Assignment 1 CS205: Introduction to Artificial Intelligence, Dr. Eamonn Keogh

Marcus Martin

SID 862026991

Email mmart149@ucr.edu

Date May-12-2021

In completing this assignment I consulted:

- Blind Search part 1, Blind Search part 2, Heuristic Search slides and videos from lecture
- Eight Puzzle briefing and debriefing project 1 videos from lecture
- Eight Puzzle Project Directions and Example Assignment
- Various websites for Python 3 documentation that are shown in code comments

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

- Heapq for node structure
- Copy (deepcopy) for expanding nodes
- Time and Math for keeping track of the run time of the program
- The truncate() function used to truncate the run time of the program to four decimal places, which was from: https://www.delftstack.com/howto/python/python-truncate-float-python/
- Matplotlib.pyplot for charting data

Outline of this report:

CS205 Assignment 1 Eight Puzzle Project

Marcus Martin, SID 862026991 May-12-2021

Introduction

The Eight-Puzzle problem is a 3x3 grid with a single unoccupied space. The rest of the spaces are filled with unique numbers ranging from 1-8. The objective for the Eight Puzzle problem is to arrange the tiles in a specific order. In the case of the Eight Puzzle project, we order the numbers from least to greatest by row. So, the solution to the Eight Puzzle problem (with 0 representing the blank space) would have the first row contain the numbers 1, 2, 3, the second row would contain 4, 5, 6, and the third row would contain 7, 8, 0 as shown in Figure 1. To get to the solution, a move would consist of moving a tile into the blank space. The tile would have to be next to the blank space to be a valid move. By repeating this process, it is possible to reach the Eight Puzzle solution.

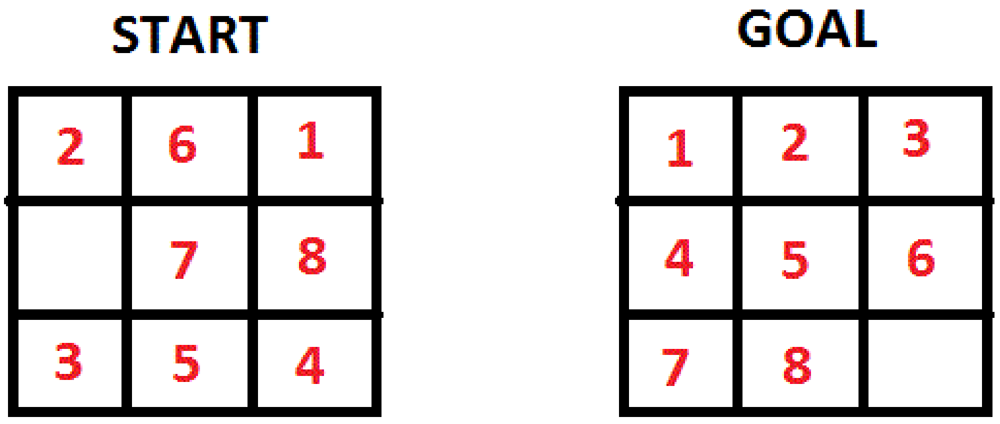


Figure 1[1]: An example of a start and goal state of the Eight Puzzle problem

The following project uses Python 3 to implement Uniform Cost Search, Misplaced Tile, and Manhattan Distance heuristics with A* Search for Dr. Eamonn Keogh's Eight Puzzle Project in his class CS205 Introduction to Artificial Intelligence. First, we will explore the process of A* Search and the three heuristics mentioned above. Next, we will explore how each heuristic affects the number of expanded nodes, the max queue size, and the run time for varying depths and examples of the Eight Puzzle problem. Finally, we will present a trace of an easy and hard Eight Puzzle as well as the code for the project.

Algorithms

A* Search

A* Search first starts at a starting node and will then start searching for the cheapest cost to a goal node. From the current node, A* Search gets the cost to reach neighboring nodes (g(n)) and gets their

[1] https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288

estimated cost to reach the goal state (h(n)), and then proceeds to travel to the neighboring node (expand) with the lowest cost (f(n)) as shown in Figure 2. The cost for a node consists of the cost from the starting node to the current node and the estimated cost from the current node to the goal node (f(n) = g(n) + h(n)). Eventually, A* Search will find the cheapest path from the start node to the goal node. A* Search is both optimal and complete, meaning that A* Search will always find the best solution if a solution exists for the problem.

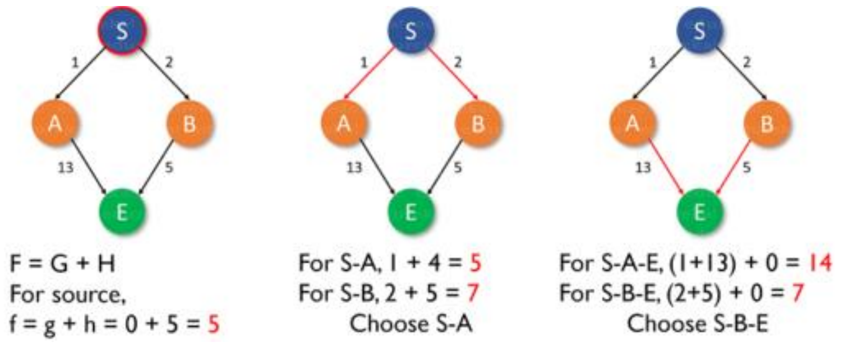


Figure 2[2]: An example of A* Search being used on a graph

For the Eight Puzzle problem, A* Search will start at a given Eight Puzzle problem and will proceed to explore many variations of the puzzle until it finds the goal state of the puzzle as shown in Figure 3. The path that A* Search finds will be the cheapest solution. In A* Search, the cost for g(n) will be 1. To estimate h(n), we used Uniform Cost Search, Misplaced Tile, and Manhattan Distance heuristics.

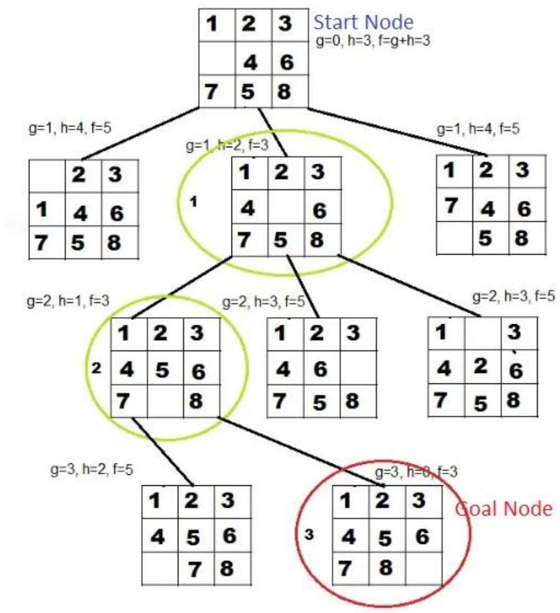


Figure 3[3]: An example of the process of solving an Eight Puzzle problem using A* Search

[2] https://www.edureka.co/blog/a-search-algorithm/
[3] https://www.goeduhub.com/8062/8-puzzle-problem-in-ai-artificial-intelligence

Uniform Cost Search

For the Eight Puzzle Project, Uniform Cost Search has h(n) = 0. When A* Search is performed with Uniform Cost Search, A* Search will only consider g(n) in its calculation of the cost since f(n) = g(n) + h(n).

Misplaced Tile

The Misplaced Tile heuristic compares the current puzzle tiles to the goal state tiles and counts the number of differences between them, not including the blank space. h(n) will be equal to the number of differences. Misplaced Tile enables A* Search to perform better than Uniform Cost Search since it includes an estimate that influences the overall cost. Compared to Uniform Cost Search in terms of performance, Misplaced Tile will expand less nodes and run faster since it decreases the number of possible expansions A* Search needs to explore.

Manhattan Distance

The Manhattan Distance heuristic calculates the distances between the current puzzle tiles and the goal state tiles, not including the blank space. h(n) will be equal to the distance. Manhattan Distance improves upon Misplaced Tile because it also considers the distance from a current puzzle tile to its corresponding goal state tile. Manhattan Distance enables A* Search to perform better than Misplaced Tile because Manhattan Distance considers the differences between the current puzzle tiles and the goal state tiles, and the distance from a current puzzle tile to its goal state tile.

Comparison of Algorithms

Figure 4 shows the test cases that Dr. Keogh provided for the Eight Puzzle problem. The puzzle difficulty ranges from easy (depth = 0) to hard (depth = 24).

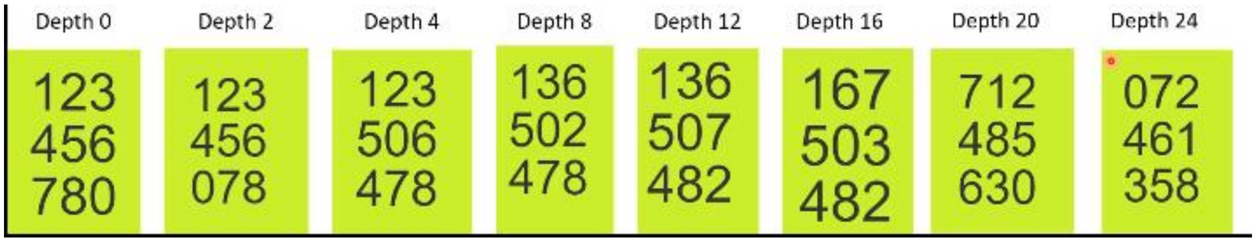
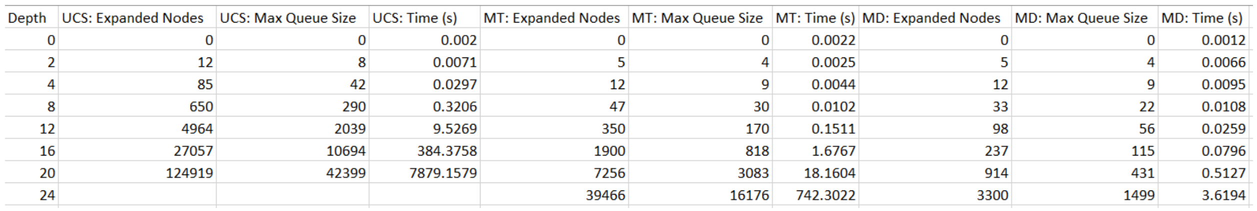


Figure 4: Provided Eight Puzzle Examples from Dr. Keogh

When we ran the algorithms on these puzzles, we discovered that the results were very similar for easy puzzles but were very different for harder puzzles due to the heuristic. The factors that we kept track of was the number of nodes expanded, the max queue size, the solution depth, and the run time for a puzzle as shown in Figure 5.

As a description of these factors, the number of nodes expanded corresponds to the nodes that are traveled to. The max queue size corresponds to the maximum number of nodes that are in the queue while the algorithm is being performed. The run time is the amount of time in seconds that the algorithm takes while running. The solution depth is the depth of the solution for a puzzle.



Depth	UCS: Expanded Nodes	UCS: Max Queue Size	UCS: Time (s)	MT: Expanded Nodes	MT: Max Queue Size	MT: Time (s)	MD: Expanded Nodes	MD: Max Queue Size	MD: Time (s)
0	0	0	0.002	0	0	0.0022	0	0	0.0012
2	12	8	0.0071	5	4	0.0025	5	4	0.0066
4	85	42	0.0297	12	9	0.0044	12	9	0.0095
8	650	290	0.3206	47	30	0.0102	33	22	0.0108
12	4964	2039	9.5269	350	170	0.1511	98	56	0.0259
16	27057	10694	384.3758	1900	818	1.6767	237	115	0.0796
20	124919	42399	7879.1579	7256	3083	18.1604	914	431	0.5127
24				39466	16176	742.3022	3300	1499	3.6194

Figure 5: A table that shows the corresponding numerical values for Depth, Expanded Nodes, Max Queue Size, and Time. Note that UCS = Uniform Cost Search, MT = Misplaced Tile, MD = Manhattan Distance, (s) = seconds. Also note that UCS depth = 24 is empty

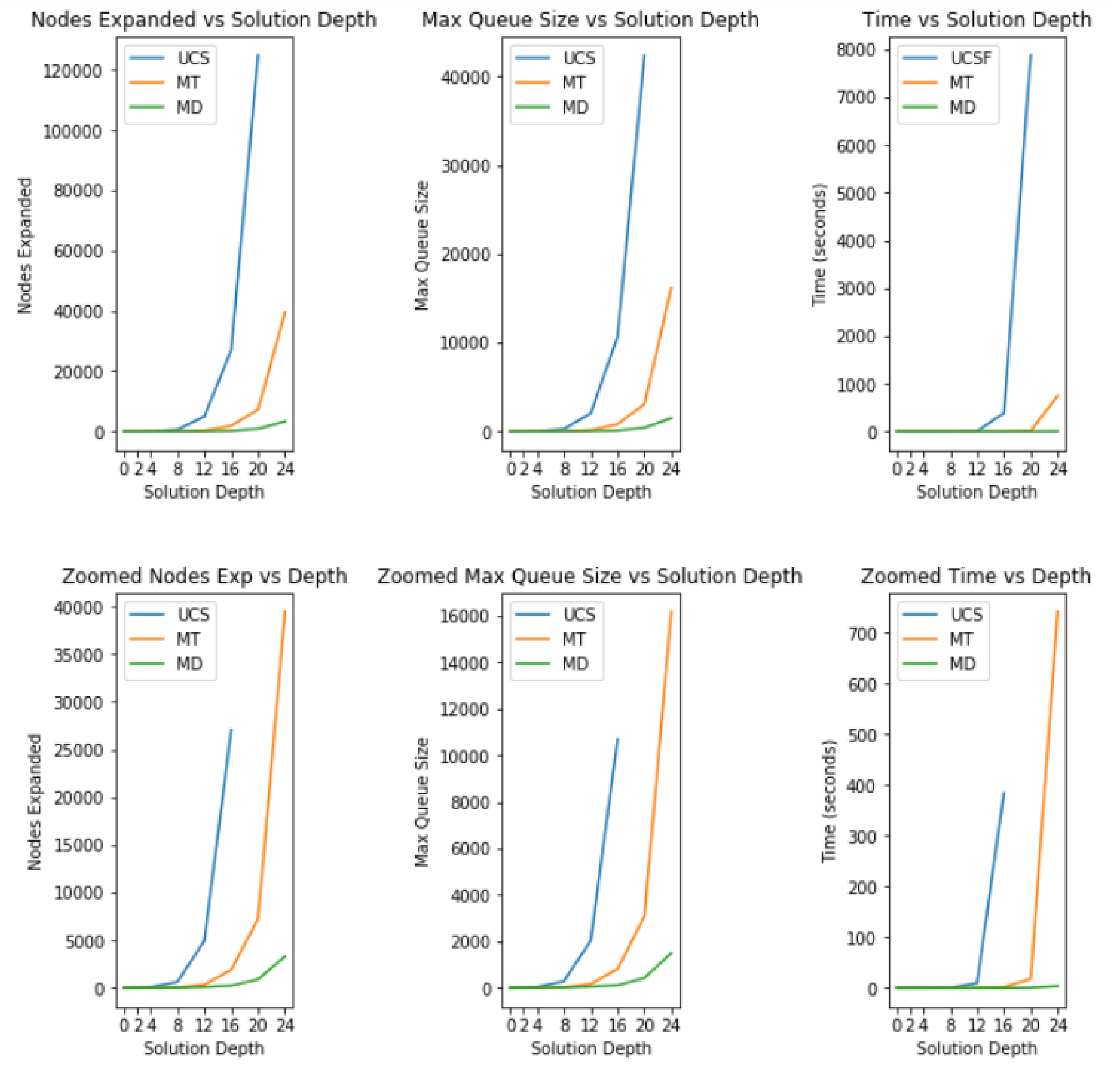


Figure 6: The number of nodes expanded, max queue size, and time for Uniform Cost Search, Misplaced Tile, and Manhattan Distance. Note that UCS = Uniform Cost Search, MT = Misplaced Tile, MD = Manhattan Distance. Also note that the first row of graphs includes the Uniform Cost Search depth = 20 result, and the second row of graphs do not include the Uniform Cost Search depth = 20 result (for better readability of the graphs).

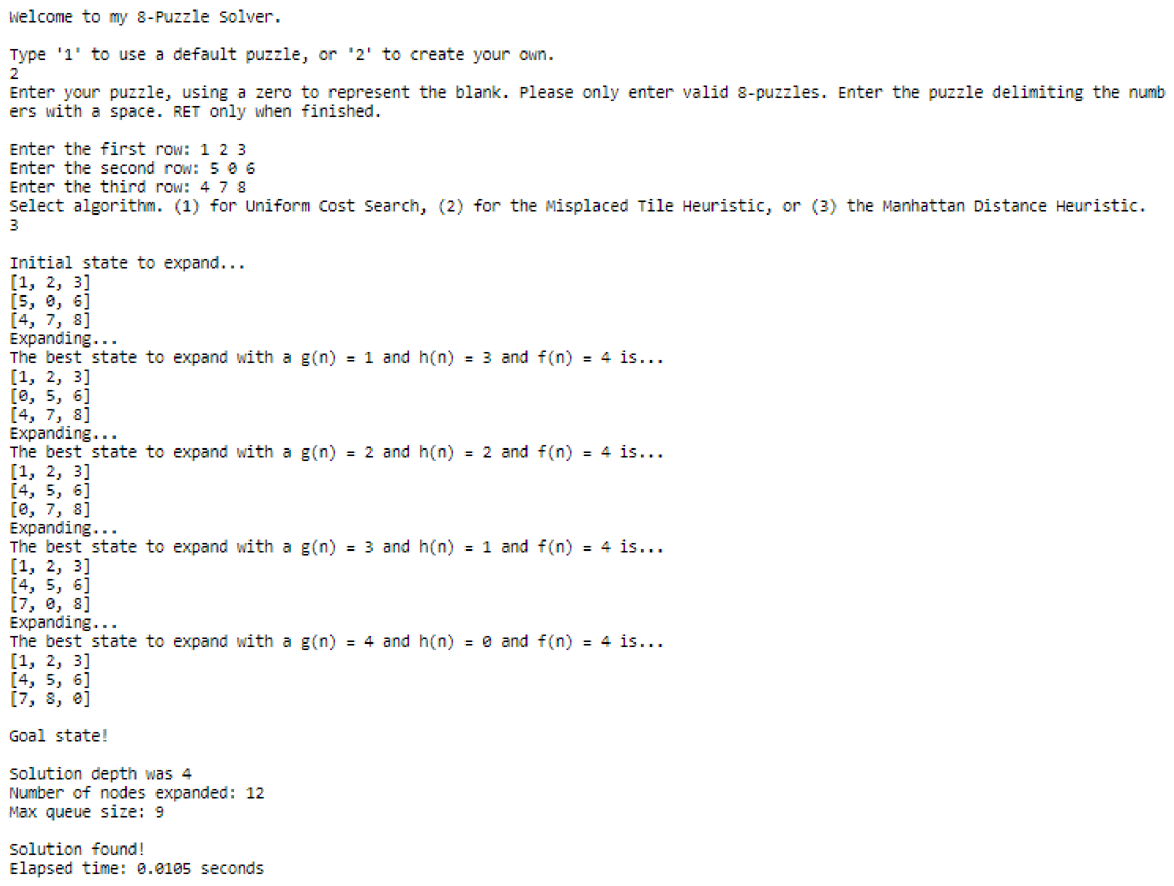
As we can see from Figure 5 and Figure 6, the number of nodes expanded from depth 0 to 4 are very similar, depth 8 is roughly similar, and depth 12 to 24 are very different. Although the result for Uniform Search Cost with depth 24 was not able to be calculated due to the long run time of the algorithm, we can see from the graphs that Uniform Cost Search would likely not perform as well as the other two heuristics at that depth. The effectiveness of the heuristics can clearly be seen as the Uniform Cost Search expanded many more nodes, had a greater max queue size, and had a longer run time on harder puzzles than Misplaced Tile or Manhattan Distance. As we described earlier, Misplaced Tile did perform better than Uniform Cost Search, and Manhattan Distance performed better than Misplaced Tile. Estimating the cost to the goal node clearly influenced the performance of A* Search.

Conclusion

The time and space complexity of A* Search can be greatly affected by heuristics. As seen from the above graphs, A* Search with Uniform Cost Search had the worst time and space complexity, A* Search with Misplaced Tile had the second worst time and space complexity, and A* Search with Manhattan Distance had the best time and space complexity. Overall, heuristics have the potential to greatly improve the time and space complexity of a search algorithm, and different heuristics can have varying results of improving a search algorithm.

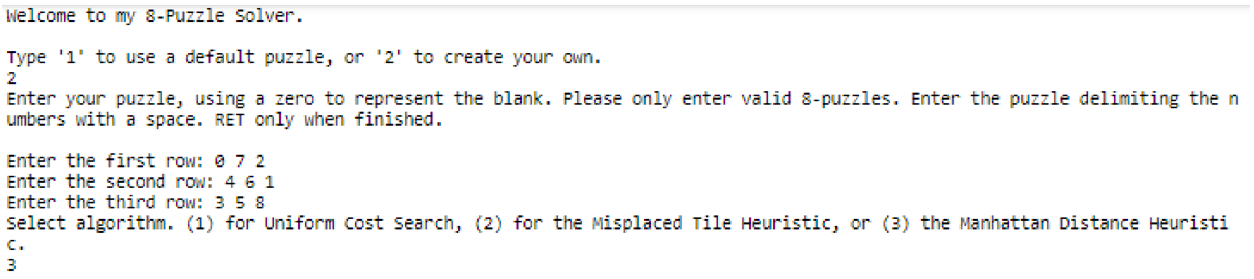
Traces

Trace (Easy)

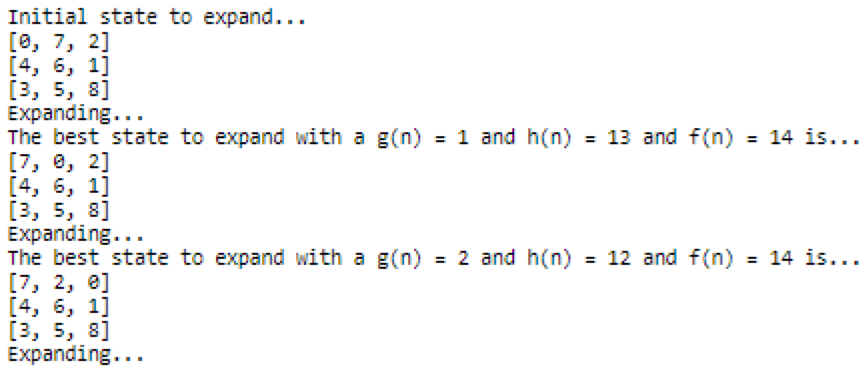


```
Welcome to my 8-Puzzle Solver.

Type '1' to use a default puzzle, or '2' to create your own.
2
Enter your puzzle, using a zero to represent the blank. Please only enter valid 8-puzzles. Enter the puzzle delimiting the numb
ers with a space. RET only when finished.

Enter the first row: 1 2 3
Enter the second row: 5 0 6
Enter the third row: 4 7 8
Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.
3

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

Goal state!

Solution depth was 4
Number of nodes expanded: 12
Max queue size: 9

Solution found!
Elapsed time: 0.0105 seconds
```

Trace (Hard, only included the first and last couple of trace for brevity)

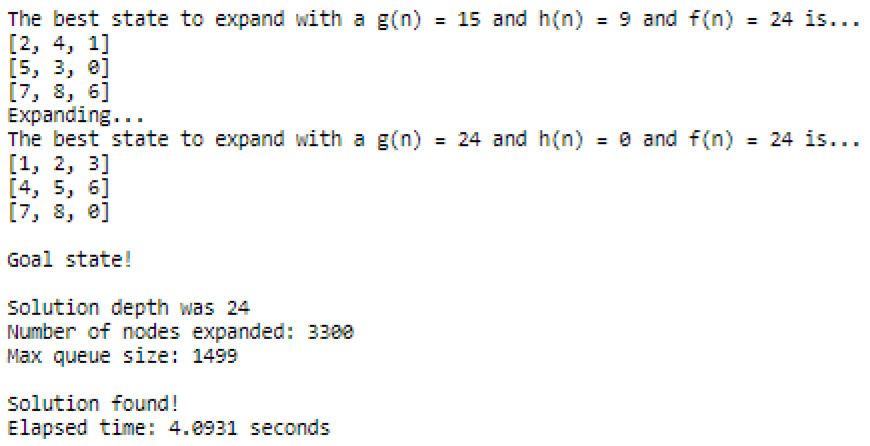


```
Welcome to my 8-Puzzle Solver.

Type '1' to use a default puzzle, or '2' to create your own.
2
Enter your puzzle, using a zero to represent the blank. Please only enter valid 8-puzzles. Enter the puzzle delimiting the n
umbers with a space. RET only when finished.

Enter the first row: 0 7 2
Enter the second row: 4 6 1
Enter the third row: 3 5 8
Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristi
c.
3
```



```
Initial state to expand...
[0, 7, 2]
[4, 6, 1]
[3, 5, 8]
Expanding...
The best state to expand with a g(n) = 1 and h(n) = 13 and f(n) = 14 is...
[7, 0, 2]
[4, 6, 1]
[3, 5, 8]
Expanding...
The best state to expand with a g(n) = 2 and h(n) = 12 and f(n) = 14 is...
[7, 2, 0]
[4, 6, 1]
[3, 5, 8]
Expanding...
```

…



```
The best state to expand with a g(n) = 15 and h(n) = 9 and f(n) = 24 is...
[2, 4, 1]
[5, 3, 0]
[7, 8, 6]
Expanding...
The best state to expand with a g(n) = 24 and h(n) = 0 and f(n) = 24 is...
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Goal state!

Solution depth was 24
Number of nodes expanded: 3300
Max queue size: 1499

Solution found!
Elapsed time: 4.0931 seconds
```

Code

Project Code Description

I used Python 3 and used the heapq, copy, time, and math libraries in the project. A* Search was performed with the Uniform Cost Search, Misplaced Tile, and Manhattan Distance heuristics. First, the user has the option to choose to use a default puzzle or enter their own puzzle, and then select the heuristic to be used for the puzzle. A priority queue is used to store the nodes (puzzles), and then the expansion of nodes is based on the four possible moves that can be performed. If a move is possible, the move is performed, and the node is added to a “visited” node list. The number of expanded nodes and max queue size are also changed accordingly, and the trace of the process is outputted. This process continues until the node to be expanded is found to be the goal state, and then the number of expanded nodes, the max queue size, and the run time is outputted to the user. Their corresponding graphs are then displayed. Since it takes a very long time to run Uniform Cost Search on puzzles with depth = 24, I did not display the point on the graph.

Project Code

The GitHub repository URL for this project is: https://github.com/marcusamartin/CS205_8Puzzle

Below is my code for the project:

```
# Project Implementation

import heapq
import copy
from copy import deepcopy
import time   # track elapsed time
import math   # for truncate elapsed time

class Node():
  def __init__(self, puzzle):
    self.puzzle = puzzle
    self.gn = 0
    self.hn = 0
    self.fn = 0
    self.goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

  # needed for heapq.heappush comparison https://stackoverflow.com/questio
ns/7803121/in-python-heapq-heapify-doesnt-take-cmp-or-key-functions-as-
arguments-like-sor
  def __lt__(self, other):
      return self.fn < other.fn

# globals
expandedNodes = 0
maxQueueSize = 0
visitedPuzzles = []
start = 0

def main():
    # reset globals
    global expandedNodes
    global maxQueueSize
    global visitedPuzzles
    global start
    expandedNodes = 0
    maxQueueSize = 0
    visitedPuzzles = []
    start = 0


    print("Welcome to my 8-Puzzle Solver.\n")
    puzzleChoice = int(input("Type '1' to use a default puzzle, or '2' to
create your own.\n"))
    puzzle = []
    if (puzzleChoice == 1):
        puzzle = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]   # default puzzle is g
oal state
    elif (puzzleChoice == 2):
        print("Enter your puzzle, using a zero to represent the blank. Ple
ase only enter valid 8-
puzzles. Enter the puzzle delimiting the numbers with a space. RET only wh
en finished.\n")
        firstRow = input("Enter the first row: ").split()   # https://www.
w3schools.com/python/ref_string_split.asp
        secondRow = input("Enter the second row: ").split()
        thirdRow = input("Enter the third row: ").split()

        # convert string to int https://www.kite.com/python/answers/how-
to-convert-all-elements-of-a-list-to-int-in-
python#:~:text=Use%20int()%20to%20convert,x)%20applied%20to%20all%20elemen
ts.
        firstRow = [int(value) for value in firstRow]
        secondRow = [int(value) for value in secondRow]
        thirdRow = [int(value) for value in thirdRow]

        puzzle = [firstRow, secondRow, thirdRow]

    algorithm = int(input("Select algorithm. (1) for Uniform Cost Search,
(2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuris
tic.\n"))

    puzzle = Node(puzzle)   # initialize puzzle

    start = time.clock()
    success = generalsearch(puzzle, algorithm)

    print()   # for new line separation
    if (success == 1):
      print("Solution found!")
    elif (success == 0):
      print("No solution found!")
    elif (success == 15):
      print("No solution found due to taking too long (more than 15 minute
s)!")

    end = time.clock()
    print("Elapsed time: " + str(truncate(end - start, 4)) + " seconds")

def generalsearch(puzzle, algorithm):
  success = 0
  global maxQueueSize   # able to use and change global value
  nodes = []   # initial empty list
  heapq.heapify(nodes)   # https://www.geeksforgeeks.org/heap-queue-or-
heapq-in-python/
  heapq.heappush(nodes, puzzle)   # push puzzle onto list
  initialExpansion = True

  while (True):
    if (len(nodes) == 0):
      success = 0
      break
    if (truncate((time.clock() - start), 4) > 900):   # fail if elapsed ti
me greater than 15 minutes
      success = 15
      break
    node = heapq.heappop(nodes)
    if (node.puzzle == node.goal):
      print("\nGoal state!\n")
      print("Solution depth was " + str(node.gn))
      print("Number of nodes expanded: " + str(expandedNodes))
      print("Max queue size: " + str(maxQueueSize))
      success = 1
      break

    if (initialExpansion):
      print()   # for new line separation
      print("Initial state to expand...")
      outputPuzzle(node)
      initialExpansion = False

    print("Expanding...")
    nodes = expand(nodes, node, algorithm)

  return success
```

```
def expand(nodes, node, algorithm):
  moves = ["up", "left", "right", "down"]
  visited = False
  for curMove in moves:
    expandNode = copy.deepcopy(node) # deepcopy so changes are not made to
 original node https://www.programiz.com/python-programming/shallow-deep-
copy
    i, j = getZeroIndex(expandNode)
    validMove = isValidMove(i, j, curMove)
    if (validMove):
      expandedNode = move(expandNode, i, j, curMove, node, algorithm)
      visited = checkVisited(nodes, expandedNode)
      if (not visited):
        global expandedNodes   # able to change global value
        expandedNodes += 1

  global maxQueueSize   # able to change global value
  if (maxQueueSize < len(nodes)):
      maxQueueSize = len(nodes)

  cheapestSolution = nodes[0]
  print("The best state to expand with a g(n) = " + str(cheapestSolution.g
n) + " and h(n) = " + str(cheapestSolution.hn) + " and f(n) = " + str(chea
pestSolution.fn) + " is...")
  outputPuzzle(cheapestSolution)

  return nodes


# used code for truncating from https://www.delftstack.com/howto/python/py
thon-truncate-float-python/
def truncate(number, decimal):
    integer = int(number * (10 ** decimal)) / (10 ** decimal)
    return float(integer)

def outputPuzzle(node):
  for i in node.puzzle:
    print(i)

def getZeroIndex(node):
  index = []
  for i, values in enumerate(node.puzzle):   # https://treyhunner.com/2016
/04/how-to-loop-with-indexes-in-python/
    for j, value in enumerate(values):
      if (value == 0):
        index.append(i)
        index.append(j)

  return index

# values based on 8-puzzle dimensions
def isValidMove(i, j, move):
  isValid = True
  if (move == "up" and i == 0):
    isValid = False
  elif (move == "left" and j == 0):
    isValid = False
  elif (move == "right" and j == 2):
    isValid = False
  elif (move == "down" and i == 2):
    isValid = False

  return isValid

def move(expandNode, i, j, direction, node, algorithm):
  tempVal = expandNode.puzzle[i][j]

  if (direction == "up"):
    expandNode.puzzle[i][j] = expandNode.puzzle[i - 1][j]
    expandNode.puzzle[i - 1][j] = tempVal
  elif (direction == "left"):
    expandNode.puzzle[i][j] = expandNode.puzzle[i][j - 1]
    expandNode.puzzle[i][j - 1] = tempVal
  elif (direction == "right"):
    expandNode.puzzle[i][j] = expandNode.puzzle[i][j + 1]
    expandNode.puzzle[i][j + 1] = tempVal
  elif (direction == "down"):
    expandNode.puzzle[i][j] = expandNode.puzzle[i + 1][j]
    expandNode.puzzle[i + 1][j] = tempVal

  node = performAlgo(node, expandNode, algorithm)

  return expandNode

def performAlgo(node, expandNode, algorithm):
  if (algorithm == 1):
    hn = uniformCost()
  elif (algorithm == 2):
    hn = misplacedTile(expandNode)
  elif (algorithm == 3):
    hn = manhattanDistance(expandNode)
  expandNode.gn = node.gn + 1
  expandNode.hn = hn
  expandNode.fn = expandNode.gn + expandNode.hn

  return expandNode

def checkVisited(nodes, node):
  visited = False
  for i in visitedPuzzles:
    if (i == (node.puzzle, node.gn, node.hn, node.fn)):
      visited = True
  if (not visited):
      heapq.heappush(nodes, node)
      visitedPuzzles.append((node.puzzle, node.gn, node.hn, node.fn))

  return visited


def uniformCost():
  return 0

def misplacedTile(node):
  result = 0
  for i, values in enumerate(node.puzzle):   # https://treyhunner.com/2016
/04/how-to-loop-with-indexes-in-python/
    for j, value in enumerate(values):
        if (value != node.goal[i][j]):
          if (value != 0):
            result += 1

  return result

def manhattanDistance(node):   # https://xlinux.nist.gov/dads/HTML/manhatt
anDistance.html#:~:text=Definition%3A%20The%20distance%20between%20two,y1%
20%2D%20y2%7C.
  result = 0
  for i, values in enumerate(node.puzzle):
    for j, value in enumerate(values):
        k, l = getGoalIndex(node, value)
        if (value != 0):
            manhattanFormula = (abs(k - i) + abs(l - j))
            result += manhattanFormula
  return result

def getGoalIndex(node, number):
  index = []
  for i, values in enumerate(node.goal):
    for j, value in enumerate(values):
      if (value == number):
        index.append(i)
        index.append(j)

  return index


main()

# Chart Data

import matplotlib.pyplot as plt   # https://matplotlib.org/stable/api/_as_
gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot


depths = [0, 2, 4, 8, 12, 16, 20, 24]

uniformCostExpandedNodes = [0, 12, 85, 650, 4964, 27057, 124919, 0]   # la
st result for depth 24 took too long to be calculated
misplacedTileExpandedNodes = [0, 5, 12, 47, 350, 1900, 7256, 39466]
manhattanDistanceExpandedNodes = [0, 5, 12, 33, 98, 237, 914, 3300]

uniformCostMaxQueueSize = [0, 8, 42, 290, 2039, 10694, 42399, 0]   # last
result for depth 24 took too long to be calculated
misplacedTileMaxQueueSize = [0, 4, 9, 30, 170, 818, 3083, 16176]
manhattanDistanceMaxQueueSize = [0, 4, 9, 22, 56, 115, 431, 1499]

uniformCostTime = [0.002, 0.0071, 0.0297,  0.3206, 9.5269, 384.3758, 7879.
1579, 0]   # last result for depth 24 took too long to be calculated
misplacedTileTime = [0.0022, 0.0025, 0.0044, 0.0102, 0.1511, 1.6767, 18.16
04, 742.3022]
manhattanDistanceTime = [0.0012, 0.0066, 0.0095, 0.0108, 0.0259, 0.0796, 0
.5127, 3.6194]


plt.figure(figsize=(10, 10))

plt.subplot(2, 3, 1, xlabel = "Solution Depth", ylabel = "Nodes Expanded")
plt.title("Nodes Expanded vs Solution Depth")
plt.xticks(depths)
plt.plot(depths[:7], uniformCostExpandedNodes[:7], label = "UCS")
plt.plot(depths, misplacedTileExpandedNodes, label = "MT")
plt.plot(depths, manhattanDistanceExpandedNodes, label = "MD")
plt.legend(loc = 2, prop={"size": 10})

plt.subplot(2, 3, 2, xlabel = "Solution Depth", ylabel = "Max Queue Size")
plt.title("Max Queue Size vs Solution Depth")
plt.xticks(depths)
plt.plot(depths[:7], uniformCostMaxQueueSize[:7], label = "UCS")
plt.plot(depths, misplacedTileMaxQueueSize, label = "MT")
plt.plot(depths, manhattanDistanceMaxQueueSize, label = "MD")
plt.legend(loc = 2, prop={"size": 10})

plt.subplot(2, 3, 3, xlabel = "Solution Depth", ylabel = "Time (seconds)")
plt.title("Time vs Solution Depth")
plt.xticks(depths)
plt.plot(depths[:7], uniformCostTime[:7], label = "UCSF")
plt.plot(depths, misplacedTileTime, label = "MT")
plt.plot(depths, manhattanDistanceTime, label = "MD")
plt.legend(loc = 2, prop={"size": 10})

plt.subplot(2, 3, 4, xlabel = "Solution Depth", ylabel = "Nodes Expanded")
plt.title("Zoomed Nodes Exp vs Depth")
plt.xticks(depths)
plt.plot(depths[:6], uniformCostExpandedNodes[:6], label = "UCS")
plt.plot(depths, misplacedTileExpandedNodes, label = "MT")
plt.plot(depths, manhattanDistanceExpandedNodes, label = "MD")
plt.legend(loc = 2, prop={"size": 10})

plt.subplot(2, 3, 5, xlabel = "Solution Depth", ylabel = "Max Queue Size")
plt.title("Zoomed Max Queue Size vs Solution Depth")
plt.xticks(depths)
plt.plot(depths[:6], uniformCostMaxQueueSize[:6], label = "UCS")
plt.plot(depths, misplacedTileMaxQueueSize, label = "MT")
plt.plot(depths, manhattanDistanceMaxQueueSize, label = "MD")
plt.legend(loc = 2, prop={"size": 10})

plt.subplot(2, 3, 6, xlabel = "Solution Depth", ylabel = "Time (seconds)")
plt.title("Zoomed Time vs Depth")
plt.xticks(depths)
plt.plot(depths[:6], uniformCostTime[:6], label = "UCS")
plt.plot(depths, misplacedTileTime, label = "MT")
plt.plot(depths, manhattanDistanceTime, label = "MD")
plt.legend(loc = 2, prop={"size": 10})
plt.tight_layout(pad=4.0)
plt.show()
```