Project 1: The Eight Puzzle

Name: Shaoyu Tu
SID: 862137500
Email: stu024@ucr.edu
Date: May 03, 2024
Github Link (https://github.com/Kevin20201/CS_205)

The following report's format takes inspiration from Dr. Eamonn Keogh's example report. [1].

Below is a list of links and short description attached that I consulted in order to complete this project.

- StackOverflow for creating citation links in markdown: https://stackoverflow.com/questions/26587527/cite-a-paper-using-github-markdown-syntax
- CodeHS to understand how to retreive input from users cli: https://codehs.com/tutorial/rachel/user-input-in-python#:~:text=In Python%2C we use the,the information in our program
- Geeks for Geeks on how to call deepcopy(): https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/
- The Python Standard Library for understanding the use of its built-in Queue data structure: https://docs.python.org/3/library/queue.html
- StackOverflow to understand how to create a generic tree node class: https://stackoverflow.com/questions/2482602/a-general-tree-implementation
- The Python Tutorial to understand how to write a problem class: https://docs.python.org/3/tutorial/classes.html
- Professor Eamon Keogh's slide deck 3__Heuristic Search 2 slide 4 for setting heuristic values
- Professor Eamon Keogh's operators variable naming convention from Slide deck 2__Blind Search_part1 slide 27

All major parts in solving the 8-puzzle code is original. Subroutines borrowed from Python libraries includes...

- Subroutines from `copy`: `deepcopy()` function is used to distinguish between copy by value from the standard copy by reference.
- Subroutines from `queue`: `Queue()` and `PriorityQueue()` classes are used to handle the different A* heuristics. Functions such as `empty()`, `get()`, and `qsize()` were used to check the queue status.

"*I affirm that I did not use ChatGPT or similar to write the code or text in this work.*" [1]

Contents

Objective

This project aims to solve the 8-puzzle, a sliding puzzle, with heuristic search. Specifically, our objective is to implement the Uniform Cost Search and A* Search to then evaluate their performance.

The sliding puzzle comes in a variety of sizes such as 4x4, 5x5, 6x6, and etc. But to make the explanation simple, our focus for this report will just be on the 3x3 or 8-puzzle. The 8-puzzle, similar to most other puzzles can be broken down into a simple search problem.

Below we have defined our problem space.

- The representation of a state is having 8 tiles in a locked position such that there is one empty tile at any given time.
- The initial state is setting up the board such that there are 8 tiles with a distinct empty tile in the 3x3 grid.

- The operators that are acceptable in our case is to think of the puzzle as moving the empty tile. Therefore, the legal moves for the empty tile is at most 4 distinct moves (up, down, left, or right) provided its current position on the board.
- The goal state is to have the tiles end up in ascending order such that for the 8-puzzle, we would have values 1, 2, and 3 on top row, 4, 5, and 6 in the middle row, and lastly 7, 8, and the empty tile be on the last row.

Algorithm

For our implementation, we were asked to implement 3 algorithms for our 8-puzzle, the Uniform Cost Search, the A* Search with the Misplaced Tile heuristic, and the A* Search with the Manhattan Distance heuristic.

Uniform Cost Search

We were provided a reminder in the project description that the "Uniform Cost Search is [simply] the A* search [provided a heuristic function] hard coded to zero" [1]. Therefore, the Uniform Cost Search in implementation for this problem is also equivalent to running Breadth First Search. The order for travering the node is simply the cost function, g(n), which is set to the depth of the node. Thus, the algorithm will just search depth by depth providing a solution that is both optimal and complete.

A* Search with Misplaced Tile Heuristic

For our first useful heuristic, we implement the misplaced tile heuristic. As its name suggests, this heuristic simply returns a count equal to the number of tiles misplaced when compared to the goal state excluding the empty tile. The cost function for this algorithm is still the depth of the tree which is equal to how many steps it has already taken from the initial state.

A* Search with Manhattan Distance Heuristic

For this final heuristic, we implement the manhattan distance heuristic. The Manhattan Distance heuristic acts similar to the Misplaced Tile heuristic in that it accounts for all tiles misplaced when compared to the goal state. But instead of a simple incremented count, the Manhattan Distance heuristic counts how far away the tile is currently from its destination while restricting its movement to a 2d space (up, down, left and right).

Evaluation

We were given a set of puzzles to test our implementation along with their optimal solution depth and below are results from these test cases with optimal solutions at depth, 0, 2, 4, 8, 12, 16, 20, and 24.

The first graph we will look at is Figure 1 as shown below. There are a few things we can take away from this graph. We first notice that puzzles with depth 8 or shallower have similar count of nodes expanded between all 3 algorithms. However, in the case of Uniform Cost search where no heuristic function exists, this depth rapidly grows to as much as 50 times larger when compared to the best performing algorithm, the A* with Manhattan Distance heuristic search. Lastly, we also see that with the additional heuristic functions in A*, the total nodes expanded is similar between the two algorithms.

Nodes Traversed vs Depth

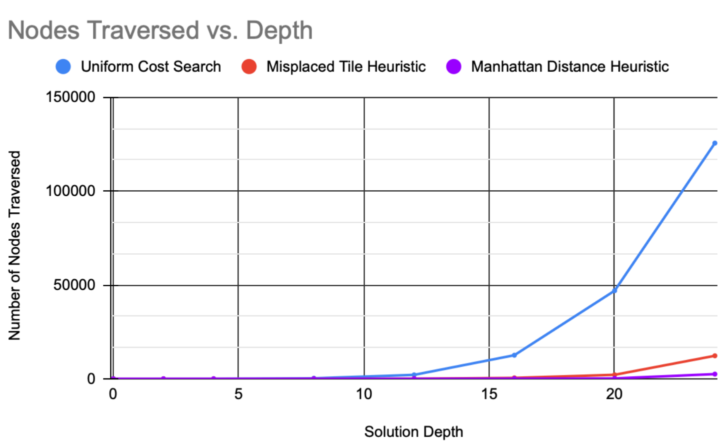

Figure 1: *A graph showcasing the number of nodes traversed as solution depth grows.*

Figure 2 shows the relationship between max queue size and solution depth from the puzzles. Similar to Figure 1, we see trends of the differences being negligible for puzzles that have solution of depth 8 or shallower. Then we see this rapid growth in the queue size for Uniform Cost search as the depth of the solution increases. For the relationship between max queue size and depth the difference is over 17 times between Uniform Cost search, the largest, and A* with Manhattan Distance heuristic search, the smallest, when comparing solution depth at 24. Finally, we also notice that the difference between the two A* search algorithms have similar proportional differences between the two figures at each test case.

Max Queue Size vs Depth

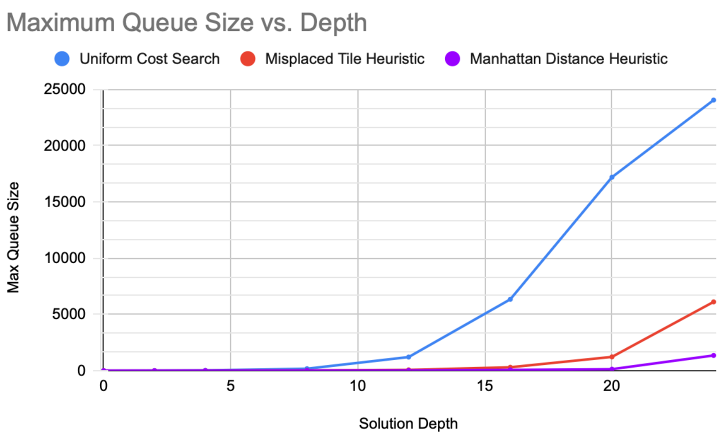

Figure 2: *A graph showcasing the maximum queue size as solution depth grows.*

Final Thoughts

It was shocking to see what a drastic difference it makes when a useful heuristic function was chosen for A* over not having one at all. These heuristic functions that we tried are so simple in its idea giving an estimate to how many more moves we think it would take to solve the puzzle. Even with its simplicity, we are still able to capture a massive improvement over not having one.

The order from best performing to worst given the 3 algorithms we implemented ranks as follows: A* Search with the Manhattan Distance heuristic, A* Search with the Misplaced Tile heuristic, and Uniform Cost Search or in our project being equivalent to Breadth First Search and A* with heuristic function outputting the value 0.

These test cases with optimal solutions ranging from 0 to 24 shows us that given puzzles with shallow solution depths (8 or less) the algorithm of choice is not so important. But when it comes to searching for solutions at for example depth 20, choosing the A* with a useful heuristic becomes imperative.

Trace

The following is a traceback for a puzzle of depth 4 with Manhattan Distance Heuristic.

```
❯ python3 8_puzzle.py
Welcome to N puzzle solver!
What size puzzle would you like to generate? (e.g. 8, 15, 25)
Please INSERT a valid natural number and press ENTER: 8

Please INSERT a valid puzzle you would like to test row by row.
Note: for the empty space in the puzzle, please INSERT the number 0.

Please INSERT values for row 1 with a space in between each number:
Press ENTER when you are ready.
1 2 3

Please INSERT values for row 2 with a space in between each number:
Press ENTER when you are ready.
5 0 6

Please INSERT values for row 3 with a space in between each number:
Press ENTER when you are ready.
4 7 8

Please select the algorithm you would like the program to use by entering its correspoding number as shown:

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

Select an algortihm and press ENTER: 3

The program has successfully loaded the puzzle and will begin to search for the goal_state...

Calling the function...
SUCCESS
Printing the Traversed Solution...

Node to expand has g(n) = 4 and h(n) = 0 is...
[[1 2 3]
[4 5 6]
[7 8 0]]

Node to expand has g(n) = 3 and h(n) = 1 is...
[[1 2 3]
[4 5 6]
[7 0 8]]

Node to expand has g(n) = 2 and h(n) = 2 is...
[[1 2 3]
[4 5 6]
[0 7 8]]

Node to expand has g(n) = 1 and h(n) = 3 is...
[[1 2 3]
[0 5 6]
[4 7 8]]

Node to expand has g(n) = 0 and h(n) = 4 is...
[[1 2 3]
[5 0 6]
[4 7 8]]


Max Queue Size:  6

Total Nodes Traversed:  5

Solution Depth:  4
```

The following is a traceback for a puzzle of depth 16 with Misplaced Tile Heuristic.

```
❯ python3 8_puzzle.py
Welcome to N puzzle solver!
What size puzzle would you like to generate? (e.g. 8, 15, 25)
Please INSERT a valid natural number and press ENTER: 8

Please INSERT a valid puzzle you would like to test row by row.
Note: for the empty space in the puzzle, please INSERT the number 0.

Please INSERT values for row 1 with a space in between each number:
Press ENTER when you are ready.
1 6 7

Please INSERT values for row 2 with a space in between each number:
Press ENTER when you are ready.
5 0 3

Please INSERT values for row 3 with a space in between each number:
Press ENTER when you are ready.
4 8 2

Please select the algorithm you would like the program to use by entering its correspoding number as shown:

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

Select an algortihm and press ENTER: 2

The program has successfully loaded the puzzle and will begin to search for the goal_state...

Calling the function...
SUCCESS
Printing the Traversed Solution...

Node to expand has g(n) = 16 and h(n) = 0 is...
[[1 2 3]
[4 5 6]
[7 8 0]]

Node to expand has g(n) = 15 and h(n) = 1 is...
[[1 2 3]
[4 5 6]
[7 0 8]]

... # Removed traces to shorten the print

Node to expand has g(n) = 2 and h(n) = 6 is...
[[1 6 0]
[5 3 7]
[4 8 2]]

Node to expand has g(n) = 1 and h(n) = 6 is...
[[1 6 7]
[5 3 0]
[4 8 2]]

Node to expand has g(n) = 0 and h(n) = 6 is...
[[1 6 7]
[5 0 3]
[4 8 2]]


Max Queue Size:  301

Total Nodes Traversed:  489

Solution Depth:  16
```

The following is a traceback for a 15-puzzle of depth 2 with Manhattan Distance Heuristic to show that the implementation is also generalized to allow puzzle of different input sizes.

```
❯ python3 8_puzzle.py
Welcome to N puzzle solver!
What size puzzle would you like to generate? (e.g. 8, 15, 25)
Please INSERT a valid natural number and press ENTER: 15

Please INSERT a valid puzzle you would like to test row by row.
Note: for the empty space in the puzzle, please INSERT the number 0.

Please INSERT values for row 1 with a space in between each number:
Press ENTER when you are ready.
1 2 3 4

Please INSERT values for row 2 with a space in between each number:
Press ENTER when you are ready.
5 6 7 8

Please INSERT values for row 3 with a space in between each number:
Press ENTER when you are ready.
9 10 11 12

Please INSERT values for row 4 with a space in between each number:
Press ENTER when you are ready.
13 0 14 15

Please select the algorithm you would like the program to use by entering its correspoding number as shown:

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

Select an algortihm and press ENTER: 3

The program has successfully loaded the puzzle and will begin to search for the goal_state...

Calling the function...
SUCCESS
Printing the Traversed Solution...

Node to expand has g(n) = 2 and h(n) = 0 is...
[[1 2 3 4]
[5 6 7 8]
[9 10 11 12]
[13 14 15 0]]

Node to expand has g(n) = 1 and h(n) = 1 is...
[[1 2 3 4]
[5 6 7 8]
[9 10 11 12]
[13 14 0 15]]

Node to expand has g(n) = 0 and h(n) = 2 is...
[[1 2 3 4]
[5 6 7 8]
[9 10 11 12]
[13 0 14 15]]


Max Queue Size:  4

Total Nodes Traversed:  3

Solution Depth:  2
```

Implementation

Github Link (https://github.com/Kevin20201/CS_205)

8_puzzle.py

```
import numpy as np
import sys
##### Referenced Geeks for Geeks on how to call deepcopy() https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/
import copy
##### Referenced The Python Standard Library https://docs.python.org/3/library/queue.html
from queue import PriorityQueue
from queue import Queue
import time

##### Referenced StackOverflow https://stackoverflow.com/questions/2482602/a-general-tree-implementation to see how to create a generic tree structure
You, yesterday | 1 author (You)
class Tree:
    def __init__(self, state, parent, depth, heuristic):
        self.state = state
        self.parent = parent
        self.depth = depth
        self.heuristic = heuristic

    def get_parent(self):
        return self.parent

    def get_state(self):
        return self.state

##### Referenced The Python Tutorial https://docs.python.org/3/tutorial/classes.html to understand how to write a problem class
You, 41 minutes ago | 1 author (You)
class Problem:
    def __init__(self, initial_state, goal_state, puzzle_size, heuristic):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.size = puzzle_size
        self.heuristic = heuristic

    def node_weight(self, puzzle):
        weight = 0
        # uniform cost search
        if (self.heuristic == 1):
            # For this problem it is considered breath first search
            # Referenced Professor Eamon Keogh's slide deck 3__Heuristic Search 2 slide 4
            return 0
        # misplaced tile heuristic
        elif (self.heuristic == 2):
            for row in range(self.size):
                for column in range(self.size):
                    # Check if location of value is same as goal state (expected)
                    if puzzle[row][column] != self.goal_state[row][column]:
                        if puzzle[row][column] == 0:
                            continue
                        else:
                            weight+=1
            return weight
        #manhattan Distance heuristic\n")
        elif (self.heuristic == 3):
            for row in range(self.size):
                for column in range(self.size):
                    value = puzzle[row][column]
                    # The row for the value lies in the multiplplicity of the size value -1 because of indexing
                    correct_row = int(value / self.size)
                    # The column is the remainder of the value from the size so we take the mod
                    correct_column = int(value % self.size)
                    # If value is the blank spot, then row is set at the bottom
                    if value == 0:
                        correct_row=self.size
                        continue
                    # Adjustment for indexing on row and special cases for modulo 0 values
                    if correct_column == 0:
                        correct_column = self.size-1
                        correct_row-=1
                    # Adjustment for indexing
                    else:
                        correct_column-=1
                    # Calculate the Manhattan Distance of row and column
                    weight_row = abs(row - correct_row)
                    weight_column = abs(column - correct_column)
                    weight += (weight_row + weight_column)
            return weight
        return weight

##### Referenced Professor Eamon Keogh's operators variable naming convention from Slide deck 2__Blind Search_part1 slide 27
# Move blank left swaps blank with the value on its left
def move_blank_left(node, row, column):
    # Make copy so we dont overwrite
    puzzle = copy.deepcopy(node)
    blank = puzzle[row][column]
    temp = puzzle[row][column-1]
    puzzle[row][column-1] = blank
    puzzle[row][column] = temp
    return puzzle

# Move blank left swaps blank with the value on its right
def move_blank_right(node, row, column):
    # Make copy so we dont overwrite
    puzzle = copy.deepcopy(node)
    blank = puzzle[row][column]
    temp = puzzle[row][column+1]
    puzzle[row][column+1] = blank
    puzzle[row][column] = temp
    return puzzle

```

```python
# Move blank left swaps blank with the value above
def move_blank_up(node, row, column):
    # Make copy so we dont overwrite
    puzzle = copy.deepcopy(node)
    blank = puzzle[row][column]
    temp = puzzle[row-1][column]
    puzzle[row-1][column] = blank
    puzzle[row][column] = temp
    return puzzle

# Move blank left swaps blank with the value below
def move_blank_down(node, row, column):
    # Make copy so we dont overwrite
    puzzle = copy.deepcopy(node)
    blank = puzzle[row][column]
    temp = puzzle[row+1][column]
    puzzle[row+1][column] = blank
    puzzle[row][column] = temp
    return puzzle

def expand(node, problem, queue, states):
    # First find where the blank is located
    blank_position = None
    puzzle = None
    for row in range(problem.size):
        for column in range(problem.size):
            # If value is 0 (blank) then return its position
            if node[1][row][column] == 0:
                blank_position = (row, column)
                break
        if blank_position is not None:
            break
    # Check valid operators then perform the move if valid
    # Corner Cases: Top Left, Top Right, Bottom Left, Bottom Right
    if blank_position == (0,0):
        puzzle = move_blank_right(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_down(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
    elif blank_position == (0,problem.size-1):
        puzzle = move_blank_left(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_down(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
```

```python
    elif blank_position == (problem.size-1,0):
        puzzle = move_blank_right(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_up(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
    elif blank_position == (problem.size-1,problem.size-1):
        puzzle = move_blank_left(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_up(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
    # Border Cases: Top, Left Side, Right Side, Bottom
    elif blank_position[0] == 0:
        puzzle = move_blank_left(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_right(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_down(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
    elif blank_position[1] == 0:
        puzzle = move_blank_up(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_right(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_down(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
    elif blank_position[1] == problem.size-1:
        puzzle = move_blank_up(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_left(node[1], blank_position[0], blank_position[1])
```

```python
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_down(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
    elif blank_position[0] == problem.size-1:
        puzzle = move_blank_up(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_left(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_right(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
    # For the rest all four operations are valid
    else:
        puzzle = move_blank_up(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_left(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_right(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
        puzzle = move_blank_down(node[1], blank_position[0], blank_position[1])
        if puzzle not in states:
            states.append(puzzle)
            queue.put((problem.node_weight(puzzle)+node[2].depth+1, puzzle, Tree(puzzle, copy.deepcopy(node[2]), node[2].depth+1, problem.node_weight(puzzle))))
    return

def print_node(puzzle, problem, node):
    stats = 'Node to expand has g(n) = ' + str(node.depth) + ' and h(n) = ' + str(node.heuristic) + ' is... '
    print(stats)
    print_puzzle = '['
    for row in range(problem.size):
        print_puzzle += '['
        for column in range(problem.size):
            print_puzzle += str(puzzle[row][column])
            print_puzzle += ' '
        print_puzzle = print_puzzle[:-1] + ']\n'
    print_puzzle = print_puzzle[:-1] + ']\n'
    print(print_puzzle)
    return
```

```python

def print_solution(node, solution_depth, problem):
    traverse_node = node[2]
    while traverse_node.get_parent() is not None:
        print_node(traverse_node.state, problem, traverse_node)
        traverse_node = traverse_node.get_parent()
        solution_depth+=1
    print_node(traverse_node.state, problem, traverse_node)
    return solution_depth

##### Pseudo Code Main "Driver" Program from Professor Eamon Keogh's slides
# function general-search(problem, QUEUEING-FUNCTION)
# nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
# loop do
# if EMPTY(nodes) then return "failure"
# node = REMOVE-FRONT(nodes)
# if problem.GOAL-TEST(node.STATE) succeeds then return node
# nodes = QUEUEING-FUNCTION(nodes, EXPAND(node, problem.OPERATORS))
# end

# Recall that Uniform Cost Search is just A* with h(n) hardcoded to equal zero.
# Quote above was taken from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
def a_star_search(problem, queue):
    # Keep track of repeated states
    ##### Referenced w2school https://www.w3schools.com/python/python_sets.asp on how to use sets in python
    states = [problem.initial_state]
    # Inserts initial_state in the queue as a tuple (weight, puzzle)
    queue.put((problem.node_weight(problem.initial_state), problem.initial_state, Tree(problem.initial_state, None, 0, problem.node_weight(problem.initial_state))))
    node = None
    max_queue_size = max(queue.qsize(), 0)
    total_nodes_traversed = 0
    solution_depth = 0
    while True:
        # If queue is empty then there is no solution to the puzzle
        if queue.empty():
            print("FAILURE")
            return
        # Otherwise we continue to retrieve from the queue
        node = queue.get()
        total_nodes_traversed += 1
        if problem.goal_state == node[1]:
            print("SUCCESS")
            print("Printing the Traversed Solution...\n")
            solution_depth = print_solution(node, solution_depth, problem)
            print("\nMax Queue Size: ", max_queue_size)
            print("\nTotal Nodes Traversed: ", total_nodes_traversed)
            print("\nSolution Depth: ", solution_depth)
            return
        # Otherwise we check valid operators and insert results into queue
        # Call the expand function to only traverse valid states
        expand(node, problem, queue, states)
        max_queue_size = max(queue.qsize(), max_queue_size)

```

```python
##### Referenced CodeHS to understand how to retreive input from users.
# https://codehs.com/tutorial/rachel/user-input-in-python#:~:text=In%20Python%2C%20we%20use%20the,the%20information%20in%20our%20program.
if __name__ == "__main__":
    # Asks the user for puzzle_size
    print("Welcome to N puzzle solver!")
    print("What size puzzle would you like to generate? (e.g. 8, 15, 25)")
    # print("Please INSERT a valid natural number and press ENTER: ")
    puzzle_size = input("Please INSERT a valid natural number and press ENTER: ")
    # puzzle_size = 8
    puzzle_size = int(puzzle_size)
    # Asks the user for number of rows in the puzzle
    # print("Please INSERT the amount of rows needed for the puzzle: ")
    rows = pow(puzzle_size+1, 1/2)
    # print(rows)
    print("\nPlease INSERT a valid puzzle you would like to test row by row.\n" +
          "Note: for the empty space in the puzzle, please INSERT the number 0.")
    # Asks the user to enter the puzzle they would like to test
    initial_state = []
    for row in range(int(rows)):
        puzzle_row = input("\nPlease INSERT values for row " + str(row+1) + " with a space in between each number:\n" +
                           "Press ENTER when you are ready.\n").split()
        for i, val in enumerate(puzzle_row):
            puzzle_row[i] = int(val)
        initial_state.append(puzzle_row)
    # Asks the user the algorithm they would like to use
    print("\nPlease select the algorithm you would like the program to use by entering its correspoding number as shown: \n")
    print("1. Uniform Cost Search\n" +
          "2. A* with the Misplaced Tile heuristic\n" +
          "3. A* with the Manhattan Distance heuristic\n")
    heuristic = input("Select an algortihm and press ENTER: ")
    heuristic = int(heuristic)
    ## The program now have everything it needs to begin
    print("\nThe program has successfully loaded the puzzle and will begin to search for the goal_state...")
    goal_state = []
    puzzle_row = []
    for i in range(int(rows*rows)):
        puzzle_row.append(i+1)
        if len(puzzle_row) == rows:
            goal_state.append(puzzle_row)
            puzzle_row = []
    goal_state[int(rows-1)][int(rows-1)] = 0
    # print(goal_state)
    # goal_state = [[1, 2, 3],
    #               [4, 5, 6],
    #               [7, 8, 0]]
    ##### Depth 0 test case from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
    # initial_state = [[1, 2, 3],
                        # [4, 5, 6],
                        # [7, 8, 0]]
    ##### Depth 2 test case from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
    # initial_state = [[1, 2, 3],
    #                  [4, 5, 6],
    #                  [0, 7, 8]]
    ##### Depth 4 test case from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
    # initial_state = [[1, 2, 3],
    #                  [5, 0, 6],
    #                  [4, 7, 8]]
    ##### Depth 8 test case from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
    # initial_state = [[1, 3, 6],
    #                  [5, 0, 2],
    #                  [4, 7, 8]]
    ##### Depth 12 test case from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
    # initial_state = [[1, 3, 6],
    #                  [5, 0, 7],
    #                  [4, 8, 2]]
    ##### Depth 16 test case from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
    # initial_state = [[1, 6, 7],
    #                  [5, 0, 3],
    #                  [4, 8, 2]]
    ##### Depth 20 test case from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
    # initial_state = [[7, 1, 2],
    #                  [4, 8, 5],
    #                  [6, 3, 0]]
    ##### Depth 24 test case from Professor Eamon Keogh's Project_1_The_Eight_Puzzle_CS_205_2024.pdf handout
    # initial_state = [[0, 7, 2],
    #                  [4, 6, 1],
    #                  [3, 5, 8]]
    queue = None
    if heuristic == 1:
        queue = Queue()
    else:
        queue = PriorityQueue()

    print("\nCalling the function...")

    start_time = time.time()

    problem = Problem(initial_state, goal_state, int(rows), heuristic)

    a_star_search(problem, queue)

    end_time = time.time()

    total_time = end_time - start_time

    print("\nTotal Time: ", total_time)
```

References

[1] Project 1 direction file: https://www.dropbox.com/scl/fi/ntcwot5x8zihrtqot0ysi/Project_1_The_Eight_Puzzle_CS_205_2024.pdf?rlkey=wvosnvls0lkgpiv7h5xsoicz8&e=1&dl=0