node_testers_manhattan.append(exp_node)

    #Depth and nodes expanded
    plt.plot(depth_testers_uniform,exp_node_testers_uniform)
    plt.plot(depth_testers_misplaced,exp_node_testers_misplaced)
    plt.plot(depth_testers_manhattan,exp_node_testers_manhattan)
    plt.title("Depth vs Number of expanded nodes")
    plt.xlabel("Depth")
    plt.ylabel("Expanded nodes")
    plt.show()

    #Depth and time
    plt.plot(depth_testers_uniform,time_testers_uniform)
    plt.plot(depth_testers_misplaced,time_testers_misplaced)
    plt.plot(depth_testers_manhattan,time_testers_manhattan)
    plt.title("Depth vs Time")
    plt.xlabel("Depth")
    plt.ylabel("Time")
    plt.show()

    #Depth and maximum queue size
    plt.plot(depth_testers_uniform,max_queue_testers_uniform)
    plt.plot(depth_testers_misplaced,max_queue_testers_misplaced)
    plt.plot(depth_testers_manhattan,max_queue_testers_manhattan)
    plt.title("Depth vs Queue Size")
    plt.xlabel("Depth")
    plt.ylabel("Max Queue")
    plt.show()


if __name__ == "__main__":
    main()

```

SAMPLE OUTPUT:

a. Traceback for Depth 8 with misplaced tile heuristic

```
8 puzzle solver!


The puzzle can be a default and also the one of your choice. Choose 1 for default or choose 2 for your own puzzle

1
The default puzzle looks something like this

[[1, 3, 6], [5, 0, 2], [4, 7, 8]]
There are three ways to solve it! Please choose one of the following methods:


1. Uniform Cost Search
2. A* with the Misplaced Tile heuristic
3. A* with Manhattan Distance heuristic

3
The best state to expand with a g(n) =  0 and h(n) =  0 is...

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


The best state to expand with a g(n) =  8 and h(n) =  1 is...

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


The best state to expand with a g(n) =  8 and h(n) =  1 is...
```

```
The best state to expand with a g(n) =  8 and h(n) =  1 is...

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


The best state to expand with a g(n) =  10 and h(n) =  1 is...

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


The best state to expand with a g(n) =  10 and h(n) =  1 is...

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


The best state to expand with a g(n) =  15 and h(n) =  2 is...

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

```
The best state to expand with a g(n) =  15 and h(n) =  2 is...

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


The best state to expand with a g(n) =  17 and h(n) =  2 is...

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


The best state to expand with a g(n) =  17 and h(n) =  2 is...

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


The best state to expand with a g(n) =  19 and h(n) =  2 is...

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


The best state to expand with a g(n) =  21 and h(n) =  2 is...

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

```
The best state to expand with a g(n) =  21 and h(n) =  2 is...

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


The best state to expand with a g(n) =  21 and h(n) =  2 is...

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


The best state to expand with a g(n) =  21 and h(n) =  3 is...

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


The best state to expand with a g(n) =  21 and h(n) =  3 is...

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

```
The best state to expand with a g(n) =  26 and h(n) =  4 is...

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


The best state to expand with a g(n) =  26 and h(n) =  4 is...

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


The best state to expand with a g(n) =  27 and h(n) =  3 is...

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


The best state to expand with a g(n) =  27 and h(n) =  3 is...

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


The best state to expand with a g(n) =  28 and h(n) =  4 is...

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

```
The best state to expand with a g(n) =  28 and h(n) =  4 is...

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


The best state to expand with a g(n) =  29 and h(n) =  3 is...

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


The best state to expand with a g(n) =  30 and h(n) =  5 is...

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


The best state to expand with a g(n) =  31 and h(n) =  3 is...

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

```
The best state to expand with a g(n) =  36 and h(n) =  5 is...

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


The best state to expand with a g(n) =  36 and h(n) =  5 is...

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


The best state to expand with a g(n) =  36 and h(n) =  8 is...

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


Expanded nodes: 37
Depth: 8
Maximum size of queue:29
0.02
Choose 1 to plot the graph, 0 to terminate
```

b. Traceback for Depth 4 with Manhattan Distance

```
8 puzzle solver!

The puzzle can be a default and also the one of your choice. Choose 1 for default or choose 2 for your own puzzle

2
Enter number of rows

3
Enter number of columns

3
Enter the elements one by one. Use zero to represent a blank element in the puzzle

1
2
3
5
0
6
4
7
8


The puzzle entered by you looks something like this

1 2 3
5 0 6
4 7 8
```

```
There are three ways to solve it! Please choose one of the following methods:


1. Uniform Cost Search
2. A* with the Misplaced Tile heuristic
3. A* with Manhattan Distance heuristic

3
The best state to expand with a g(n) =  0 and h(n) =  0 is...

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


The best state to expand with a g(n) =  4 and h(n) =  1 is...

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


The best state to expand with a g(n) =  6 and h(n) =  1 is...

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

```
The best state to expand with a g(n) =  6 and h(n) =  1 is...

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


The best state to expand with a g(n) =  6 and h(n) =  1 is...

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


The best state to expand with a g(n) =  7 and h(n) =  2 is...

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


The best state to expand with a g(n) =  9 and h(n) =  2 is...

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

```
The best state to expand with a g(n) =  9 and h(n) =  3 is...

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


The best state to expand with a g(n) =  10 and h(n) =  4 is...

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


Expanded nodes: 9
Depth: 4
Maximum size of queue:9
0.01
Choose 1 to plot the graph, 0 to terminate
```