Project 1
CS 205 Artificial Intelligence, Dr. Eamonn Keogh

Shubham Sharma
SID: 862253820
Email: sshar150@ucr.edu
Date: 12th May, 2022

In completion of this project I referred the following sources:

- Lecture slides and annotations for understanding the concept of Uniform cost search, A* with the Misplaced Tile heuristic and A* with the Manhattan Distance heuristic.
- Python documentation for using various inbuilt library methods https://docs.python.org/3.6/contents.html
- The sample report provided by professor for the project.
- To get a better understanding about 8-puzzle problem and getting started with the implementation I also read online articles. https://faramira.com/solving-8-puzzle-problem-using-a-star-search/

The implementation code is original except for the use of inbuilt functions and code used to manipulate the queue.

Outline of the report:

CS205 Artifical Intelligence - 8 puzzle problem
Project Report

Introduction:

As a part of coursework for CS205 Prof. Eamonn Keogh gave us a project to implement algorithm for solving 8 puzzle problem. 8 puzzle problem or commonly known as 3x3 sliding tile puzzle is formally defined by Edward Hordern as “A combination puzzle that challenges a player to slide pieces along certain routes to establish a certain end-configuration” [1]. User can move tiles in 4 directions up, down, left and right based on the position of the tile. Out of 9 positions, 1 position is empty and a user can move other tiles into that position to get the desired goal state.

As instructed in the project document provided by the professor, I implemented uniform cost search without any heuristic and A* algorithm with manhattan distance and displacement tile heuristic to solve the 8 puzzle problem. The report contains detailed analysis of my findings for solving the problem using given 3 approaches. The main task of the problem is to slide the tiles and get to the goal state. The initial state can be defined by user or could be a default state specified in the program and the goal state is the desired state in the end. The figure 1 demonstrates an example of 8 puzzle problem with initial and goal state. The same goal state is used in my algorithm as well. I used python3 for the implementation of my algorithm and the code is attached at the last section of this report.

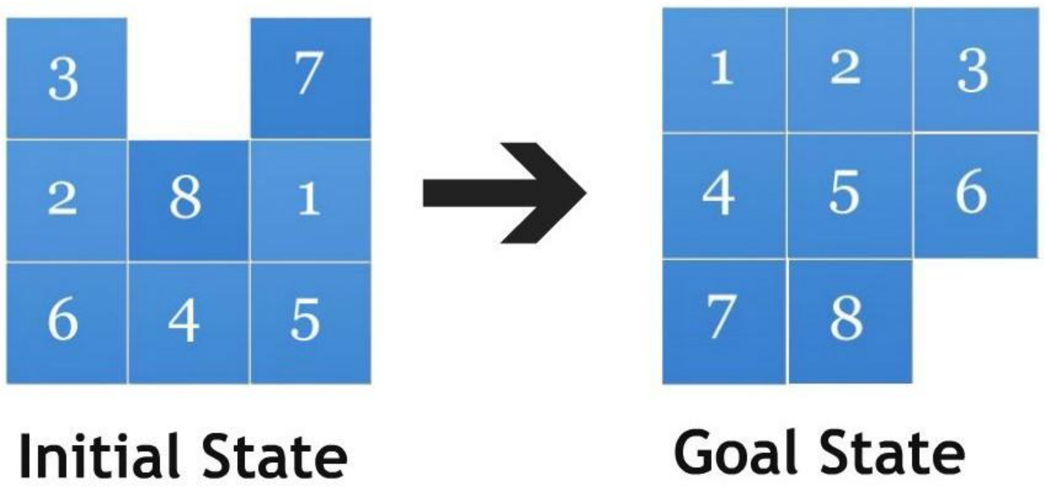


Figure 1: A picture of 8 puzzle problem with initial and goal state.[2]

Description of Algorithms

All the 3 search algorithms have a common structure and the decision they make to choose which node to expand depends on the cost function f(n) which can be defined as:

f(n) = h(n) + g(n)

Where g(n) is the depth cost and h(n) is the heuristic cost.

1. **Uniform Cost Search:**

   "Uniform-cost search is an uninformed search algorithm that uses the lowest cumulative cost to find a path from the source to the destination. Nodes are expanded, starting from the root, according to the minimum cumulative cost" [3]. For 8 puzzle problem, we implement uniform cost search with heuristic value of 0 since it's an uninformed search, it doesnot consider any metric before moving to next state. Since h(n) = 0, it only considers the cost of g(n) which is simply the depth of the node in the tree. Uniform Cost Search is the same as a breadth first search(BFS) hence it is both complete and optimal. This serves as a good baseline to compare the two heuristics version of A* against it.

2. **A* with Misplaced Tiles Heuristic:**

   For evaluating which node to expand first, A* considers both the depth cost g(n) as well as the heuristic cost h(n). In the version of A* With misplaced tiles heuristic the h(n) is given by the number of tiles that do not match with the goal state except the blank tile (value 0 in our program). Consider the following example:

   Input State:

1	2	3
0	4	5
7	6	8

   The total number of displaced tiles = 4. Hence, h(n) = 4.

   Since this gives us a better estimation instead of just blinding expanding any node, it converges faster than uniform cost search.

3. A* with Manhattan Distance Heuristic:

Manhattan Distance is much similar to misplaced tile heuristic but instead of just calculating the number of misplaced tiles, we use the sum of the distances of each tile from its goal position as the heuristic value and we expand the node which has least value. Consdier a following example:

Input State:

4	2	7
5	0	6
8	3	1

The manhattan distance of this state is = 4 + 0 + 3 + 1 + 1 + 0 + 4 + 1 = 14

This problem has a solution depth of 24 and it can be noted that this is a better heuristic than Misplaced Tiles as it would just give us a value of h(n) = 6. This makes the Manhattan Distance work better than Misplaced Tile heuristic in most cases.

Implementation

For writing the code python3 language is used. All the three algorithms follow the same structure in the sense only the value of h(n) needs to be calculated separately and passed along to get the node and perform further execution.

To further improve the efficiency of my algorithm I used priority queue (heapq in python) instead of a generic list. I stored the nodes in increasing order of their f(n) value. So everytime I perform a pop operation, I get the nodes with lowest f(n) and they are expanded in the course of the algorithm. To make the algorithm generic for N puzzle, I tried to find the correlation between movement of tiles and its affect on the state of the problem. I found that for 3x3 tile sliding puzzle, if we move a blank space in any direction, its index value is getting changed by a value of 3. So for 15 puzzle problem we can easily change the index by 4 instead of 3 to get the solution. Furthermore, its also helpful in checking if the move is valid or not by checking if the index is getting out of bound on performing that operation. Hence, with just a minimal change in the code, it could be made generic for N puzzle. For comparing the algorithms, I executed the sample states provided by professor in the project document and I calculated the time and space (maximum size of queue at any time) each algorithm takes to find the solution at different depths.

Comparison of algorithms based on sample examples

The examples provided by the professor in the project handout are used here on all the three algorithms and the below contains the information about Nodes Expanded, Max queue size and execution time with respect to the depth of solution.

Solution Depth	Nodes Expanded	Max queue size	Execution time (sec)
2	7	8	0.003
4	28	22	0.006
8	264	169	0.029
16	12378	7,208	0.764
24	235573	71268	16.061

Table 1: Solution depth vs Nodes Expanded vs Max queue size vs Execution time for Uniform Cost Search

Solution Depth	Nodes Expanded	Max Queue Size	Execution Time(sec)
2	5	4	0.005
4	33	22	0.012
8	256	169	0.057
16	5717	3386	0.871
24	192449	47205	26.971

Table 3: Solution depth vs Nodes Expanded vs Max queue size vs Execution time for A* with misplaced tiles heuristic.

Solution Depth	Nodes Expanded	Max Queue Size	Execution Time(sec)
2	3	3	0.005
4	16	13	0.01
8	127	87	0.038
16	3,658	2,196	0.548
24	139283	45190	20.587

Table 2: Solution depth vs Nodes Expanded vs Max queue vs Execution time size for A* with Manhattan distance heuristic.

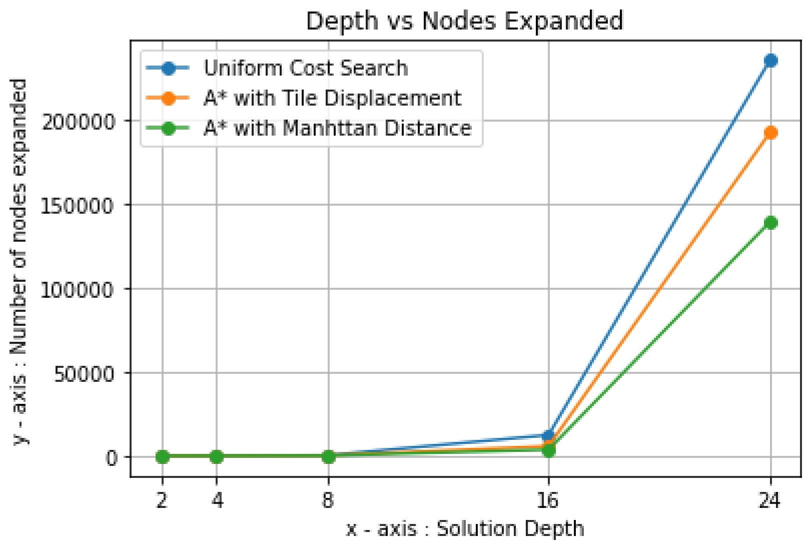


Figure 2 : The relationship between solution depth and nodes expanded to find the solution

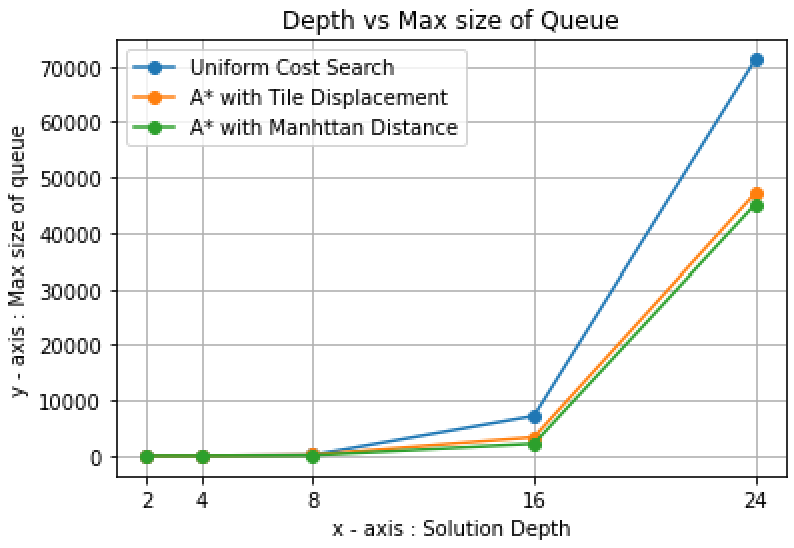


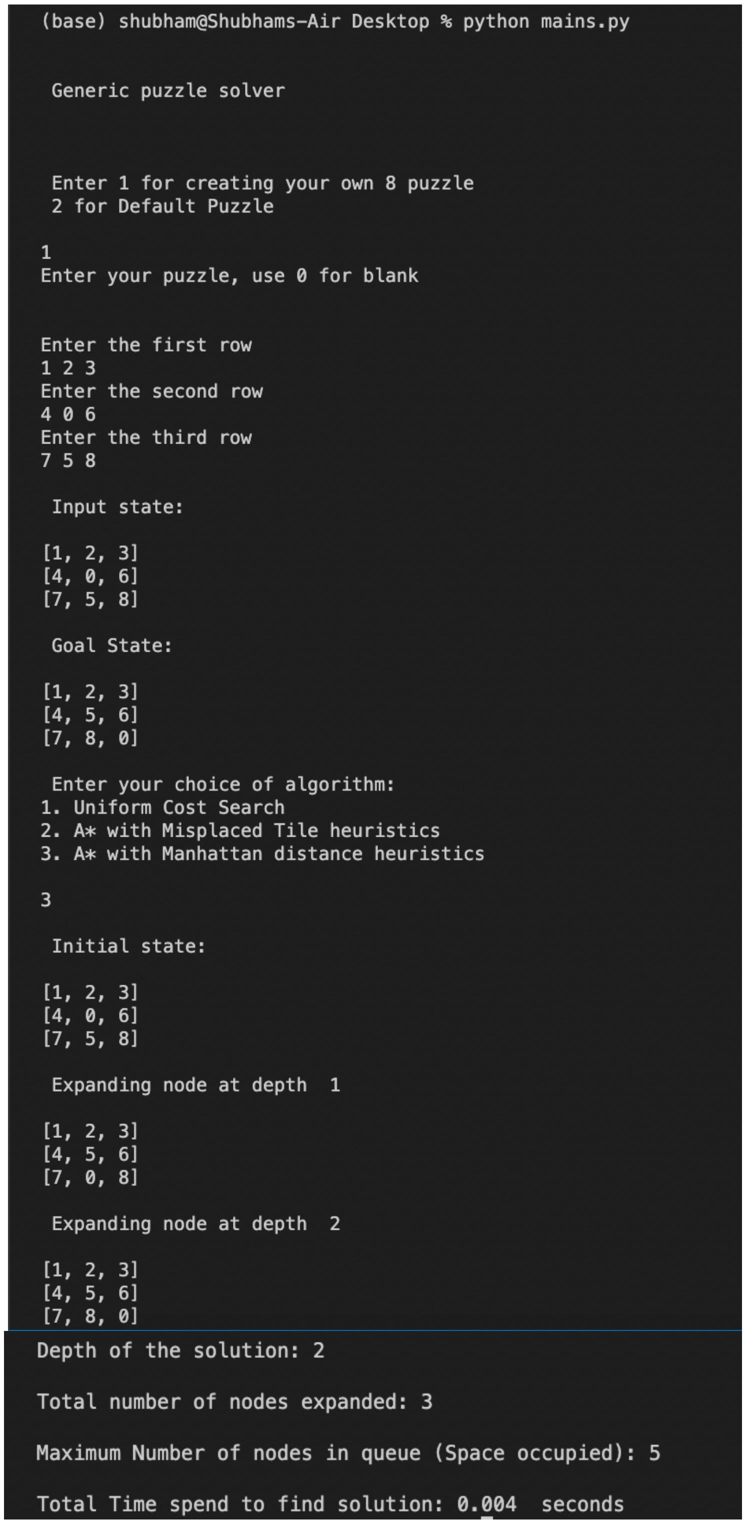
Figure 3 : The relationship between solution depth and Max queue size at any time

Conclusion

Based on the experiments and graphs plotted above it is clear that A* algorithm with manhattan distance heuristic performs the best for 8 puzzle problem followed by A* with displaced tile heuristic and the Uniform Cost Search has the worst performance with the increase of difficulty of puzzle. As the depth of solution increases the heuristic costs start playing an important role in converging to the solution. Sometimes we can get lucky with Tile Displacement if the arrangement of tiles are favourable to get the desired solution. However, manhattan distance certainly provides a better heuristic and coverges faster in most cases. The execution time is more because everytime we need to calculate the heuristic value for other 2 algorithms. But the point that plays an important role is the Nodes expanded at each depth and the maximum size of queue at any given time for getting the solution.

After implementing this project on my own I got a better understanding on building games where user plays against a computer. I used to play computer games and wonder how is it possible to train a computer to know all the moves and toggle its difficulty mode to challenge against normal humans. This project helped me to get a better insight on that and I can better understand now what the term artificial intelligence actually represents and the process through which we make machines “artificially intelligent”.

Execution Output of Code



```
(base) shubham@Shubhams-Air Desktop % python mains.py


 Generic puzzle solver



 Enter 1 for creating your own 8 puzzle
 2 for Default Puzzle

1
Enter your puzzle, use 0 for blank


Enter the first row
1 2 3
Enter the second row
4 0 6
Enter the third row
7 5 8

 Input state:

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

 Goal State:

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

 Enter your choice of algorithm:
1. Uniform Cost Search
2. A* with Misplaced Tile heuristics
3. A* with Manhattan distance heuristics

3

 Initial state:

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

 Expanding node at depth  1

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

 Expanding node at depth  2

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Depth of the solution: 2

Total number of nodes expanded: 3

Maximum Number of nodes in queue (Space occupied): 5

Total Time spend to find solution: 0.004  seconds
```

Code

```
#!/usr/bin/python
# Shubham Sharma
# CS205 Project 1 (8-Puzzle Problem)

import time
import math
import numpy as np
from heapq import heappush, heappop

#class to define the node and some helper helper functions
class Node():
   def __init__(self, state, parent, cost, n):
       self.state = state
       self.parent = parent
       self.cost = cost
       self.n = n
       self._left = None
       self._right = None
       self._up = None
       self._down = None

#These 4 functions check if the move is possible or not in their direction and returns
the new state after making the movement
   def left(self):
       zero_index = np.where(self.state == 0)
       if zero_index[0][0] % self.n == 0:
           return False
       else:
           temp = self.state[zero_index[0][0] - 1]
           new_state = self.state.copy()
           new_state[zero_index] = temp
           new_state[zero_index[0][0] - 1] = 0
           return new_state, temp

   def right(self):
        zero_index = np.where(self.state == 0)
        if zero_index[0][0] % self.n >= self.n - 1:
            return False
        else:
            temp = self.state[zero_index[0][0] + 1]
            new_state = self.state.copy()
            new_state[zero_index] = temp
            new_state[zero_index[0][0] + 1] = 0
            return new_state, temp

    def up(self):
        zero_index = np.where(self.state == 0)
        if zero_index[0][0] <= self.n - 1:
            return False
        else:
            temp = self.state[zero_index[0][0] - self.n]
            new_state = self.state.copy()
            new_state[zero_index] = temp
            new_state[zero_index[0][0] - self.n] = 0
            return new_state, temp

    def down(self):
        zero_index = np.where(self.state == 0)
        if zero_index[0][0] >= self.n * 2:
            return False
        else:
            temp = self.state[zero_index[0][0] + self.n]
            new_state = self.state.copy()
            new_state[zero_index] = temp
            new_state[zero_index[0][0] + self.n] = 0
            return new_state, temp

    def __lt__(self, other):
        return self.state[0] < other.state[0]

    def print_path(self):
        state_trace = [self.state]
        # Adds the node information as going back up the tree.
        while self.parent:
            self = self.parent
            state_trace.append(self.state)

        # Prints the complete path with depth of the solution.
        depth = 0
        state_trace.pop()
        while state_trace:
            state_to_print = state_trace.pop()
            level = depth +1
            print(" \n Expanding node at depth " , level,"\n")
            array = [[state_to_print[j * self.n + i] for i in range(self.n)] for j in
range(self.n)]
            for row in array:
                print(row)
            depth += 1
        print("\n Depth of the solution:",depth)

    def solve(self, goalState, res):
        start = time.time()
        n_value = self.n
        priority_queue = []  # Priority queue to store unvisited nodes wrt to path cost
        heappush(priority_queue, (0, self))
        nodes_processed = 0  # Counter for the number of nodes popped from the queue, to
measure the performance of time
        max_size = 1  # maximum number of nodes in queue as an upper limit on the space
used
        visited_states = set([])  # to remember which states have been visited
        while priority_queue:

            # updating maximum size of the queue
            if len(priority_queue) > max_size:
                max_size = len(priority_queue)

            tc, currentNode = heappop(priority_queue)  # This will give us the node with
least cost.
            nodes_processed += 1
            visited_states.add(tuple(currentNode.state))  # avoid repeated states
            # when the goal state is found, trace back to the root node and print out the
path
            if np.array_equal(currentNode.state, goalState):
                currentNode.print_path()
                print('\n Total number of nodes expanded:', str(nodes_processed))
                print('\n Maximum Number of nodes in queue (Space occupied):',
str(max_size))
                end = time.time()
                time_elapsed = round(end - start, 3)
                print('\n Total Time spend to find solution:', time_elapsed,' seconds')
                return True

            else:
                # check if right move is valid
                if currentNode.right():
                    new_state, x = currentNode.right()
                    # check if the resulting node is already visited
                    if tuple(new_state) not in visited_states:
                        # create a new child node
                        if res == 1:
                            h_n = 0
                        elif res == 2:
                            h_n = tile_displacement(new_state, goalState)
                        else:
                            h_n = manhattan_distance(new_state, goalState)
                        currentNode._right = Node(state=new_state, parent=currentNode,
                                                  cost=tc + h_n + 1, n=n_value)
                        heappush(priority_queue, (tc + h_n+1, currentNode._right))

                # check if left move is valid
                if currentNode.left():
                    new_state, x = currentNode.left()
                    # check if the resulting node is already visited
                    if tuple(new_state) not in visited_states:
                        if res == 1:
                            h_n = 0
                        elif res == 2:
```

```
                    h_n = tile_displacement(new_state, goalState)
                else:
                    h_n = manhattan_distance(new_state, goalState)
                # create a new child node
                currentNode._left = Node(state=new_state, parent=currentNode,
                                         cost=tc + h_n + 1, n=n_value)
                heappush(priority_queue, (tc + h_n+1, currentNode._left))

        # check if down move is valid
        if currentNode.down():
            new_state, x = currentNode.down()
            # check if the resulting node is already visited
            if tuple(new_state) not in visited_states:
                if res == 1:
                    h_n = 0
                elif res == 3:
                    h_n = tile_displacement(new_state, goalState)
                else:
                    h_n = manhattan_distance(new_state, goalState)
                # create a new child node
                currentNode._down = Node(state=new_state, parent=currentNode,
                                         cost=tc + h_n + 1, n=n_value)
                heappush(priority_queue, (tc + h_n+1, currentNode._down))

        # check if up move is valid
        if currentNode.up():
            new_state, x = currentNode.up()
            # check if the resulting node is already visited
            if tuple(new_state) not in visited_states:
                if res == 1:
                    h_n = 0
                elif res == 3:
                    h_n = tile_displacement(new_state, goalState)
                else:
                    h_n = manhattan_distance(new_state, goalState)
                # create a new child node
                currentNode._up = Node(state=new_state, parent=currentNode,
                                       cost=tc + h_n + 1, n=n_value)
```

```
                        heappush(priority_queue, (tc + h_n+1, currentNode._up))

# returns h(n): count of total displaced tiles to reach the goal state
def tile_displacement(new_state, goal_state):
   h_n = np.sum(
       new_state != goal_state) - 1  # Remove 1 for the blank tile.
   if h_n > 0:
       return h_n
   else:
       return 0  # If its the goal state means all are at correct position


# returns h(n): sum of Manhattan distance to reach the goal state
def manhattan_distance(new_state, goal_state):
   distance = 0
   n = len(new_state)
   c = int(math.sqrt(n))
   for i in range(1, n + 1):
       first = np.where(new_state == 0)
       second = np.where(goal_state == 0)
       fx = first[0][0] % c
       fy = first[0][0] / c
       sx = second[0][0] % c
       sy = second[0][0] / c
       distance += abs(fx - sx) + abs(fy - sy)
   return distance

def Execute(input_state, goalState):
    print("\n Enter your choice of algorithm: \n"
    "1. Uniform Cost Search \n2. A* with Misplaced Tile heuristics \n"
    "3. A* with Manhattan distance heuristics \n ")
    res = int(input())

    if res == 1:
         print("\n Initial State: \n ")
         array = [[input_state[j * 3 + i] for i in range(3)] for j in range(3)]
         for row in array:
              print(row)
```

```
        root_node = Node(state=input_state, parent=None, cost=0, n=3)
        root_node.solve(goalState, 1)
    elif res == 2:
        print("\n Initial state: \n ")
        array = [[input_state[j * 3 + i] for i in range(3)] for j in range(3)]
        for row in array:
            print(row)
        root_node = Node(state=input_state, parent=None, cost=0, n=3)
        root_node.solve(goalState, 2)
    else:
        print("\n Initial state: \n ")
        array = [[input_state[j * 3 + i] for i in range(3)] for j in range(3)]
        for row in array:
            print(row)
        root_node = Node(state=input_state, parent=None, cost=0, n=3)
        root_node.solve(goalState, 3)

def print_state(state):
    array = [[state[j * 3 + i] for i in range(3)] for j in range(3)]
    for row in array:
        print(row)

def printMenu():

    print("\n \n Generic puzzle solver \n \n ")

    option = input("\n Enter 1 for creating your own 8 puzzle \n 2 for Default Puzzle
\n")

    if option == "1":
        print("Enter your puzzle, use 0 for blank \n \n")
        first_row = input("Enter the first row \n")
        input_row_1 = [int(x) for x in first_row.split()]
        second_row = input("Enter the second row\n")
        input_row_2 = [int(x) for x in second_row.split()]
        third_row = input("Enter the third row\n")
        input_row_3 = [int(x) for x in third_row.split()]
        input_state = input_row_1 + input_row_2 + input_row_3
        input_state = np.array(input_state)
        print("\n Input state: \n ")
        print_state(input_state)
        goal_State = np.array([1, 2, 3, 4, 5, 6, 7, 8, 0])
        print("\n Goal State: \n")
        print_state(goal_State)
        Execute(input_state, goal_State)

    else:
        input_state = np.array([1,2,3,4,5,6,0,7,8])
        print("\n Input state: \n ")
        print_state(input_state)
        goal_State = np.array([1, 2, 3, 4, 5, 6, 7, 8, 0])
        print("\n Goal State: \n")
        print_state(goal_State)
        Execute(input_state, goal_State)

printMenu()
```

[Link to the source code]()

References

[1] Sliding Piece Puzzles (by Edward Hordern, 1986, Oxford University Press, ISBN 0-19-853204-0)
[2] https://medium.com/swlh/looking-into-k-puzzle-heuristics-6189318eaca2
[3] https://www.educative.io/edpresso/what-is-uniform-cost-search