Project:1, CS205: Artificial Intelligence, Spring 2023

Shahriar M Sakib
SID 862393922
Email ssaki004@ucr.edu
Date May-15-2023

In completing this assignment, I referred to the following sources:

- ✓ The Blind Search and Heuristic Search lecture slides and notes provided in the course.
- ✓ I sought assistance from Stack Overflow for specific coding challenges.
- ✓ I utilized the official Python documentation available at https://docs.python.org/3/whatsnew/3.8.html.
- ✓ To gain a deeper understanding of the problem, I studied the explanation provided at https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/.

All the critical sections of code are original and developed independently. However, certain non-essential subroutines rely on existing libraries and packages. Specifically:

- ✓ The heapq library is used to manage the node structure of states efficiently.
- ✓ The numpy library is utilized to handle instances of different states in the 8-puzzle, enabling tasks such as deep copying and correct state modifications.

Report Structure:

Introduction:

A sliding puzzle, also known as a sliding block puzzle, is a type of puzzle where the player must slide blocks or tiles within a confined space to achieve a specific configuration or solve a given problem. The challenge in sliding puzzles can involve achieving a match in either the shape, configuration, or number arrangement of the puzzle. Below are two examples of sliding puzzles to provide a visual understanding of the concept.

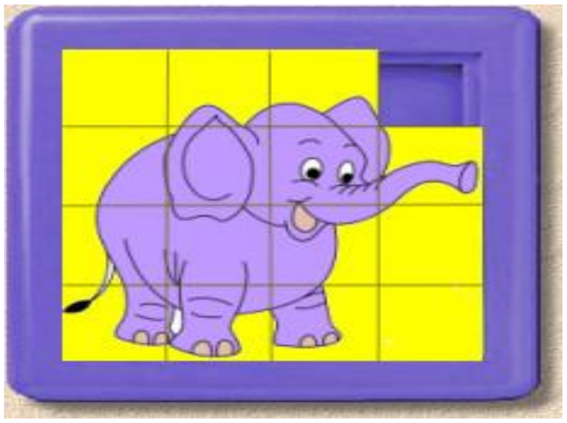


Figure 1: A 15-piece shape matching puzzle.

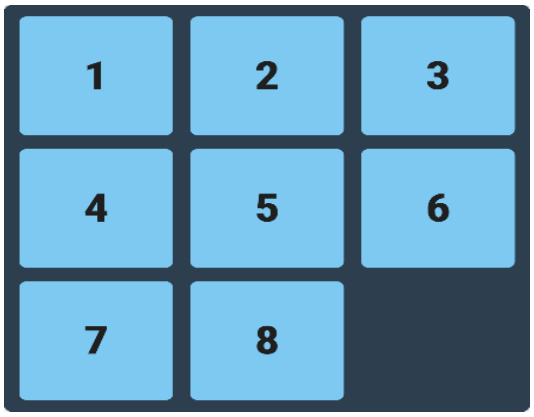


Figure 2: A 8-piece solved sliding puzzle.

The 8-puzzle is a specific variant of the sliding puzzle genre, played on a 3-by-3 grid with numbered blocks and a blank space. The objective of the puzzle is to rearrange the blocks by sliding them into the blank square until they are in the correct order. The following illustrates the step-by-step progression of legal moves from the initial board position on the left to the goal position on the right.

0	1	3
4	2	5
7	8	6

Fig3(a): Initial position

1	0	3
4	2	5
7	8	6

Fig 3(b): Move Right.

1	2	3
4	0	5
7	8	6

Fig 3(c): Move Down.

1	2	3
4	5	0
7	8	6

Fig 3(d): Move Right

1	2	3
4	5	6
7	8	0

Fig3(b): Goal position

In this assignment, our objective is to implement a solver for the 8-puzzle problem. We will utilize the A* search algorithm and evaluate its performance using three different heuristics: uniform cost search, misplaced tile, and Manhattan distance. If the heuristics are admissible, A* is guaranteed to find the optimal solution. In the upcoming sections, we will delve into three different methods in detail and analyze their performances.

Exploration of Heuristics:

When comparing search algorithms, it is important to note that they share similar characteristics, with the main difference lying in their queueing mechanisms. In the context of our project, we will explore three distinct heuristics that assess states using different criteria, leading to variations in the queue order. In A* search, each state is assigned a value, f(n) = g(n) + h(n), where g(n) represents the current depth from the initial state (based on the past), and h(n) estimates the future cost to reach the goal state. While g(n) remains constant for all states in our assignment, h(n) varies depending on the chosen heuristic, thus resulting in different f(n) values. In each iteration, we expand the next node from the frontier set based on the minimum f(n) value. In the following subsections, we will outline how the different heuristics evaluate problem states.

Uniform Cost Search:

Uniform Cost Search is a complete and optimal search algorithm that considers the cost to reach each node from the initial state. It assigns a uniform cost to each node based on the path taken to reach it. However, it can be computationally slow when the cost of reaching nodes varies significantly. In the case of A* search with h(n)=0, it degenerates to Breadth First Search.

The Misplaced Tiles Heuristic:

The Misplaced Tiles Heuristic compares the current state with the goal state to count the number of tiles out of position. It assigns a number to node expansion based on the count of misplaced tiles. This heuristic guides the search algorithm to expand nodes with the lowest cost, making it more efficient than Uniform Cost Search. In Figure 4, the red-colored tiles represent the misplaced tiles in the input state, resulting in a value of g(n) = 4.

2	1	3
4	5	6
8	7	0

Input state

1	2	3
4	5	6
7	8	0

Goal State

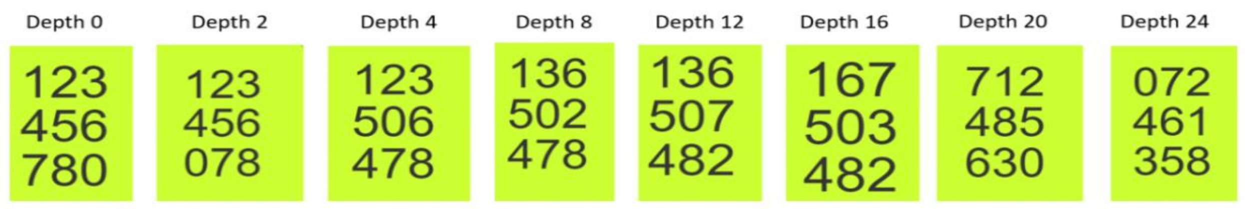
Figure 4: Comparison between given state and goal State.

The Manhattan Distance Heuristic:

The Manhattan Distance Heuristic calculates the total distance that each tile in the current state is away from its desired position in the goal state. It sums up the horizontal and vertical distances for each tile and provides an estimate of the minimum number of moves required to reach the goal state. The Manhattan Distance Heuristic is admissible and provides a more informed search compared to the Misplaced Tiles Heuristic. In Figure 4, consider the input state. For the tile '7', it is located in row 3 and column 2, but in the goal state, it should be in row 3 and column 1. Therefore, the Manhattan distance for '7' is calculated as 0 + 1 = 1. Similarly, the Manhattan distance for tiles 2, 8, and 1 is also 1. Adding up these individual Manhattan distances, the total Manhattan distance from the goal state is 1 + 1 + 1 + 1 = 4. Hence, the value of h(n), which represents the Manhattan distance, will be 4, is the sum of all individual Manhattan distance.

Performance Analysis of Heuristics on Sample Puzzles:

When comparing different heuristics, it is important to consider their admissibility, which ensures they never overestimate the cost and guarantee an optimal solution. Additionally, among admissible heuristics, the one with a higher estimated cost is expected to outperform the others in terms of time and space efficiency. In my evaluation, we utilize a set of problem instances with varying difficulty, as provided by Prof. Eamonn Keogh. These instances, depicted in Figure 5, allow us to analyze the performance of the three heuristics. The depth of each instance represents the optimal number of moves required for a solution.



Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16	Depth 20	Depth 24
123	123	123	136	136	167	712	072
456	456	506	502	507	503	485	461
780	078	478	478	482	482	630	358

Figure 5: Sample instances of problem used for analysis as provided by Prof. Eamonn Keogh

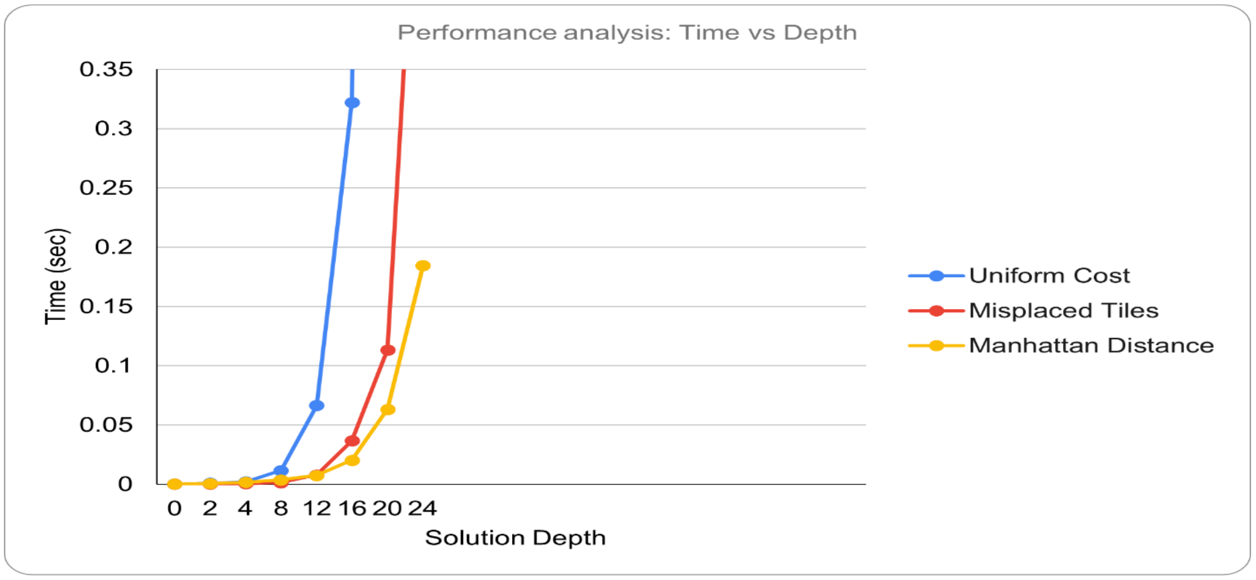


Figure 6: Plot showing time (sec) required for each heuristic in different solution depts.

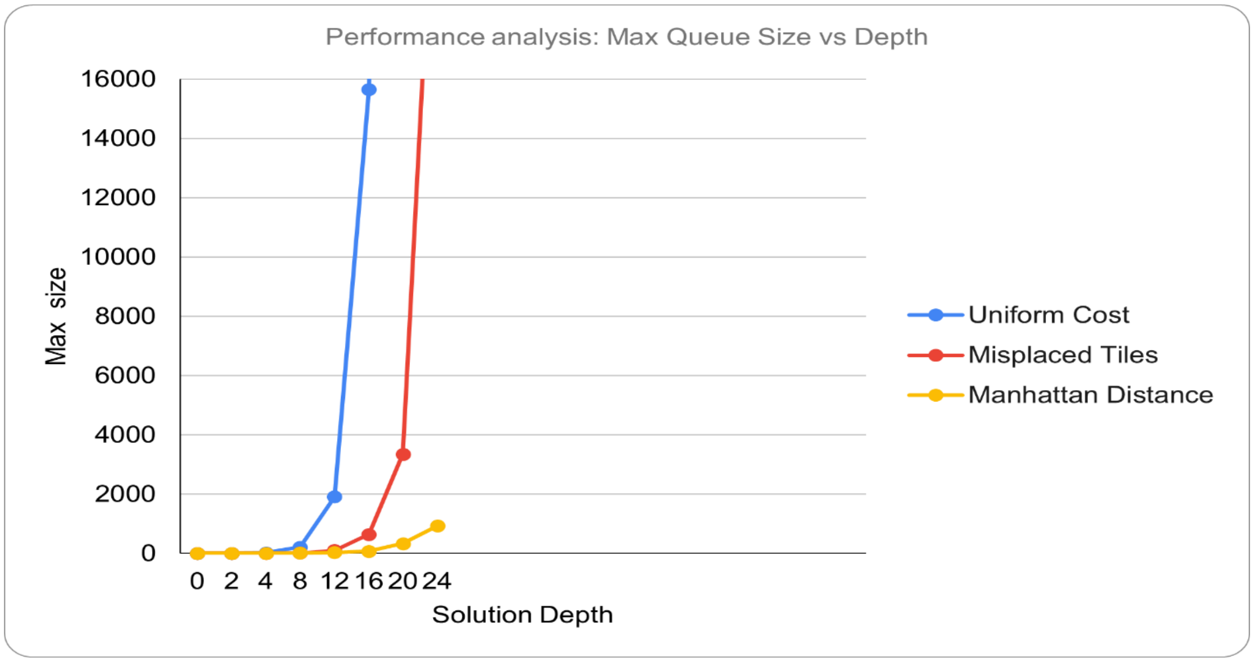


Figure 7: Plot showing Maximum queue size used for each heuristic in different solution depts.

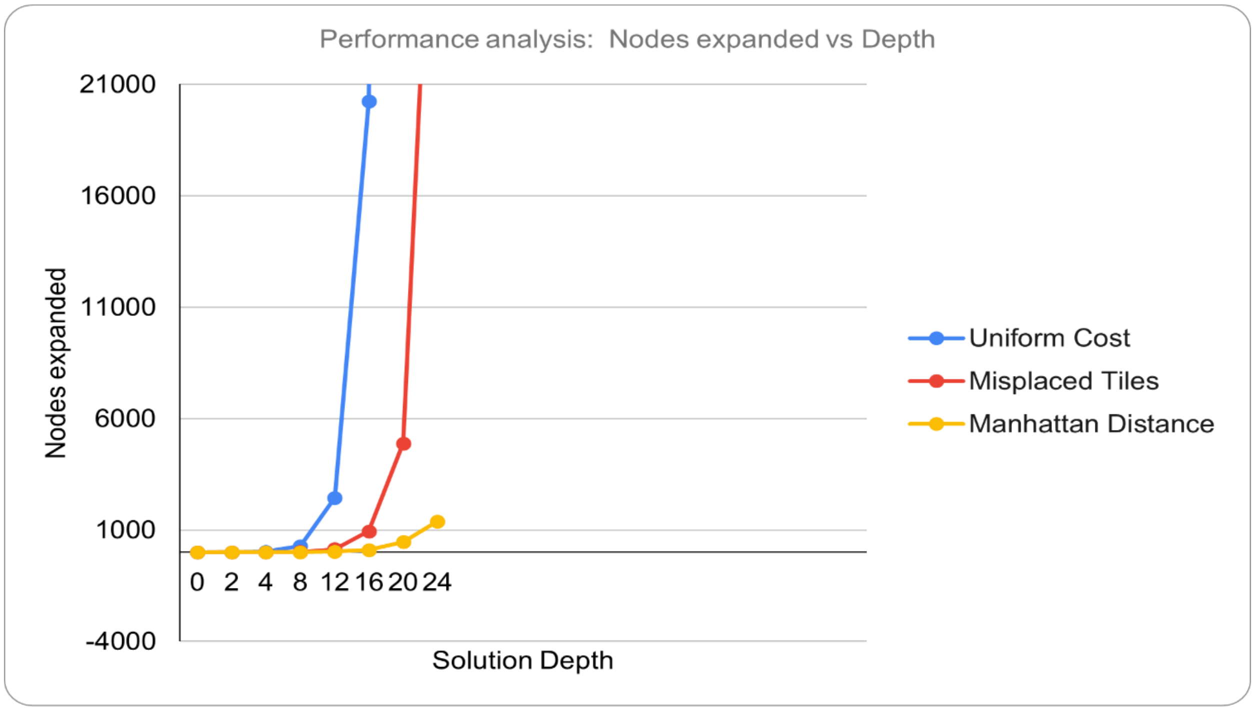


Figure 8: Plot showing nodes expanded for each heuristic in different solution depts.

The performance of the three algorithms was found to be relatively similar for easier puzzles with small depths. However, as the depth increases, or the scenario becomes more complicated, significant differences in time and space complexity can be observed. Specifically, the algorithms utilizing the Manhattan distance and misplaced tile heuristics demonstrate improved efficiency compared to the uniform cost algorithm.

Adapting the 8-Puzzle Solver to Larger Puzzle Dimensions:

It is a simple task to modify the program for solving the 8-puzzle to solve for larger puzzles such as the 15-puzzle or 24-puzzle.

Conclusions:

After analyzing the plots presented in figures 6, 7, and 8, it can be concluded that the three heuristics - Uniform Cost Search, Misplaced Tiles, and Manhattan Distance - performed as expected. Summarizing the findings from this assignment:

1. Manhattan Distance heuristic exhibited the best performance, followed by Misplaced Tiles, while Uniform Cost Search performed the worst among the three.

2. Both Misplaced Tiles and Manhattan Distance heuristics improved the time and space complexity of the algorithm. On the other hand, Uniform Cost Search, with its hardcoded h(n) as 0, transformed into a simple

Breadth First Search with time and space complexity of O(bd), where b represents the branching factor and d denotes the depth.

3. Both Misplaced Tiles and Manhattan Distance heuristics are admissible. However, the estimated cost provided by Manhattan Distance is always greater than or equal to that of Misplaced Tiles, making it the dominant choice in terms of performance between the two.

Overall, the study concludes that the selection of an appropriate heuristic greatly influences the efficiency and effectiveness of the search algorithm in solving the 8-puzzle problem.

The following is a trace of an Easy puzzle:

Input Settings

heuristic = Manhattan distance

Initial State = np.array([[1, 2, 3], [5, 0, 6], [4, 7, 8]], dtype=int)

Traceback of solution:

```
Choose your input option:
1 -> Select default option
2 -> Option for Custom choice
1
Time elapsed:  0.0015 sec
Solution Depth:  4
Number of Nodes Expanded:  5
Max queue size:  6
The solution steps are as follows:
The best state to expand with g(n) =  0 and h(n) =  4
[[1 2 3]
 [5 0 6]
 [4 7 8]]
The best state to expand with g(n) =  1 and h(n) =  3
[[1 2 3]
 [0 5 6]
 [4 7 8]]
The best state to expand with g(n) =  2 and h(n) =  2
[[1 2 3]
 [4 5 6]
 [0 7 8]]
The best state to expand with g(n) =  3 and h(n) =  1
[[1 2 3]
 [4 5 6]
 [7 0 8]]
The best state to expand with g(n) =  4 and h(n) =  0
[[1 2 3]
 [4 5 6]
[7 8 0]]
```

The following is a trace of a difficult puzzle:

Input Settings:

heuristic = Manhattan distance

Initial State = np.array([[0, 7, 2], [4, 6, 1], [3, 5, 8]], dtype=int)

Traceback of solution:

```
Choose your input option:
1 -> Select default option
2 -> Option for Custom choice
1
Time elapsed:  0.18 sec
Solution Depth:  24
Number of Nodes Expanded:  1388
Max queue size:  931
The solution steps are as follows:
The best state to expand with g(n) =  0 and h(n) =  14
[[0 7 2]
 [4 6 1]
 [3 5 8]]
The best state to expand with g(n) =  1 and h(n) =  13
[[7 0 2]
 [4 6 1]
 [3 5 8]]
The best state to expand with g(n) =  2 and h(n) =  14
[[7 6 2]
 [4 0 1]
 [3 5 8]]
.
.
.
The best state to expand with g(n) =  22 and h(n) =  2
[[1 2 3]
 [4 0 6]
 [7 5 8]]
The best state to expand with g(n) =  23 and h(n) =  1
[[1 2 3]
 [4 5 6]
 [7 0 8]]
The best state to expand with g(n) =  24 and h(n) =  0
[[1 2 3]
 [4 5 6]
 [7 8 0]]
```

code snippet:

```
import numpy as np
from heapq import heapify, heappop, heappush
import time
#Default Initial State
Dimension = 3
puzzle =Dimension**2 - 1
heuristic = 3
init_st = np.array([[1, 2, 3],
                   [5, 0, 6],
                   [4, 7, 8]], dtype=int)
#init = init_st
goal_st = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 0]], dtype=int)
# Choice of Algorithms
def evaluate(heuristic, node):
    #Uniform Cost Search
    if heuristic == 1:
        return 0
    #A* with the Misplaced Tile heuristic
    elif heuristic == 2:
      mismatch = np.sum(node != goal_st)
      return mismatch if mismatch > 0 else 0
    # A* with the Manhattan Distance heuristic
    elif heuristic == 3:
        dist = 0
        for i in range(1, puzzle + 1):
            pos = np.where(node == i)
            goal_pos = pos_map[i]
            dist += abs(pos[0][0] - goal_pos[0]) + abs(pos[1][0] -
goal_pos[1])
        return dist
    else:
        return -1
#Function for checking goal state
def goal_test(node):
    return np.array_equal(node, goal_st)

#Checking for 4 way valid move
def get_valid_moves(state, x, y):
    moves = []
    for i, j in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
        new_x, new_y = x + i, y + j
        if (0 <= new_x < state.shape[0]) and (0 <= new_y <
state.shape[1]):
            new_state = np.copy(state)
            new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
            moves.append(new_state)
    return moves
#Expansion of nodes in the valid direction
def expand(state):
    x, y = np.where(state == 0)
    return get_valid_moves(state, x[0], y[0])
pos_map = dict()
for i in range(1, puzzle+1):
    ind = np.where(goal_st == i)
    x, y = ind[0][0], ind[1][0]
    pos_map[i] = (x, y)

class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent
        self.g = 0
        self.h = evaluate(heuristic, state)
        if parent:
            self.g = parent.g + 1
    def f(self):
        return self.g + self.h
    def __lt__(self, other):
        return self.f() < other.f()
    def __eq__(self, other):
        return np.array_equal(self.state, other.state)

def a_star():
    nodes_expanded = 0
    queue = [Node(init_st)]
    heapify(queue)
    max_queue_size = 1
    while len(queue) > 0:
        current = heappop(queue)
        nodes_expanded = nodes_expanded + 1
        if goal_test(current.state):
            return current, nodes_expanded, max_queue_size
        children = [Node(ch, parent=current) for ch in
expand(current.state)]
        for ch in children:
            if current.parent is None or np.count_nonzero(ch.state ==
current.parent.state) < puzzle + 1:
                heappush(queue, ch)
        if max_queue_size < len(queue):
            max_queue_size = len(queue)
    return None, nodes_expanded, max_queue_size
if __name__ == '__main__':
    result, nodes_expanded, max_queue_size = None, 0, 0
    print("Choose your input option:\n1 -> Select default option\n2 ->
Option for Custom choice")
    choice = int(input())
    if choice == 1:
        start = time.time()
        result, nodes_expanded, max_queue_size = a_star()
        end = time.time()
        print("Time elapsed: ", end - start)
    elif choice == 2:

        print("Please input your puzzle, using a zero to represent the
blank space.")
        print("Enter the puzzle row by row, ensuring that only valid 8-
puzzles from 1 to 8 are entered. Separate each number with a space.")
        print("Enter the first row: ")
        r1 = input()
        print("Enter the second row: ")
        r2 = input()
        print("Enter the third row: ")
        r3 = input()
        init_st = np.array([list(map(int, r1.split())),
                    list(map(int, r2.split())),
                    list(map(int, r3.split()))])
        print("Choose the heuristic Algorithm:\n1 -> Uniform Cost
Search\n2 -> Misplaced Tiles\n3 -> Manhattan Distance")
        heuristic = int(input())
        result, nodes_expanded, max_queue_size = a_star()
    else:
        print("Your choice is invalid")
    if result is not None:
        solution_steps = []
        st = result
        while st is not None:
            solution_steps.append(st)
            st = st.parent
        print("Solution Depth: ", result.g)
        print("Number of Nodes Expanded: ", nodes_expanded)
        print("Max queue size: ", max_queue_size)
        print("The solution steps are as follows:")
        for k in reversed(solution_steps):
            print("The best state to expand with g(n) = ", str(k.g), "and
h(n) = ", str(k.h))
            print(k.state)
```