Project 1 CS 170. Introduction to Artificial Intelligence

Jessica Gonzalez
ID 861195307
jgonz117@ucr.edu
Instructor: Dr Eamonn Keogh
5 November 2018

In completing this homework, I consulted…

- https://www.youtube.com/watch?v=ySN5Wnu88nE (For an explanation of A* search)
- https://en.wikipedia.org/wiki/A*_search_algorithm (For more explanation of A* search)
- https://cs.stackexchange.com/questions/37795/why-is-manhattan-distance-a-better-heuristic-for-15-puzzle-than-number-of-til (For a comparison between the manhattan distance heuristic and the misplaced tile heuristic)
- http://www.puzzlopia.com/puzzles/puzzle-8/play (To generate random puzzles)
- Slides from lecture

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

- attrgetter()
    - https://docs.python.org/3/library/operator.html
    - I used this function to grab the Node object's g(n) + h(n)
- max()
    - https://docs.python.org/3/library/functions.html#max
    - I used this function to calculate max queue size
- sort()
    - https://docs.python.org/3/howto/sorting.html
    - I used this function to sort the queue from the attrgetter()'s value returned

CS 170: Project 1 Report

Introduction

In this project, I created a Python 3.7 program that solves the 8-puzzle problem. I used three different searching algorithms: Uniform Cost Search, A-star search with a Misplaced Tile Heuristic, and A-star search with a Manhattan Distance Heuristic. In this report I will analyze the results of these three searching algorithms and then make some concluding statements. At the end of this report you will find an example problem and a snippet of my code. *For all three algorithms, the **cost** of the node is equal to the number of operations it takes to get there. In other words, the cost is equal to the depth of the node.*

I implemented these search algorithms in hopes of answering the following questions:

- How do these algorithms compare in easy and difficult problems?
- How do weak and good heuristics affect the path to the solution?
- How do the nodes expanded (time), and max queue size of the algorithms compare for the same problems?

Uniform Cost Search

The uniform cost search is an algorithm that expands first the node with the lowest cost. Essentially, it is the same as A-star search, but the **heuristic**, h(n), is set to 0.

A-star search: Misplaced Tile Heuristic

The A-star search with Misplaced Tile Heuristic is an algorithm that expands first the node with the lowest sum of the cost and heuristic (g(n) + h(n)). The **heuristic** in this search is calculated by counting the number of tiles that are not in their goal location.

Example

Initial State:	Goal State:
1 2 3	1 2 3
5 6 8	4 5 6
4 7 0	7 8 0

In this example, you can see that 4, 5, 6, 7, and 8 are not in their goal location (0 is ignored). Therefore, the heuristic in this example is equal to **5**.

A-star search: Manhattan Distance Heuristic

The A-star search with Manhattan Distance Heuristic is an algorithm that expands first the node with the lowest sum of the cost and heuristic. The heuristic in this search is calculated by counting the total distance of each tile to its goal location.

Example

Initial State:	Goal State:
1 2 3	1 2 3
5 6 8	4 5 6
4 7 0	7 8 0

In this example, the following are misplaced: 4, 5, 6, 7, and 8. Respectively, their distances to their goal location are 1, 1, 1, 1, 2. The heuristic is therefore **6** (1+1+1+1+2).

Analysis & Comparison

I compared the three algorithms by using the test cases provided to us and a few random puzzles. The test cases provided to us varied by difficulty, easiest being the trivial solution and hardest being an impossible solution.

All three algorithms were very quick when solving problems with a small depth; however, as the problems got more difficult, the time difference became more noticeable among the three algorithms. The general pattern I saw was that the time to find the solution of uniform cost was the greatest and the time of manhattan distance was the smallest.

In the “Oh Boy” test case from the class slides, their differences were very noticeable.
8 7 1
6 0 2
5 4 3

Time it took to complete for the “Oh Boy” test case

	Uniform Cost	Misplaced Tile	Manhattan Distance
Times	1468.7675 s (24 minutes!)	13.2022 s	0.1716 s

Figures

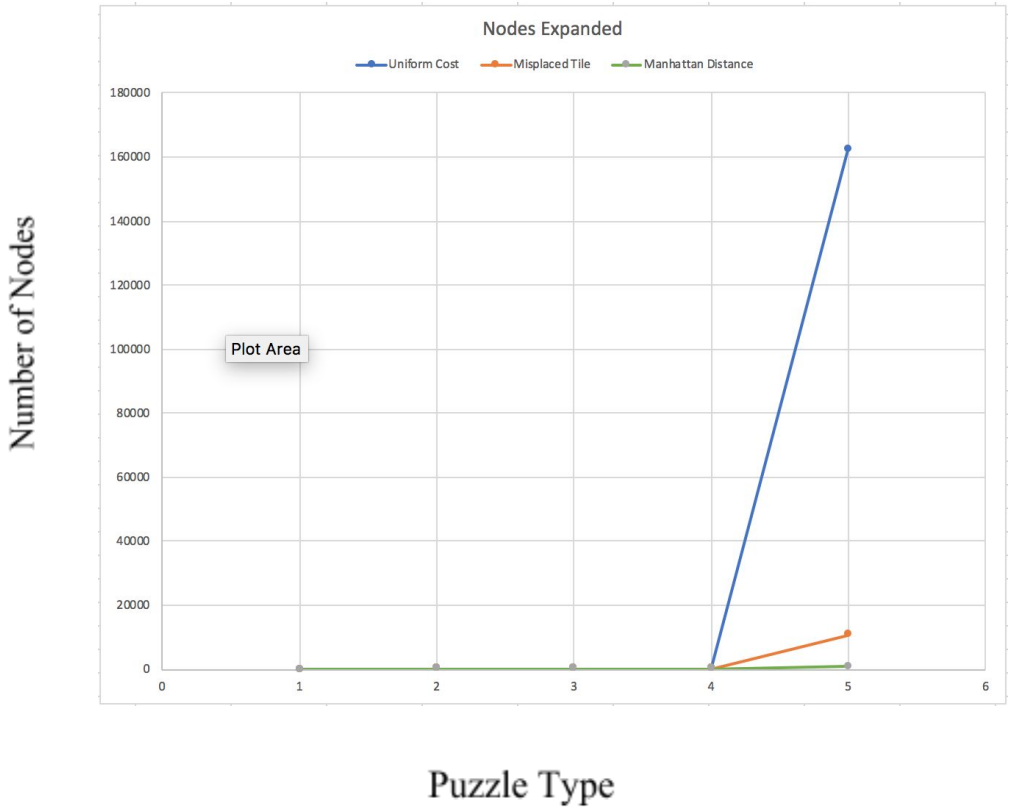
In **figures 1 to 4**, I used the first five test cases from the slides. I excluded the impossible case because each search algorithm searches every node, then gets a "failure" message. In **figures 5 to 8**, I used random puzzles.

F**igure 1** is a table that displays the nodes expanded for each search algorithm among the 5 test cases. In **figure 2**, this same table is plotted. From these two figures we can see that the number of nodes grows very quickly, especially for the search algorithms with weaker heuristics. We also see that Manhattan Distance is the fastest, while Uniform Cost is the slowest.

Figure 1

Nodes Expanded			
	Uniform Cost	Misplaced Tile	Manhattan Distance
Trivial (1)	0	0	0
Very Easy (2)	2	1	1
Easy (3)	3	2	2
Doable (4)	29	4	4
Oh Boy (5)	162093	10719	735

Figure 2

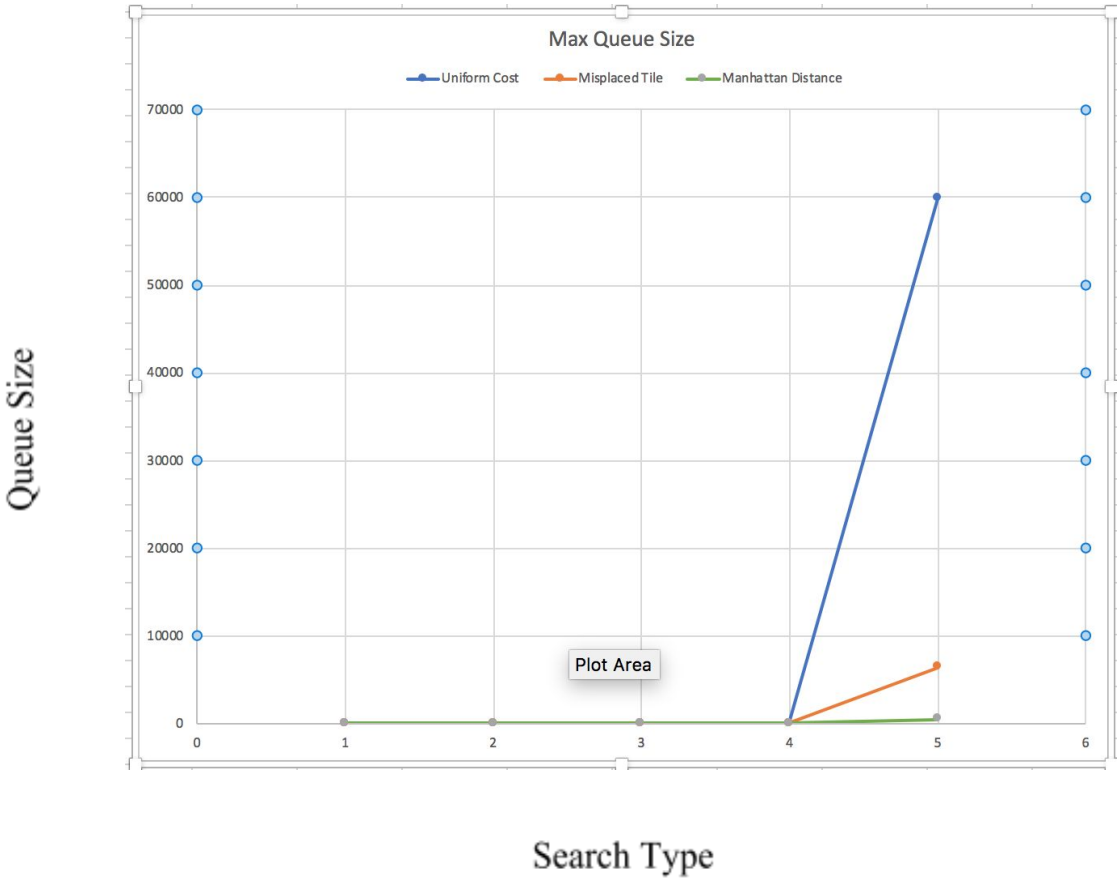


F**igure 3** is a table that displays the max queue size for each search algorithm among the 5 test cases. In **figure 4**, this same table is plotted. Similar to the figures above, the queue size grows very quickly, especially for the search algorithms with weaker heuristics. As we can see, Manhattan Distance has the smallest queue, while Uniform Cost has the largest queue. This means Manhattan Distance is likely to find the solution the quickest.

Figure 3

Max Queue Size			
	Uniform Cost	Misplaced Tile	Manhattan Distance
Trivial (1)	1	1	1
Very Easy (2)	5	3	3
Easy (3)	4	3	3
Doable (4)	18	4	4
Oh Boy (5)	59828	6385	434

Figure 4



F**igure 5** is a table that displays the nodes expanded for each search algorithm among 3 random puzzles. In **figure 6**, this same table is plotted. As we can see, Manhattan Distance is the fastest, while Uniform Cost is the slowest.

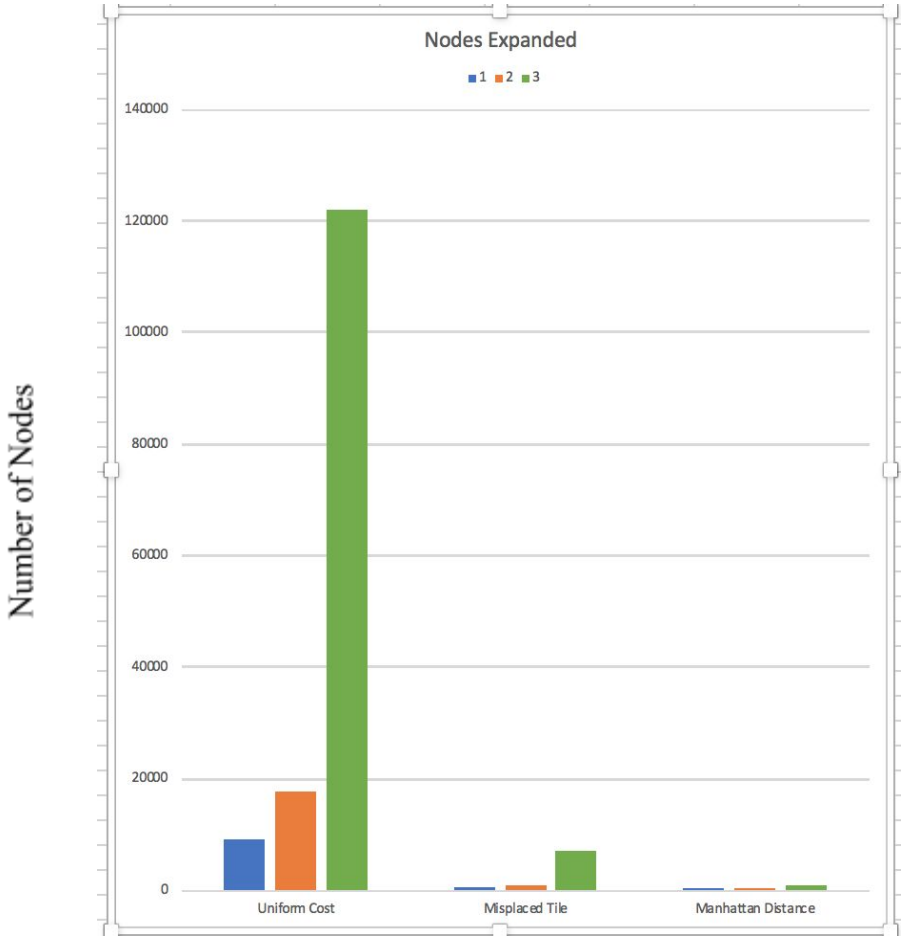
Legend

Puzzle 1	Puzzle 2	Puzzle 3
1 2 6	1 3 5	5 1 3
7 5 8	6 0 4	8 6 0
3 4 0	7 8 2	2 7 4

Figure 5

Nodes Expanded			
	Uniform Cost	Misplaced Tile	Manhattan Distance
1	8917	608	144
2	17676	796	164
3	122055	6931	891

Figure 6

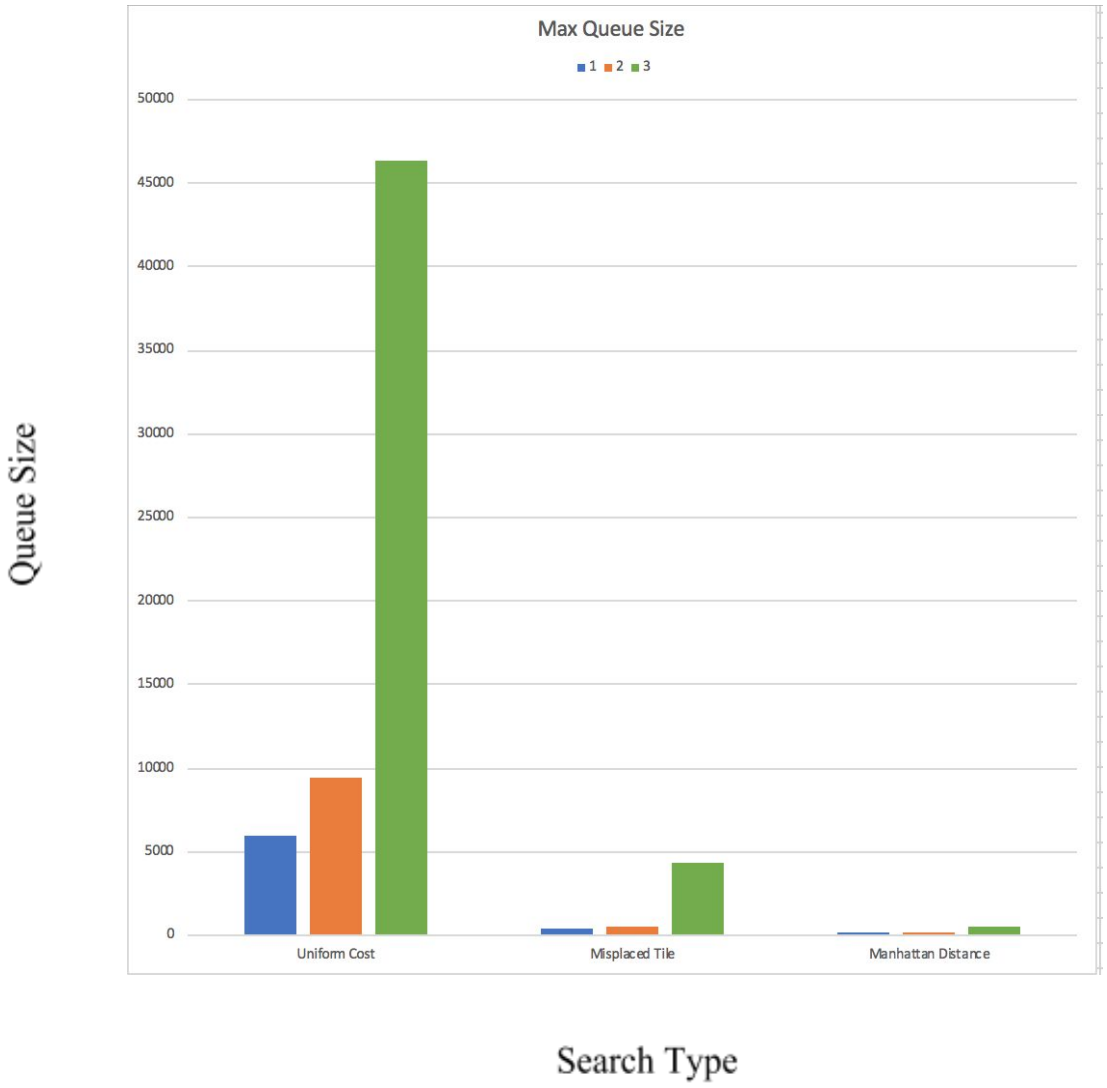


F**igure 7** is a table that displays the maximum queue size for each search algorithm among 3 random puzzles. In **figure 8**, this same table is plotted. As we can see, Manhattan Distance has the smallest queue, while Uniform Cost has the largest queue. This means Manhattan Distance is likely to find the solution the quickest.

Figure 7

Max Queue Size			
	Uniform Cost	Misplaced Tile	Manhattan Distance
1	5903	379	91
2	9457	553	114
3	46295	4310	543

Figure 8



Conclusion

After comparing the three search algorithms, I found that the weaker the heuristic was, the slower it took to find the solution. The heuristic made almost no difference in the easy problems, but in the difficult problems, the stronger heuristics performed better.

For the difficult problems, Manhattan Distance expanded significantly less nodes so it was much faster. The worst was the uniform cost algorithm because it did a poor job at choosing which node to expand next, which resulted in a long time to get to the solution.

Thus, I concluded that the best heuristic for the 8-puzzle project is the Manhattan Distance heuristic. The Misplaced heuristic was not as good, but it performed better than Uniform Cost.

Trace of the Manhattan Sample Problem

The following output example is for the problem in the project 1 specifications.

```
Welcome to Jessica Gonzalez 8-puzzle solver.
Type “1” to use a default puzzle, or “2” to enter your own puzzle.
2

Enter your puzzle, use a zero to represent the blank
Enter the first row, use space or tabs between numbers: 1 2 3
Enter the second row, use space or tabs between numbers: 4 0 6
Enter the third row, use space or tabs between numbers: 7 5 8

Enter your choice of algorithm
1. Uniform Cost Search.
2. A* with the Misplaced Tile heuristic.
3. A* with the Manhattan distance heuristic.
3

Initial state
1 2 3
4 b 6
7 5 8

The best state to expand with a g(n) = 0 and h(n) = 2 is...
1 2 3
4 b 6
7 5 8
Expanding this node...

The best state to expand with a g(n) = 1 and h(n) = 1 is...
1 2 3
4 5 6
7 b 8
```

```
Expanding this node...

Goal!!
1 2 3
4 5 6
7 8 b

To solve this problem the search algorithm expanded a total of 2 nodes.
The maximum number of nodes in the queue at any one time was 5.
The depth of the goal node was 2
```

My Code Snippet

```
import operator
import sys
import time

# Global values
goal_state = ['1', '2', '3', '4', '5', '6', '7', '8', '0']
curr_depth = 0
num_nodes = 0
max_queue = 0
checked_puzzles = []

class Node:
    """This class stores attributes for each node."""

    def __init__(self, puzzle):
        self.heuristic = 0  # if uniform cost search, this doesn't change (h(n))
        self.cost = 0  # Number of actions to get there (g(n))
        self.puzzle = puzzle
        self.total = 0  # g(n) + h(n)
    def set_heuristic(self, type):
        if type == 2:
            self.heuristic = heuristic_misplaced(self.puzzle)
        elif type == 3:
            self.heuristic = heuristic_distance(self.puzzle)
        self.total = self.heuristic + self.cost

    def set_cost(self):
        self.cost = curr_depth
        self.total = self.heuristic + self.cost

def search_function(root, type):
    """Search main function."""

    global num_nodes, max_queue

    # Start queue by adding root
    nodes_queue = [root]

    print('Initial state')
    print_puzzle(root.puzzle)
    print()
```

```
    while len(nodes_queue) > 0:
        max_queue = max(max_queue, len(nodes_queue))

        # Get the cheapest node
        node = nodes_queue[0]

        if node.puzzle == goal_state:
            print('Goal!!')
            print_puzzle(node.puzzle)
            print()
            print('To solve this problem the search algorithm expanded a '
                  'total of ' + str(num_nodes) + ' nodes.\nThe maximum '
                  'number of nodes in the queue at any one time was '
                  + str(max_queue) + '.\nThe depth of the goal node was ' + str(node.cost))
            return

        # Expand Node
        print('The best state to expand with a g(n) = ' + str(node.cost)
              + ' and h(n) = ' + str(node.heuristic) + ' is...')
        print_puzzle(node.puzzle)
        print('Expanding this node...\n')
        num_nodes += 1
        checked_puzzles.append(node.puzzle)
        nodes_queue = expand(node, nodes_queue, type)

    print('Did not find a solution.')

    return

if __name__ == "__main__":

    # Intro
    print('Welcome to Jessica Gonzalez 8-puzzle solver.')
    puzzle_type = input('Type “1” to use a default puzzle, or “2” to enter your own
puzzle.\n')
    print()

    puzzle_arr = []
    start_time = 0

    # Get puzzle
    if puzzle_type == '1':
        puzzle_arr = default_puzzles()
    elif puzzle_type == '2':
        print('Enter your puzzle, use a zero to represent the blank')
        first_row = input('Enter the first row, use space or tabs between numbers:
').replace('\t', ' ')
        second_row = input('Enter the second row, use space or tabs between numbers:
').replace('\t', ' ')
        third_row = input('Enter the third row, use space or tabs between numbers:
').replace('\t', ' ')
        print()
        puzzle_arr = first_row.split(' ') + second_row.split(' ') + third_row.split(' ')

    # Get search algorithm
    print('Enter your choice of algorithm\n'
```

```
      '1. Uniform Cost Search.\n'
      '2. A* with the Misplaced Tile heuristic.\n'
      '3. A* with the Manhattan distance heuristic.')
search_type = input()
print()

# Create root node object
root = Node(puzzle_arr)

# Call appropriate search function
if search_type == '1':
    start_time = time.time()
    search_function(root, 1)
elif search_type == '2':
    start_time = time.time()
    root.set_heuristic(2)
    search_function(root, 2)
elif search_type == '3':
    start_time = time.time()
    root.set_heuristic(3)
    search_function(root, 3)
else:
    print('Invalid search type.')

elapsed_time = time.time() - start_time
print()
print('Time elapsed: ' + str(elapsed_time))
print()
```