Attila Koksal

862058705

2/5/2025

CS170: Project 1: Eight Puzzle

In completing this assignment, I consulted:

- Lecture notes from Blind Search Part 2 slides for general search algorithm
- Lecture notes from Heuristic Search slides for A* search algorithm, A* with Misplaced Tile Heuristic, A* with Manhattan Distance Heuristic

All important code is original. Built-in functions like deepcopy and queue in Python have been used.

Outline of this report:

- Cover page (1st page)
- My Report (pages 2 to 5)
- Sample Results and Conclusion (pages 5 to 7)
- My Code (pages 7 to 26)

1. Introduction

a. In this project, I am supposed to solve the eight puzzle and even some higher-order puzzles like 15-puzzle and 25-puzzle. The challenge is to solve the eight puzzle using three search algorithms implemented, which are Uniform Cost Search, A* with the Misplaced Tile heuristic, and A* with the Manhattan Distance heuristic. The program interface allows the user to control the input size for the puzzle like 3 for 8-puzzle and 4 for 15-puzzle. In addition, the interface allows the user to input the initial state and goal state as well. Using the three search algorithms, the solution is the path from the initial state to the goal state, and the interface gives the option of selecting the default goal state, the sorted puzzle.

2. Algorithms

The three search algorithms are implemented in Python for this project and the search algorithm is very similar to pseudocode given in the project manual. The search algorithm takes into account the depth (g(n)) and heuristic (h(n)) cost functions.

a. Uniform Cost Search

i. The Uniform Cost Search algorithm ignores any heuristic cost and considers the node's depth from the root. In essence, the algorithm generates all possible moves from the starting position, where the blank space can move up, down, left, or right, by expanding the root node, or the puzzle's original state. One by one, these extended states are then processed after being placed to a queue. Uniform Cost Search performs

worse than the other algorithms because the algorithm does not prioritize the queue because the depth of all these enlarged states from the root is equal (g(n) = 1). Uniform Cost Search functions similarly to breadth-first search in real-world scenarios.

b. A* with the Misplaced Tile Heuristic

i. All feasible states are expanded from the original state by the A* algorithm. However, A* also considers a heuristic cost function. This function counts the number of tiles that are out of place, omitting the blank tile, when the Misplaced Tile heuristic is applied. The algorithm keeps track of a queue of nodes that have already been extended and sorts them according to the total of the heuristic and depth costs: g(n) + h(n). The next node to be expanded is the one with the lowest overall cost. Because it never overestimates the true cost to achieve the goal, this heuristic is acceptable. A* with the Misplaced Tile heuristic outperforms Uniform Cost Search since it tracks misplaced tiles and offers extra information.

c. A* with the Manhattan Distance Heuristic

i. The heuristic cost is computed differently in the A* Manhattan Distance heuristic algorithm. The Manhattan Distance heuristic calculates the total positional difference of each tile from its correct location rather than counting tiles that are misplaced. For instance, in the state listed below:

1. 1 # 3
2. 4 5 6
3. 7 8 2

ii. The Manhattan Distance heuristic value for tile "2" is three since it is two rows and one column from its goal position. Because it never overestimates the true cost to achieve the goal, this heuristic is acceptable. The Manhattan Distance algorithm performs better than the other two algorithms because it takes into consideration the precise movement needed to get to the right state.

3. Implementation

a. The project interface is made to be easy to use and intuitive. Users start by entering information such as the goal state, initial puzzle state, and puzzle size. The algorithm selection is the last necessary input. The target state is by default a solved puzzle. The two primary classes used in the project's structure are Node and PuzzleGame. Properties like the puzzle state, depth from the original state, heuristic value, and a Boolean flag indicating if the node has been extended are all stored in the Node class. Key variables such as the initial state, goal state, puzzle size, and initial node (although the initial state alone can be adequate) are included in the PuzzleGame class, which specifies the challenge.

b. In order to move the blank tile in the following valid directions—up, down, left, or right—and show the state matrix, additional utility methods are implemented. Border conditions are examined to make sure the movement is feasible in order to accomplish this. Furthermore, as it is not yet known whether a 25-puzzle can be solved, the program checks to see if the 8-puzzle or 15-puzzle can be solved. In particular, only half of the 9! (362,880) potential states for the 8-puzzle may be solved. Counting inversions is the basis for the solvability check; if the number of

inversions is even, the problem can be solved; if not, it cannot. This technique is only applicable when the default goal state is being used; otherwise, the procedure can only run for an hour at most.

4. Results and Conclusions

Figure 1: Expanded Nodes vs. Solution depth

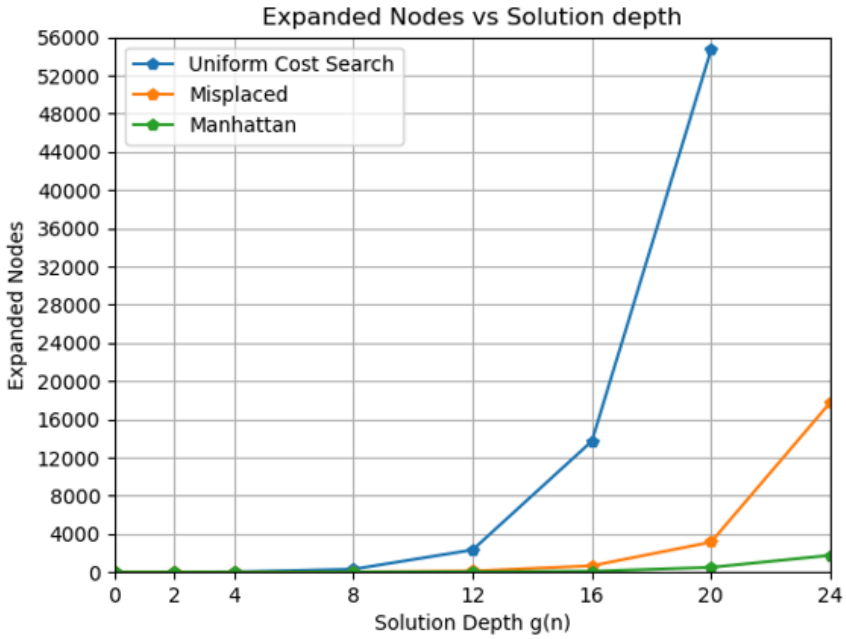


Figure 2: Maximum Queue Size vs. Solution depth

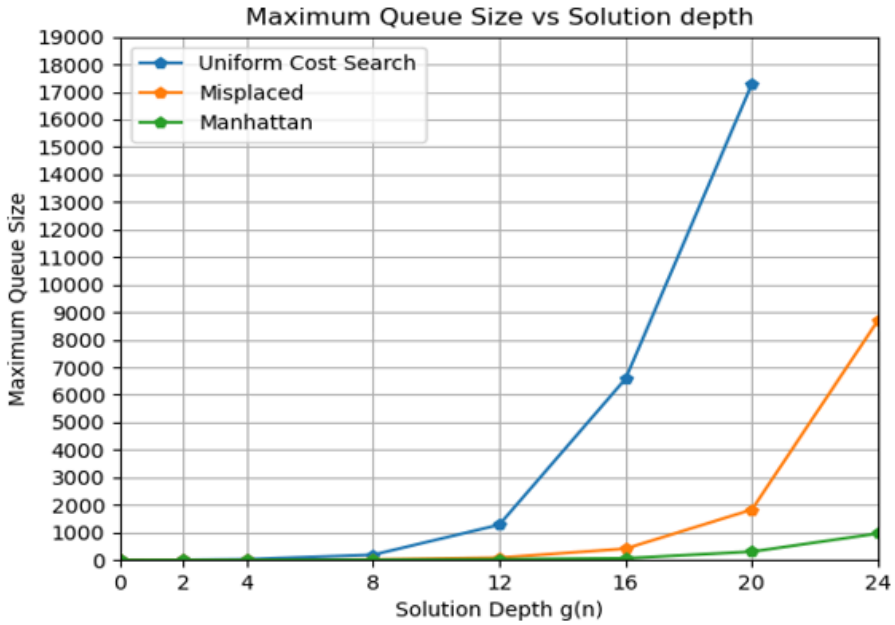
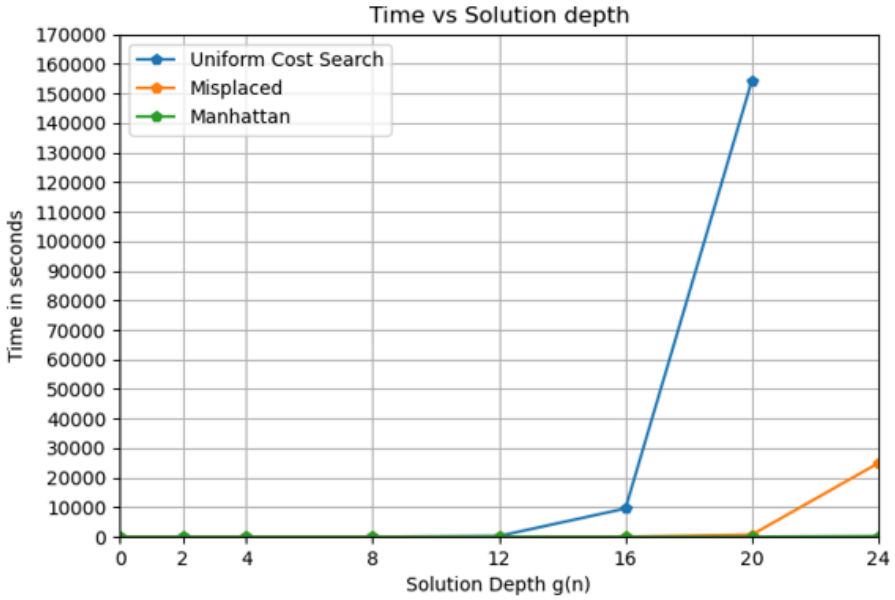


Figure 3: Time vs. Solution depth



In comparison to A* Manhattan Distance heuristic and A* Misplaced Tile heuristic, which expand fewer nodes to reach the goal state, Uniform Cost Search extends a greater number of nodes, as illustrated in Figure 1. This happens because, regardless of the real differences between

the children, Uniform Cost Search only takes into account the depth from the parent node, which is always 1. Because the A* Misplaced Tile heuristic only considers the amount of tiles that are misplaced and ignores positional differences, it produces more extended nodes than A* Manhattan Distance heuristic. Figures 2 and 3 provide similar insights.

In conclusion, in this project, I implemented three different search algorithms, which were Uniform Cost Search, A* with Misplaced Tile heuristic, and A* with Manhattan Distance heuristic. When analyzing the results in the figures above, the A* with Manhattan Distance heuristic algorithm beats the other two algorithms in terms of performance, especially if the solution depth is greater than 12.

nPuzzle.py

```
import matplotlib.pyplot as plt
import time
import copy
import sys


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


# saves input node with properties
class Node:
    def __init__(self, nodeState, nodeDepth=0, expanded=False):
        self.parent_nodes = []
        self.heuristic = 0
```

```
        self.expanded = expanded
        self.nodeState = nodeState
        self.nodeDepth = nodeDepth

    def getState(self):
        return self.nodeState

# creates a puzzle game with properties
class PuzzleGame:
    def __init__(self, initialState, goal_state, node=Node, length=3):
        self.node = node
        self.puzzle_length = length
        self.initialState = initialState
        self.goal_state = goal_state

    def setInitialState(self, state):
        self.initialState = state

    def goal_state(self):
        return self.goal_state

# update position in puzzle
def movement(currentState, action=0):
```

```
for i in currentState:
    for k in i:
        if k == 0:
            index = (currentState.index(i), i.index(k))
# left
if action == 2:
    if index[1] != 0:
        currentState[index[0]][index[1]] = currentState[index[0]][index[1] - 1]
        currentState[index[0]][index[1] - 1] = 0
        return currentState
    else:
        return None


# right
if action == 3:
    if index[1] != len(currentState)-1:
        currentState[index[0]][index[1]] = currentState[index[0]][index[1] + 1]
        currentState[index[0]][index[1] + 1] = 0
        return currentState
    else:
        return None
# up
if action == 0:
```

```
        if index[0] != 0:
            currentState[index[0]][index[1]] = currentState[index[0] - 1][index[1]]
            currentState[index[0] - 1][index[1]] = 0
            return currentState
        else:
            return None
    # down
    if action == 1:
        if index[0] != len(currentState)-1:
            currentState[index[0]][index[1]] = currentState[index[0] + 1][index[1]]
            currentState[index[0] + 1][index[1]] = 0
            return currentState
        else:
            return None


# verify existence of blank
def isStateValid(matrix, row):
    for i in range(row):
        for k in range(row):
            if matrix[i][k] == 0:
                return True
    return False
```

```
# display states to console
def printState(currentState):
    for i in currentState:
        print( "-------")
        string = "|"
        for k in i:
            string += str(k) + "|"
        print(string)
    print( "-------")


# generates every possible node
def possible_states(stateMatrix):
    expansions_matrix = []


    up = copy.deepcopy(stateMatrix)
    down = copy.deepcopy(stateMatrix)
    left = copy.deepcopy(stateMatrix)
    right = copy.deepcopy(stateMatrix)


    expansions_matrix.append(movement(up, 0))
    expansions_matrix.append(movement(down, 1))
    expansions_matrix.append(movement(left, 2))
    expansions_matrix.append(movement(right, 3))
```

```
    return expansions_matrix


# user inteface for algorithm selection
def get_algo():
    print("Select an algorithm: ")
    print("1: Uniform Cost Search")
    print("2: A* with the Misplaced Tile heuristic")
    print("3: A* with the Manhattan Distance heuristic")
    algorithm = input("Your choice: ")
    try:
        algorithm = int(algorithm)
    except ValueError:
        print("Invalid Input: " + algorithm)
    if algorithm < 1 or algorithm > 3:
        print("Enter 1, 2, or 3!")


    return algorithm


# search algorithm
def general_search(puzzle=PuzzleGame, function=1):


    time_started = time.time()
```

```
myQueue = []
visited_states = []
n_visited = 0
queue_size = 0
maxQueueSize = 0
goal_state = puzzle.goal_state

if function == 1:
    heuristic = 0
elif function == 2:
    heuristic = heuristicMisplaced(puzzle.initialState, goal_state)
elif function == 3:
    heuristic = heuristicManhattan(puzzle.initialState, goal_state)

node = Node(puzzle.initialState, 0, False)
node.heuristic = heuristic

myQueue.append(node)
visited_states.append(node.nodeState)
queue_size += 1
maxQueueSize += 1
```

```
while True:

    if function != 1:
        myQueue = sorted(myQueue, key=lambda x: (x.nodeDepth + x.heuristic, x.nodeDepth))

    if len(myQueue) == 0:
        print("Failure! No solution found within an hour!")
        return None, None, None

    nextNode = myQueue.pop(0)
    queue_size -= 1

    if nextNode.expanded == False:
        n_visited += 1
        nextNode.expanded = True

    if n_visited != 1:
        print("Best node chosen to expand with g(n) = " + str(nextNode.nodeDepth) + \
              " and h(n) = " + str(nextNode.heuristic) + ":")
        printState(nextNode.getState())
    else:
        print("Expanding node: ")
        printState(nextNode.getState())
```

```
if nextNode.getState() == goal_state:


    print("Success:)")
    print("The total of "+str(n_visited-1)+" nodes have been expanded\
        \nThe max number of nodes/states in the queue in one time instance is "\
        + str(maxQueueSize) + "\nTime taken is " + \
          str((time.time()-time_started)*1000) + " seconds")
    printState(nextNode.getState())
    return n_visited-1, maxQueueSize, (time.time()-time_started)*1000


expansion = possible_states(nextNode.getState())


for child_state in expansion:
    if isStateVisited(child_state, visited_states):
        child_node = Node(child_state)
        if function == 1:
            child_node.nodeDepth = nextNode.nodeDepth + 1
            child_node.heuristic = 0
        elif function == 2:
            child_node.nodeDepth = nextNode.nodeDepth + 1
            child_node.heuristic = heuristicMisplaced(child_node.getState(), goal_state)
        elif function == 3:
```

```
                child_node.nodeDepth = nextNode.nodeDepth + 1
                child_node.heuristic = heuristicManhattan(child_node.getState(), goal_state)

            myQueue.append(child_node)
            visited_states.append(child_node.getState())
            queue_size += 1

        if queue_size > maxQueueSize:
            maxQueueSize = queue_size

        if time.time() - time_started > 1*60*60*1000:
            print("An hour passed but no solution has been found. Exit!!!")

# checks whether state is visited or not
def isStateVisited(state, array):
    if state is None:
        return False

    for i in range(len(array)):
        if state == array[i]:
            return False

    return True
```

```
# the heuristic error algorithm
def heuristicMisplaced(currentState, goal_state):
    h = 0
    length = len(currentState)

    for i in range(length):
        for j in range(length):
            if int(currentState[i][j]) != goal_state[i][j] and int(currentState[i][j]) != 0:
                h += 1
    return h

# the heuristic manhattan distance
def heuristicManhattan(currentState, goal_state):
    h = 0
    length = len(currentState)
    gr, gc, r, c = 0, 0, 0, 0

    for l in range(1, length*length):
        for i in range(length):
            for j in range(length):
                if int(currentState[i][j]) == l:
                    r = i
```

```
                c = j
            if goal_state[i][j] == l:
                gr = i
                gc = j
        h += abs(gr-r) + abs(gc-c)

    return h



# generate a custom input and define a goal state
def create_states():
    print("Enter number of rows (=columns) in input or press Enter for 8-puzzle:")
    try:
        row_number = input() or int(3)
        row_number = int(row_number)
        if row_number <= 0:
            print("Invalid number for rows")
            raise ValueError

    except ValueError:
        print("Error: input a valid number for the number of rows in input")
        exit()
```

```
print("The selected number of rows is:" + str(row_number))
print("Now, let's create the input matrix or state")

initialState = []
for i in range(row_number):
    print("Enter the " + str(i + 1) +
          "-th row with space in between numbers and 0 for blank:")

    input_row = input()
    try:
        input_row = [int(a) for a in input_row.split()]
    except ValueError:
        print("Invalid input!")
        exit()

    initialState.append(input_row)

if not isStateValid(initialState, row_number):
    print("There is no blank symbol; 0 in the state matrix")
    return None, None

print("Initial state:")
printState(initialState)
```

```
    # default goal states:
    goal_state = [[(k+1) + (row_number*m) for k in range(row_number)] for m in
range(row_number)]
    goal_state[row_number-1][row_number-1] = 0 # blank state

    print("Do you want to have default goal states: y/n, press Enter for y")
    print("Default goal state looks like:")
    printState(goal_state)

    mode = input() or "y"
    if mode != "y":
        for i in range(row_number):
            print("Enter the " + str(i + 1) +
              "-th row with space in between numbers and 0 for blank:")

        input_row = input()
        try:
            input_row = [int(a) for a in input_row.split()]
        except ValueError:
            print("Invalid input!")
            exit()
```

```
        goal_state.append(input_row)

    if not isStateValid(initialState, row_number):
        print("There is no blank symbol; 0 in the state matrix")
        return None, None

    return initialState, goal_state

# checks if puzzle is solvable or not
def checkSolvability(state):
    length = len(state)
    total_inversion = 0
    for row in range(length):
        for column in range(length):
            total_inversion += getInversion(state, row, column)
    if total_inversion % 2 == 1:
        return False
    return True

# finds the inversion
def getInversion(state, row, column):
    length = len(state)
    inversion = 0
```

```
    for i in range(length):
        for k in range(length):
            if i >= row and k >= column:
                if state[row][column] > state[i][k] and state[i][k] != 0:
                    inversion += 1
    return inversion


# plots all algorithms
def plot():
    d0 = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    d2 = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
    d4 = [[1, 2, 3], [5, 0, 6], [4, 7, 8]]
    d8 = [[1, 3, 6], [5, 0, 2], [4, 7, 8]]
    d12 = [[1, 3, 6], [5, 0, 7], [4, 8, 2]]
    d16 = [[1, 6, 7], [5, 0, 3], [4, 8, 2]]
    d20 = [[7, 1, 2], [4, 8, 5], [6, 3, 0]]
    d24 = [[0, 7, 2], [4, 6, 1], [3, 5, 8]]
    depth = [[[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]]]
dArray = [0, 2, 4, 8, 12, 16, 20, 24]
nodeArr1 = []
maxQArr1 = []
t1 = []
nodeArr2 = []
maxQArr2 = []
t2 = []
nodeArr3 = []
maxQArr3 = []
t3 = []
puzzle = PuzzleGame(d0, goalState, len(goalState))
printState(puzzle.goal_state)

for i in range(len(dArray)-1):
    puzzle.setInitialState(depth[i])
    numNodesExpanded, maxQueueSize, time = general_search(puzzle, 1)

    nodeArr1.append(numNodesExpanded)
    maxQArr1.append(maxQueueSize)
    t1.append(time)
```

```
for i in range(len(dArray)):
    puzzle.setInitialState(depth[i])
    numNodesExpanded, maxQueueSize, time = general_search(puzzle, 2)

    nodeArr2.append(numNodesExpanded)
    maxQArr2.append(maxQueueSize)
    t2.append(time)

for i in range(len(dArray)):
    puzzle.setInitialState(depth[i])
    numNodesExpanded, maxQueueSize, time = general_search(puzzle, 3)

    nodeArr3.append(numNodesExpanded)
    maxQArr3.append(maxQueueSize)
    t3.append(time)

print("Node Array 1:")
print(nodeArr1)
print("Max Queue Array 1:")
print(maxQArr1)
print("Node Array 2:")
print(nodeArr2)
print("Max Queue Array 2:")
```

```
print(maxQArr2)
print("Node Array 3:")
print(nodeArr3)
print("Max Queue Array 3:")
print(maxQArr3)
print("3rd time")
print(t3)


plt.plot(dArray[0:-1], t1, marker='p', label='Uniform Cost Search')
plt.plot(dArray, t2, marker='p', label='Misplaced')
plt.plot(dArray, t3, marker='p', label='Manhattan')
plt.xlabel("Solution Depth g(n)")
plt.ylabel("Time in seconds")
plt.title("Time vs Solution depth")
plt.legend(loc="upper left")
plt.grid()
plt.margins(x=0, y=0)
plt.xticks(dArray)
plt.yticks([*range(0, 180000, 10000)])
plt.rc('xtick', labelsize=20)
plt.rc('ytick', labelsize=20)


plt.show()
```

```
def main():
    initialState, goal_state = create_states()
    if len(initialState) <= 4:
        if goal_state == goalState:
            if checkSolvability(initialState):
                print("Solvable")
            else:
                print("Unsolvable")
                sys.exit()
        else: print("Since the goal state is different"
                    "the solvability check is not done. \n"
                    "If the puzzle is unsolvable, the system will exit after some time)")
    else: print("Solvability check cannot be done for this puzzle")


    algorithm = get_algo()
    puzzle = PuzzleGame(initialState, goal_state, len(initialState))


    print("Search Started")
    numNodesExpanded, maxQueueSize, time = general_search(puzzle, algorithm)


if __name__ == "__main__":
    main()
```