Project 1
Monica Canto
SID: 861230103
mcant004@ucr.edu
October 29, 2018

CS170: Introduction to Artificial Intelligence
Dr. Eamonn Keogh

In completing this project, I consulted the following sources:

- The lecture slides + notes on Blind Search and Heuristic Search
- Cloud9 online IDE
- Online GDB
- MATLAB Online computing environment by MathWorks, Version R2018b
  - https://www.onlinegdb.com/online_python_compiler
- An online version of the 8-Puzzle for visualization and further understanding of the problem:
  - http://www.puzzlopia.com/puzzles/puzzle-8/play
  - Note that this version of the 8-Puzzle requires the player to manually move the numbered tiles, rather than moving just the blank space.
- Python 3 documentation
  - https://docs.python.org/3/
- MATLAB documentation
  - https://www.mathworks.com/help/matlab/

All relevant code is original. Unimportant subroutines which are not entirely original include:

- The **math** module which provides access to mathematical functions such as **sqrt()** to calculate square root, and **abs()** to calculate absolute value.
- The **time** module which provides access to functions related to time such as **time()** to return the time in seconds.
  - This function was used for the purpose of timing each algorithm as they were ran, and is commented out in the code.
- The **operator** module with **itemgetter** which returns an object (or a tuple of values) that obtains an item/s from its operand via a getItem() routine.

CS170 Project 1

Monica Canto, SID: 861230103

October 29, 2018

1 Introduction

This is the first project for the Introduction to Artificial Intelligence course taught by Dr. Eamonn Keogh at the University of California, Riverside campus during the Fall 2018 quarter. This write-up consists of my findings throughout completing the project. It explores the search algorithms of Uniform Cost, as well as A* (A-star) which is used for Misplaced Tiles and the Manhattan Distance, separately. My languages of choice were MATLAB for additional visualization of the given test cases for the 8-Puzzle as matrix arrays, and Python 3 for the elaborate coding of the algorithms. The beginning and end of the Python code is attached.

2 Algorithm Comparison

The 3 algorithms implemented for this project are: Uniform Cost Search, Misplaced Tiles Heuristic Search applied by A*, and Manhattan Distance Heuristic Search applied by A*.

2.1 Uniform Cost Search

From the original project handout, the Uniform Cost algorithm is essentially the A* search algorithm with h(n) hardcoded to 0. It involves enqueuing nodes in order of cumulative cost; in other words, it expands the cheapest node with a cost of g(n).

2.2 Misplaced Tiles Heuristic Search

The Misplaced Tile Heuristic Search is a type of A* algorithm which keeps track of the number of misplaced tiles in comparison to the goal state. This number is equivalent to the overall cost to reach the goal state.

2.3 Manhattan Distance Heuristic Search

The Manhattan Distance Heuristic Search is similar to the Misplaced Tile Heuristic where it also involves looking at misplaced tiles, as well as future expansions. However, this algorithm looks at both misplaced tiles and how many tiles away from the position of the goal state. The cost g(n) is a sum of all costs of all misplaced tile distances.

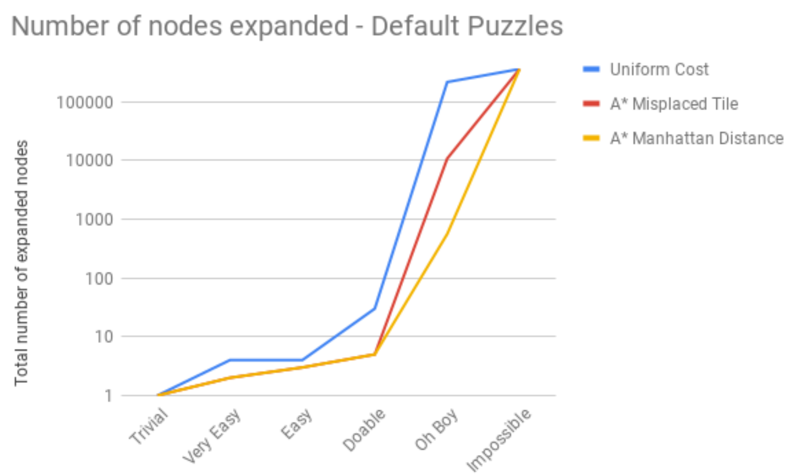
3 Comparison of Algorithms on Various Puzzles

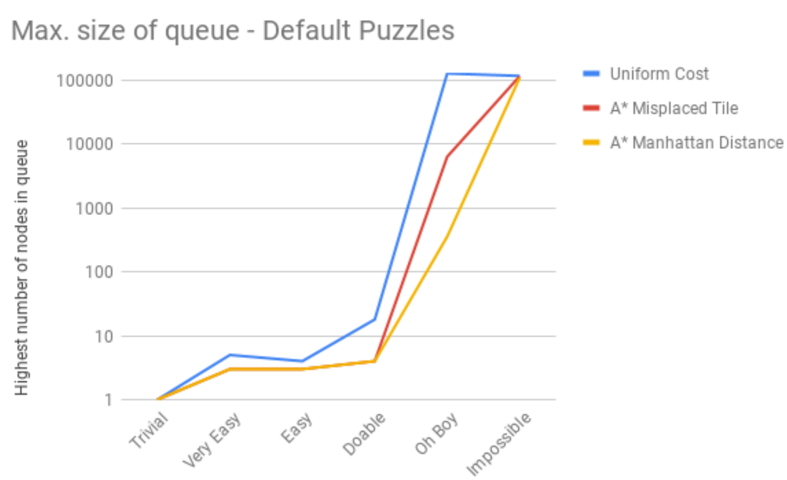
6 default puzzles were provided for implementation, each with varying levels of difficulty. The easiest of the bunch was the Trivial puzzle, which is essentially equivalent to the goal state, while the most difficult was the Impossible puzzle where no solution is found due to it being the goal state puzzle with the 7 and 8 swapped. These puzzles are declared in main.py.

In addition to the default test cases, I have generated 7 of my own puzzles and ran all 3 algorithms on each of them.

The following tables and graphs indicate my results from the aforementioned process for both the default puzzles and my additional puzzles.

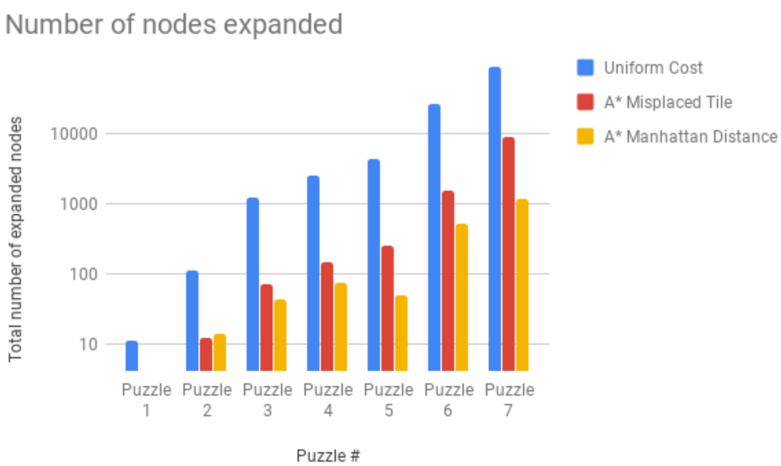
Little to no difference was presented between the 3 algorithms when running on the easier test cases; however, the space complexity became notably different for the more challenging test cases.


Number of nodes expanded - Default Puzzles
Uniform Cost
A* Misplaced Tile
A* Manhattan Distance
Total number of expanded nodes
100000
10000
1000
100
10
1
Trivial
Very Easy
Easy
Doable
Oh Boy
Impossible


Max. size of queue - Default Puzzles
Uniform Cost
A* Misplaced Tile
A* Manhattan Distance
Highest number of nodes in queue
100000
10000
1000
100
10
1
Trivial
Very Easy
Easy
Doable
Oh Boy
Impossible

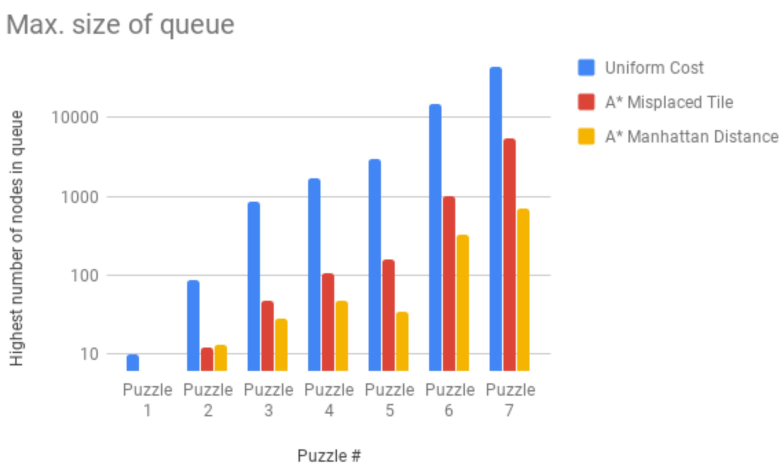
Number of nodes expanded

	Puzzle 1	Puzzle 2	Puzzle 3	Puzzle 4	Puzzle 5	Puzzle 6	Puzzle 7
Uniform Cost	11	113	1228	2510	4277	26045	87418
A* Misplaced Tile	4	12	71	147	247	1547	8953
A* Manhattan Distance	4	14	42	73	49	506	1139



Max. size of queue

	Puzzle 1	Puzzle 2	Puzzle 3	Puzzle 4	Puzzle 5	Puzzle 6	Puzzle 7
Uniform Cost	10	86	862	1686	2911	14799	42930
A* Misplaced Tile	6	12	48	104	158	993	5331
A* Manhattan Distance	6	13	28	48	34	323	695



4 Conclusion

Based on the results from running the provided test cases, it is evident that the A* Manhattan Distance Heuristic Search reigns as the superior algorithm for solving this problem, outperforming the other algorithms as the puzzles became more challenging. For the he A* Misplaced Tile Heuristic Search performed

On the other hand, Uniform Cost Search was shown to be the slowest of the 3 algorithms, as its numbers of nodes which were expanded to reach the goal state, as well as its max. queue sizes, were exponentially larger compared to A* Misplaced Tile Heuristic Search and A* Manhattan Distance Heuristic Search.

5 Trace for A* Manhattan Distance Heuristic

```
Welcome to the 8-Puzzle!
Enter "1" to view a default puzzle, or "2" to enter your own.
2

Enter the elements for your 8-Puzzle.
Type "x" for the blank space.

Enter the elements for row 1:
1 x 3


Enter the elements for row 2:
4 2 6


Enter the elements for row 3:
7 5 8


Initial state

1   x   3
4   2   6
7   5   8

Goal state

1   2   3
4   5   6
7   8   x


Now select an algorithm:
1. Uniform Cost Search
2. A* with Misplaced Tile heuristic.
3. A* with Manhattan Distance heuristic.
3


1   x   3
4   2   6
7   5   8
```

```
State expansion with g(n) = 1 and h(n) = 4:
4   x   6
7   5   8


State expansion with g(n) = 2 and h(n) = 2:

1   2   3
4   5   6
7   x   8


State expansion with g(n) = 2 and h(n) = 2:

1   2   3
4   5   6
7   x   8


State expansion with g(n) = 3 and h(n) = 0:

1   2   3
4   5   6
7   8   x

The goal state was reached!
Number of nodes expanded to solve puzzle with this algorithm: 4
Max. number of nodes in queue: 6
Goal node depth: 3
The algorithm took 0.629425048828125 ms of time.
```

6 Code

main.py
Final number of lines is over 200.

```
import math
import time
from operator import itemgetter

# goal_state = [1, 2, 3, 4, 5, 6, 7, 8, -1]

trivial = [1, 2, 3, 4, 5, 6, 7, 8, -1]

very_easy = [1, 2, 3, 4, 5, 6, 7, -1, 8]

easy = [1, 2, -1, 4, 5, 3, 7, 8, 6]

doable = [-1, 1, 2, 4, 5, 3, 7, 8, 6]

oh_boy = [8, 7, 1, 6, -1, 2, 5, 4, 3]

# debug if a solution is found for this one
impossible = [1, 2, 3, 4, 5, 6, 8, 7, -1]

N_PUZZLE = 8 # N indicates what puzzle it is; i.e. 8, 15, 24, etc.
SIZE_OF_MATRIX = int(math.sqrt(N_PUZZLE + 1))

class priority_queue(object):

    def __init__(self):
        self.elements = []
        self.max_elements = 0

    def get_max_elements(self):
        return self.max_elements

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, h=0, g=0, priority=0):
        self.elements.append((priority, h, g, item))
        self.elements.sort(key=itemgetter(0))
        self.max_elements = self.max_elements if self.max_elements > len(self.elements) else len(self.elements)

    def get_item(self):
        return self.elements.pop(0)

class Problem(object):

        def __init__(self, initial_state=None):
                self.initial_state = initial_state
                self.goal_state = self.get_goal()
                self.explored = []

        def goal_test(self, node):
                self.explored.append(node)
                return node == self.goal_state

        def get_level(self):
                return len(self.explored);

        def is_explored(self, node):
                return node in self.explored

        def get_current_state(self):
                return self.initial_state

        def get_goal_state(self):
                return self.goal_state

        def get_goal(self):
                goal = []
                for x in range(1, N_PUZZLE + 1):
                        goal.append(x)
                goal.append(-1)
                return goal

        def print_current_board(self):
                print_board(self.initial_state)

# prints current state of board
def print_board(mat):
        # print("\nPUZZLE LOOKS LIKE:")
        print("\n")
        # print("*" * 5 * SIZE_OF_MATRIX)
```

```
        count += (row_diff + col_diff)
        return count

# Misplaced Tile heuristic search queuing function
def misplaced_tile_heuristic(nodes, new_nodes):
        while not new_nodes.empty():
                node = new_nodes.get_item()
                nodes.put(node[3], calculate_misplaced(node[3]), node[2], calculate_misplaced(node[3]) + node[2])

# Misplaced Tile heuristic search queuing function
def manhattan_distance_heuristic(nodes, new_nodes):
        while not new_nodes.empty():
                node = new_nodes.get_item()
                nodes.put(node[3], manhattan_distance(node[3]), node[2], manhattan_distance(node[3]) + node[2])

# main function ft. home menu
if __name__ == "__main__":
        print("Welcome to the %d-Puzzle Solver!" % N_PUZZLE)
        print("Enter \"1\" to use a default puzzle, or \"2\" to enter your own puzzle.")
        choice = int(input())
        mat = []
        if choice == 1:
                print("Want to use a default puzzle? Choose from the following:\n1. Trivial\n2. Very Easy\n3. Easy\n4.
                default_puzzle_choice = int(input())
                if default_puzzle_choice == 1:
                        mat = trivial
                elif default_puzzle_choice == 2:
                        mat = very_easy
                elif default_puzzle_choice == 3:
                        mat = easy
                elif default_puzzle_choice == 4:
                        mat = doable
                elif default_puzzle_choice == 5:
                        mat = oh_boy
                elif default_puzzle_choice == 6:
                        mat = impossible
        elif choice == 2:
                print("Enter your %d-Puzzle." % N_PUZZLE)
                print("Type \"x\" for the blank space.\n")
                for i in range(SIZE_OF_MATRIX):
                        print("Enter elements for row %d:" % (i + 1))
                        mat.extend([-1 if x == "x" else int(x) for x in input().split()])
                        print("\n")

        problem = Problem(mat)
        print("Initial state", end=' ')
        problem.print_current_board()
        print("\n")
        print("Goal state", end=' ')
        print_board(problem.get_goal_state())
        print("\n")
        # print("*"*50)
        print("Enter your choice of algorithm:\n1. Uniform Cost Search\n2. A* with the Misplaced Tile heuristic.\n3. A
        choice = int(input())
        # t1 = 0
        if choice == 1:
                # t1 = time.time()
                general_search(problem, uniform_cost_search)
        elif choice == 2:
                # t1 = time.time()
                general_search(problem, misplaced_tile_heuristic)
        elif choice == 3:
                # t1 = time.time()
                general_search(problem, manhattan_distance_heuristic)
        else:
                print("Try again!")
        # t2 = time.time()
        # print("Time: " + str((t2-t1) * 1000)  + " ms")
```