Assignment 1 CS170: Introduction to Artificial Intelligence, Dr. Eamonn Keogh

Roshini Rangarajan
Date Feb-8-2024

In completing this assignment I consulted:

- The Project 1 Eight Puzzle Project Write-Up
- The Blind Search and Heuristic Search lecture slides
- To get a better understanding of material: Artificial Intelligence: A Modern Approach (by Stuart Russell and Peter Norvig, 2020, Pearson, ISBN 0134610997)
- To understand how implement heap queue: https://www.geeksforgeeks.org/heap-queue-or-heapq-in-python/
- To understand Uniform Cost Search and A*: https://deniz.co/8-puzzle-solver/
- For generating neighbors function, referenced this article to get a better understanding: https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288
- Referenced this source to understand how to implement the Manhattan Distance Heuristic: https://www.geeksforgeeks.org/8-puzzle-problem-in-ai/
- Referenced this article to understand how to implement misplaced tiles heuristic: https://eitzazsyed.medium.com/number-of-misplaced-tiles-h1-8-puzzle-problem-part-1-f609e3e629d9
- Referenced this source to understand looping through tuples and python tuples:https://www.w3schools.com/python/python_tuples_loop.asp

All important code is original. Unimportant subroutines that are not completely original are…

- All subroutines in **heapq**, to handle the node structures of the states
- All subroutines used from **copy,** to deep copy and modify states

*Note: Please see the references above to see how I completed certain functions or used certain classes.

Outline of this Report:

Not included in report, but wanted to include it here: CS170 Project 1 Spreadsheet

CS170 Assignment 1: The Eight Puzzle

Roshini Rangarajan, Feb-8-2025

1. Introduction

Sliding puzzles, as shown in Figure 1, are a common toy that has many different puzzle combinations that help build problem-solving intuition. The 15-puzzle, as shown in Figure 1, is commonly found in stores, however, the puzzle that is being solved in this project is the 8-puzzle. The 8-puzzle sliding puzzle is divided into a 3 by 3 grid with 8 tiles within the puzzle, similar to the format of the 15 puzzle that has a 4 by 4 grid with 15 tiles. The way this puzzle can be solved is by sliding a tile into the space and in turn leaving a space where another tile can be moved. The puzzle with 8 tiles is labeled 1-8 for the 8-puzzle, and the goal is to arrange the tiles in numerical order from 1 through 8.

Figure 1: This is a common sliding tile puzzle that can be bought today on Amazon, here is where it can be bought: https://www.amazon.com/Schylling-FPZ-15-Puzzle/dp/B000LTLTI2?source=ps-sl-shoppingads-lpcontext&ref_=fplfs&psc=1&smid=AKVINNE8RYVZG&gQT=1

This assignment is the first project in Dr.Eamonn Keogh's Introduction to AI course at the University of California, Riverside during the quarter of Winter 2025. The following write-up details my findings throughout the course of the project's completion. This project explores Uniform Cost Search along with the Misplaced Tile and Manhattan Distance heuristics applied to A*. My language of choice is Python (version 3), and the full code of the project is included.

2. Comparison of Algorithms

The three algorithms implemented in this project are as follows: Uniform Cost Search, A* using the Misplaced Tile heuristic, and A* using the Manhattan Distance heuristic.

2.1 Uniform Cost Search

As stated in the initial project assignment example, **Uniform Cost Search** is A* with the h(n) hardcoded to 0, where the cheapest node is being expanded where the cost is g(n). Hence, as the nodes are expanding there is no given cost for each expansion, and instead, each expansion is considered to have a cost of 1. This is because it takes the same effort to move each tile to the empty space each time. As similarly explained by Russell and Norvig, "*Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier*."

[1] This makes sure that the least cost path is found for the given problem, plus this algorithm is a version of Dijkstra's algorithm which finds the lowest cost similarly. Hence, making this algorithm is a good starting point because it closely relates to how humans would try to solve this problem intuitively, on the first approach when given the 8-puzzle.

2.2 The Misplaced Tile Heuristic

The second algorithm that has been implemented is A* with the **Misplaced Tile Heuristic**. This heuristic compares the current board and finds the number of tiles that have been misplaced in the puzzle. Figure 2 shows an example of how the misplaced tile heuristic works. In a given puzzle, we can see in Figure 2 that there are 3 misplaced tiles, so that would result in g(n) being set to the number of tiles that are not in the correct place compared to the goal state. Thus, making g(n) = 3. Therefore, when applied to the algorithm, the queue expands the nodes with the least cost.

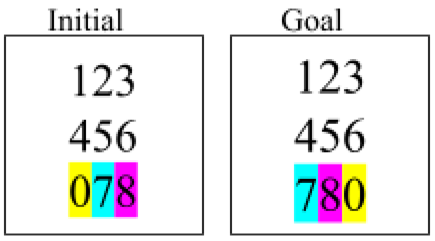


Figure 2: Using puzzle that has a Depth 2 as an example, this how the initial state misplaced tiles compare to the goal state.

2.3 The Manhattan Distance Heuristic

The **Manhattan Distance Heuristic** follows a similar idea to the Misplaced Tile Heuristic because it instead considers not only the number of tiles that are in the wrong spot in comparison to the goal state but also looks at the number of tiles away from the position of the goal state. It does this by getting the absolute values of the distances of all of the tiles in the wrong place and adding them up.

For instance, based on Figure 2, there are two tiles in the incorrect position excluding zero, based on the distance in comparison with goal state positions, g(n) = 2, because they are only one away from the goal.

3. Comparisons of Algorithms on Sample Puzzles

In order to test my algorithm, I implemented 8 different puzzles as shown in Figure 3 in varying depths from class along with the sample puzzles in the project found in the project writeup.

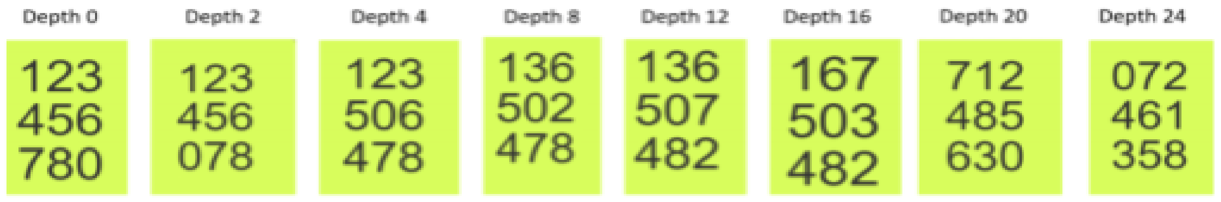


Figure 3: These are the sample puzzles in varying depths from the lecture slides. I implemented them to test my algorithm in different depths.

[1] Artificial Intelligence: A Modern Approach (by Stuart Russell and Peter Norvig, 2020, Pearson, ISBN 0134610997)

These puzzles had varying depths to ensure that the algorithm was working and implemented correctly. Based on the test cases, the simplest puzzle implemented was the puzzle with a depth of 0. The most difficult puzzle given to implement was a puzzle with a depth of 24. In order to compare, how these algorithms perform on different puzzles with different depths, I kept track of the number of nodes expanded and the maximum queue size after the puzzle was solved using the algorithms.

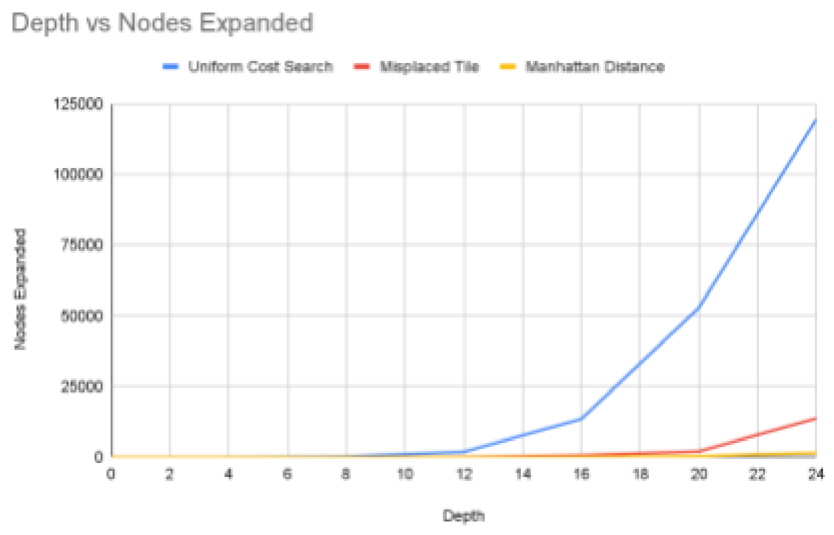


Figure 4: This chart illustrates the comparison between the depth of the puzzle and the number of nodes expanded between the three algorithms.

The difference between the three algorithms based on how they performed on the puzzles is that they performed relatively the same when it came to simpler puzzles such as those for a depth of 2 or depth of 4. However, there was a significant difference in the algorithms when it came to the puzzle starting at a depth of 12. When it comes to the nodes expanded as shown in Figure 4, I noticed that the Uniform Cost Search algorithm performed the worst with the nodes expanded exceeding 100,000 nodes such as for depth 24. This shows that the Uniform Cost Search Algorithm is not that effective and in turn, the run time will be longer as well. However, when looking at the Misplaced Tile Heuristic and Manhattan Distance Heuristic, both of them perform significantly better than the Uniform Cost Search with the Manhattan Distance performing the best of all three. Manhattan Distance for the depth of 24 doesn't exceed 2000 nodes expanded which indicates that it is much faster than Misplaced Tile.

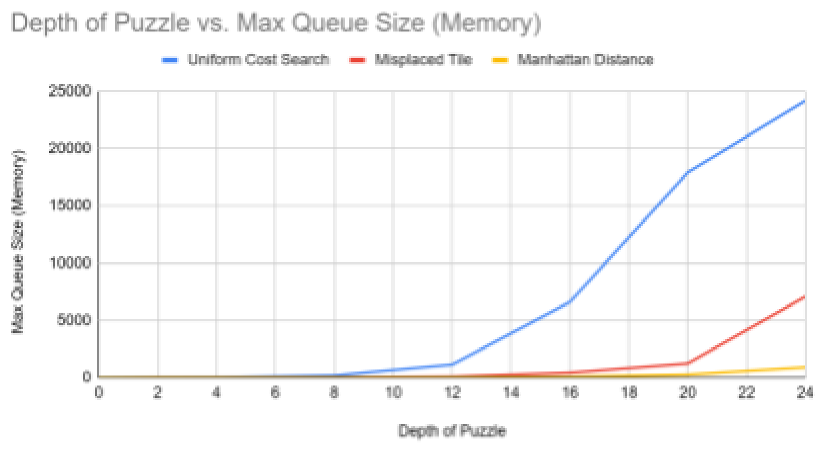


Figure 5: This graph illustrates the comparison between the depth of the puzzle and the max queue size (memory) between all three different algorithms.

When taking a closer look at the depth of the puzzle and the maximum queue size as shown in Figure 5, it shows once again similar results to that of Figure 4. When seeing the memory of each of the algorithms, Uniform Cost Search takes up a lot of memory when storing values in the queue. In comparison, Manhattan Distance takes up the least amount of memory in the queue. Therefore, these are the comparisons between the Uniform Cost Search, Misplaced Tile Heuristic, and Manhattan Distance Heuristic.

4. Additional Examples

To truly stress test the algorithm, I ran two examples of a depth 31 puzzle examples from the lecture slides as shown in Figure 6. For the eight puzzle problem, the diameter of the given problem is 31. Testing on a puzzle with a depth of 31, shows that the program can handle the worst-case scenario and it can perform well on simpler problems too. Additionally, it helps show efficiency as well. See Figure 7 for a comparison of the number of nodes expanded and the maximum queue size reached for all three algorithms between the two puzzle examples from Figure 6.

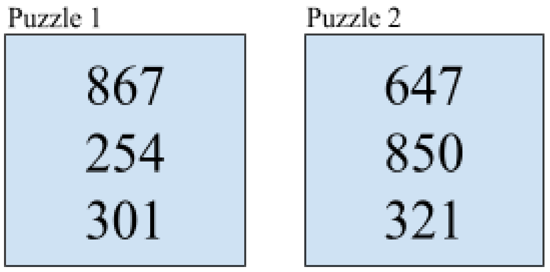
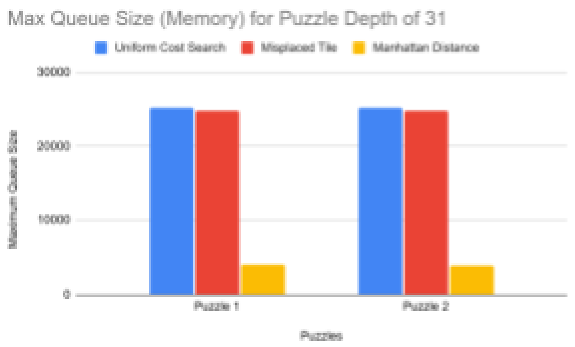


Figure 6: These are puzzle 1 and puzzle 2 examples from the lecture slides. Both of these puzzles have a depth of 31.



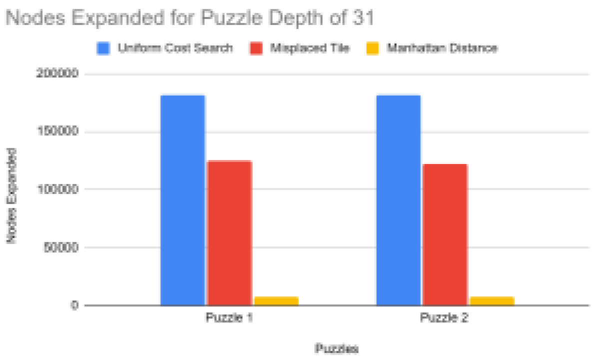


Figure 7: This is the result of my stress test of the puzzles from Figure 6. One chart shows the comparison for the maximum queue size for both puzzles. The other chart shows the nodes expanded for both puzzles.

5. Conclusion

For this project, taking a look at all three algorithms, Uniform Cost Search, Misplaced Tile, and Manhattan Distance a main conclusion can be drawn: if the solution is shallow in depth, there is no difference in what algorithm that is applied to solve the puzzle. However, as the algorithms increase further in depth the most efficient algorithm should be used. Uniform Cost Search essentially becomes Breadth First Search, a blind search algorithm, since the h(n) = 0. Based on the data, for higher-depth problems, Manhattan Distance should be used to solve the puzzle because it uses the least space and also the least amount of nodes are expanded. This shows that using a heuristic is the best approach to solving the eight-puzzle problem, compared to a blind search. However, it also shows that even if using a heuristic is better, it best to find the heuristic that works the most efficiently. This is because between the two heuristics Manhattan Distance and Misplaced Tile, Manhattan Distance performed the best. In conclusion, these are the results of the given project using the three different algorithms.

The following is a traceback of an **easy** puzzle using my program:

```
Welcome to an 8-Puzzle Solver. Type '1' to use a default puzzle,
or '2' to create your own.
2
Enter your puzzle, using zero to represent the blank. Please
enter only valid 8-puzzles.

Enter first row: 1 2 3
Enter second row: 4 0 6
Enter third row: 7 5 8
Select Algorithm. (1) Uniform Cost Search (2)Misplaced Tile
Heuristic (3)Manhattan Distance Heuristic
3
The state to expand with g(n) = 0 and h(n) = 2 is...
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]


The state to expand with g(n) = 1 and h(n) = 1 is...
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]


The state to expand with g(n) = 2 and h(n) = 0 is...
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


Solved!
Depth: 2
Number of nodes expanded: 2
Max queue size: 5
```

The following is a traceback of a **hard** puzzle (depth 16) using my program:

```
Welcome to an 8-Puzzle Solver. Type '1' to use a default
puzzle, or '2' to create your own.
2
Enter your puzzle, using zero to represent the blank.Please
enter only valid 8-puzzles.
Enter first row: 1 6 7
Enter second row: 5 0 3
Enter third row: 4 8 2

Select Algorithm. (1) Uniform Cost Search (2)Misplaced Tile
Heuristic (3)Manhattan Distance Heuristic
3

The state to expand with g(n) = 0 and h(n) = 12 is...
[1, 6, 7]
[5, 0, 3]
[4, 8, 2]

::::   # Here some lines of output are deleted to save space

The state to expand with g(n) = 4 and h(n) = 12 is...
[5, 1, 7]
[4, 6, 3]
[0, 8, 2]

The state to expand with g(n) = 16 and h(n) = 0 is...
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Solved!
Depth: 16
Number of nodes expanded: 96
Max queue size: 64
```

My code can be found at: https://github.com/roshinira123/The-Eight-Puzzle/

```
import heapq
import copy


#Puzzles to be inputted from the project psuedocode
trivial = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
veryEasy = [[1, 2, 3], [4, 5, 6], [7, 0, 8]]
easy = [[1, 2, 0], [4, 5, 3], [7, 8, 6]]
doable = [[0, 1, 2], [4, 5, 3], [7, 8, 6]]
oh_boy = [[8, 7, 1], [6, 0, 2], [5, 4, 3]]

#Puzzles of different depths from lecture slides in class for testing
depth_2 = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
depth_4 = [[1, 2, 3], [5, 0, 6], [4, 7, 8]]
depth_8 = [[1, 3, 6], [5, 0, 2], [4, 7, 8]]
depth_12 = [[1, 3, 6], [5, 0, 7], [4, 8, 2]]
depth_16 = [[1, 6, 7], [5, 0, 3], [4, 8, 2]]
depth_20 = [[7, 1, 2], [4, 8, 5], [6, 3, 0]]
depth_24 = [[0, 7, 2], [4, 6, 1], [3, 5, 8]]

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


class TreeNode:
    def __init__(self, puzzle, cost, heuristic, parent=None):
        self.puzzle = puzzle
        self.cost = cost
        self.heuristic = heuristic
        self.total_cost = cost + heuristic  # f(n) = g(n) + h(n)
        self.parent = parent


    def __lt__(self, other): #finds the least cost
        return self.total_cost < other.total_cost


    def solved(self):
        return self.puzzle == eight_goal_state

    def board_to_tuple(self):
        return tuple(tuple(row) for row in self.puzzle)

    def generate_neighbors(self):
        moves = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  #move up, move
down, move left, move right
```

```
        zeroPosrow, zeroPoscol = None, None
        #gets the position of where the zero is in the board
        for r in range(3):
            for c in range(3):
                if self.puzzle[r][c] == 0:
                    zeroPosrow, zeroPoscol = r, c
                    break
            if zeroPosrow is not None:
                break

        #checks if the zero can move in that way
        for rowOffset, columnOffset in directions:
            newRow= zeroPosrow + rowOffset
            newCol = zeroPoscol + columnOffset
            if 0 <= newRow < 3 and 0 <= newCol < 3: #checks in the newRow
and newCol are in the boundaries
                new_puzzle = copy.deepcopy(self.puzzle)
                new_puzzle[zeroPosrow][zeroPoscol],
new_puzzle[newRow][newCol] = new_puzzle[newRow][newCol],
new_puzzle[zeroPosrow][zeroPoscol] #Swaps tiles

                moves.append(new_puzzle) #adds the newpuzzle to the moves

        return moves

def main():
    puzzle_mode = input("Welcome to an 8-Puzzle Solver. Type '1' to use a
default puzzle, or '2' to create your own." + "\n")

    if puzzle_mode == "1":
        select_and_init_algorithm(init_default_puzzle_mode())
    if puzzle_mode == "2":
        print("Enter your puzzle, using zero to reprsent the blank." +
"Please enter only valid 8-puzzles." + "\n")
        puzzle_r1 = input("Enter first row: ")
        puzzle_r2 = input("Enter second row: ")
        puzzle_r3 = input("Enter third row: ")

        puzzle_r1 = puzzle_r1.split()
        puzzle_r2 = puzzle_r2.split()
        puzzle_r3 = puzzle_r3.split()


        for i in range(3):
            puzzle_r1[i] = int (puzzle_r1[i])
            puzzle_r2[i] = int (puzzle_r2[i])
            puzzle_r3[i] = int (puzzle_r3[i])

        userPuzzle = [puzzle_r1, puzzle_r2, puzzle_r3]
        select_and_init_algorithm(userPuzzle)

    return

def init_default_puzzle_mode():
    selectedDiff = input("Choose the difficulty for the puzzle on scale of
0-4 or 5-11 to get puzzles with different depths" + "\n")
    if selectedDiff == "0":
        print("Trivial selected")
        printPuzzle(trivial)
        return trivial
    if selectedDiff == "1":
        print("Very Easy selected")
        printPuzzle(veryEasy)
        return veryEasy
    if selectedDiff == "2":
        print("Easy selected")
        printPuzzle(easy)
        return easy
    if selectedDiff == "3":
        print("Doable selected")
        printPuzzle(doable)
        return doable
    if selectedDiff == "4":
        print("Oh Boy selected")
        printPuzzle(oh_boy)
        return oh_boy
    if selectedDiff == "5": #added these as cases so it would be easier
for me to test the different depths and record values
        print("Depth 2 selected")
        printPuzzle(depth_2)
        return depth_2
    if selectedDiff == "6":
        print("Depth 4 selected")
        printPuzzle(depth_4)
        return depth_4
    if selectedDiff == "7":
        print("Depth 8 selected")
        printPuzzle(depth_8)
        return depth_8
    if selectedDiff == "8":
        print("Depth 12 selected")
        printPuzzle(depth_12)
        return depth_12
    if selectedDiff == "9":
        print("Depth 16 selected")
        printPuzzle(depth_16)
        return depth_16
    if selectedDiff == "10":
        print("Depth 20 selected")
        printPuzzle(depth_20)
        return depth_20
    if selectedDiff == "11":
        print("Depth 24 selected")
        printPuzzle(depth_24)
        return depth_24
    #if selectedDiff == "5":
       # print("Trivial selected")
        #return impossible

def printPuzzle(puzzle): #general print function
    for i in range(0,3):
        print(puzzle[i])
    print("\n")


def printPath(node): #prints path of best state using parent node
    path = []
    while node:
        path.append(node.puzzle)
        node = node.parent
    for puzzle in reversed(path):
        printPuzzle(puzzle)

def select_and_init_algorithm(puzzle):
    algorithm = input ("Select Algorithm. (1) Uniform Cost Search
(2)Misplaced Tile Heuristic (3)Manhattan Distance Heuristic " + "\n")

    if algorithm == "1":
        general_search_algorithm(puzzle, uniform_cost_search)
    if algorithm == "2":
        general_search_algorithm(puzzle, misplaced_tile_heuristic)
    if algorithm == "3":
        general_search_algorithm(puzzle, manhattan_distance_heuristic)

#implemented general search algorithm, then changed the heuristics for
ucs, mth, and mdh. I implemented it this way becuase it was easier for me
to understand
def general_search_algorithm(startPuzzle, heuristic):
    startNode = TreeNode(startPuzzle, 0, heuristic(startPuzzle))
    workingQueue = []
    repeatedStates = {}
    heapq.heappush(workingQueue, startNode)

    nodesExpanded = 0
    maxQueueSize = 1
    repeatedStates[startNode.board_to_tuple()] = startNode.cost


    while len(workingQueue) > 0:
        maxQueueSize = max(len(workingQueue), maxQueueSize)
        node_from_queue = heapq.heappop(workingQueue)

        #Debug statments:
        #repeatedStates[node_from_queue.board_to_tuple()] = "This can be
anything"
        #nodesExpanded += 1
        #print(f"Expanding node: {node_from_queue.puzzle}")


        print(f"The state to expand with g(n) = {node_from_queue.cost} and
h(n) = {node_from_queue.heuristic} is...")
        printPuzzle(node_from_queue.puzzle)


        if node_from_queue.solved():
            print("Solved!")
            #while stackPrint:
                #printPuzzle(stackPrint.pop())
            print("Depth:", node_from_queue.cost)
            print("Number of nodes expanded:", nodesExpanded)
            print("Max queue size:", maxQueueSize)
            #print("Best State Path Below:") #debug statement but can
uncomment to see best path
            #printPath(node_from_queue)
            return node_from_queue

        nodesExpanded += 1

        for neighbor in node_from_queue.generate_neighbors():
                #Creates node and updated with cost and h(n) value, then
converts to tuple for comparison
                child = TreeNode(neighbor, node_from_queue.cost + 1,
heuristic(neighbor), node_from_queue)
                child_tuple = child.board_to_tuple()
                #Checks if has been visited before and less than previous
cost
```

```
                if child_tuple not in repeatedStates or
repeatedStates[child_tuple] > child.cost:
                    #Adds to queue and updates repeatedStates
                    heapq.heappush(workingQueue, child)
                    repeatedStates[child_tuple] = child.cost

    print("Failure: No solution found")
    return None

    #print("Number of nodes expanded:", nodesExpanded) Debug statement
    #print("Max queue size:", maxQueueSize) Debug statement


def uniform_cost_search(puzzle):
    return 0; #based on project writeup UCS is "A* with h(n) hardcoded to
0" and it uses the general search algo hence returning 0

def misplaced_tile_heuristic(puzzle): #need to compare to goal state then
check how many tiles misplaced
    misplacedTiles = 0
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] != 0 and puzzle[i][j] !=
eight_goal_state[i][j]:
                misplacedTiles += 1

    return misplacedTiles

def manhattan_distance_heuristic(puzzle): #calculates distance of each
tile from its goal state using absolute value
    manDist = 0

    for i in range(3):
        for j in range(3):
            tile = puzzle[i][j]
            if tile != 0:
                goalRow = (tile -1) //3 # converted the divmod to this
because it wasn't working
                goalCol = (tile -1) % 3
                manDist += abs(i - goalRow) + abs(j - goalCol)

    return manDist

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