Assignment 1
CS 170, Introduction to Artificial Intelligence

Alic Lien
Instructor: Dr. Eamonn Keogh
ID 861207479
alien002@ucr.edu
5 November, 2019

In completing this homework, I consulted…

- The Lecture Slides for Blind and Heuristic Search
- Documentation for Python 2.7, and 3.X. The documentation can be found at https://docs.python.org/2/index.html and https://docs.python.org/3.5/index.html
- StackOverflow for certain error fixing, and usage of Python subroutines listed below.
    - https://stackoverflow.com/questions/53554199/heapq-push-typeerror-not-supported-between-instances
    - https://stackoverflow.com/questions/19389490/how-do-pythons-any-and-all-functions-work
    - https://stackoverflow.com/questions/17246693/what-is-the-difference-between-shallow-copy-deepcopy-and-normal-assignment-oper
- The sample report posted in the class website as reference
- The project briefing for the different test cases.
- Daniel Li, my study partner for many classes. We consulted each other on high level implementation of the search routine, and proper utilization of Python subroutines.

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

- Python **random,** only used to generate random test cases, not utilized in the final build.
- Python **copy,** used to create deep copies of the Eight Puzzle board matrix array.
- Python **heapq,** mainly utilized to create the queues, and priority queues needed for the search algorithm.
- Python **numpy,** utilized to create and save Eight Puzzle board matrix array in the desired format. Also utilized with its **numpy.argwhere(a)** subroutine to find and locate the “0” within the board.

CS170: Project 1 Write Up

Alic Lien, 861207479

Introduction

This Project is the first assignment from Dr. Keogh's Introduction to AI course, Fall 2019, University of California, Riverside. The project utilizes Uniform Cost Search, A* Misplaced Tile Heuristic, and A* Manhattan Distance Heuristic to showcase the different memory takeup, and runtimes for the three searches to solve an Eight Puzzle. The program is written in Python 3.X, as compared to C++, I am able to utilize the functionality and language as a whole to complete this program in less time with Python. This report will explain the Three search algorithms, an example of the heuristics, comparison results of the algorithms, my analysis of the code, and a quick Trace of the A* Manhattan Distance Heuristic, along with the Source Code.

The Search Algorithms

The Three Search Algorithms implemented within this program are Uniform Cost Search, A* Misplaced Tile Heuristic, and A* Manhattan Distance Heuristic.

Uniform Cost Search

For Uniform Cost Search, the heuristic h(n) is set to equal 0, as stated within the lecture slides. The cost of each expanded node will equal to 1. Due to the underlying conditions, Uniform Cost Search in this project degrades to a basic Breadth First Search.

A* Misplaced Tile Heuristic

With Misplaced Tile Heuristic, the heuristic, h(n), counts the number of misplaced tiles (excluding the empty or "0" tile) within the puzzle. When a node is added to the Priority Queue, the queue is sorted based on the cost, the number of misplaced tiles, in ascending order.

A* Manhattan Distance Heuristic

With Manhattan Distance Heuristic, we have a similar operation to Misplaced Tile Heuristic. The difference is that, not only do we count the number of tiles that are misplaced, we also calculate the distance (number of moves) that are needed to shift the misplaced tiles back to its correct position.

Heuristics Example

If we look at the Easy Puzzle within the test cases below:

Starting Board:	Goal Board:
1 2 *	1 2 3
4 5 3	4 5 6
7 8 6	7 8 *

Misplaced Tiles:

For this algorithm, we see that there are 2 misplaced tiles: 3 and 6, that are currently not in their desired Goal location, therefore the heuristics h(n) = 2.

Manhattle Tiles:

* * ^	* * *
* * 3	* * ^
* * *	* * 6

For this algorithm, since “3” and “6” are both misplaced and only require 1 step to move to their desired spaces respectfully, we have h(n) = 1 + 1 = 2.

Test Cases:

Trival

1 2 3
4 5 6
7 8 *

Easy

1 2 *
4 5 3
7 8 6

Oh Boy

8 7 1
6 * 2
5 4 3

IMPOSSIBLE: The following puzzle is impossible to solve, if you *can* solve it, you have a bug in your code.

1 2 3
4 5 6
8 7 *

Very Easy

1 2 3
4 5 6
7 * 8

doable

* 1 2
4 5 3
7 8 6

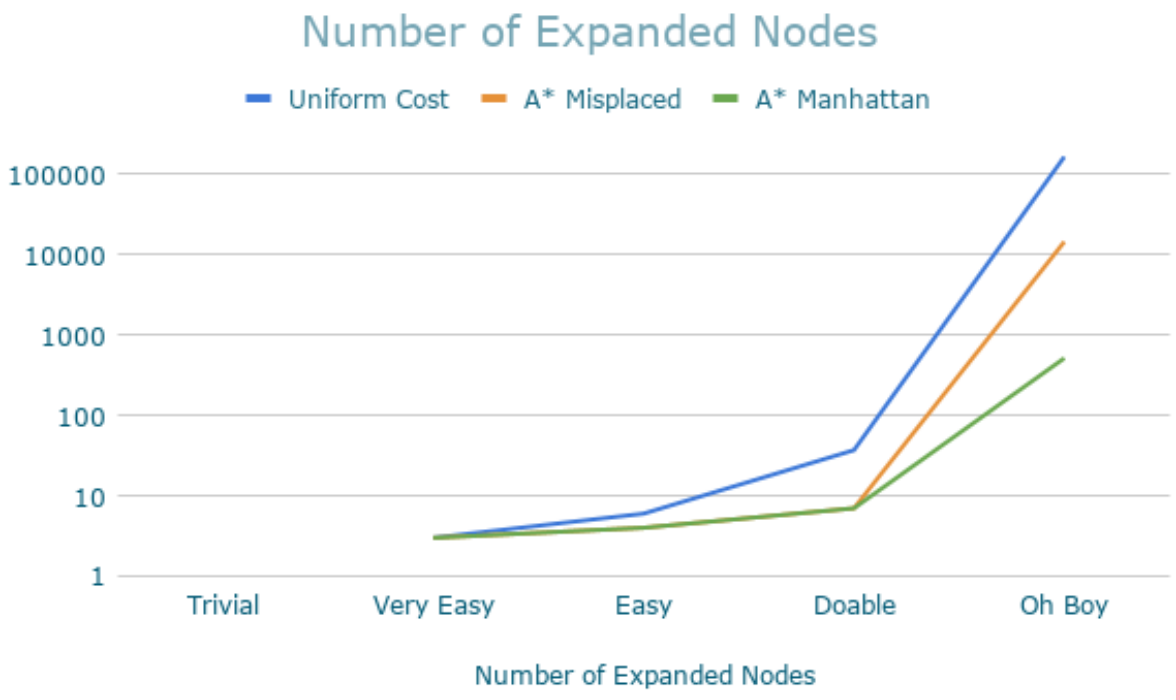
Image taken from the Eight-Puzzle_Briefing.pdf page 4, found on the class website.

Results of the Test Cases

Number of Expanded Nodes			
	Uniform Cost	A* Misplaced	A* Manhattan
Trivial	0	0	0
Very Easy	3	3	3
Easy	6	4	4
Doable	37	7	7
Oh Boy	165020	14468	515
Impossible	181439	181439	181439

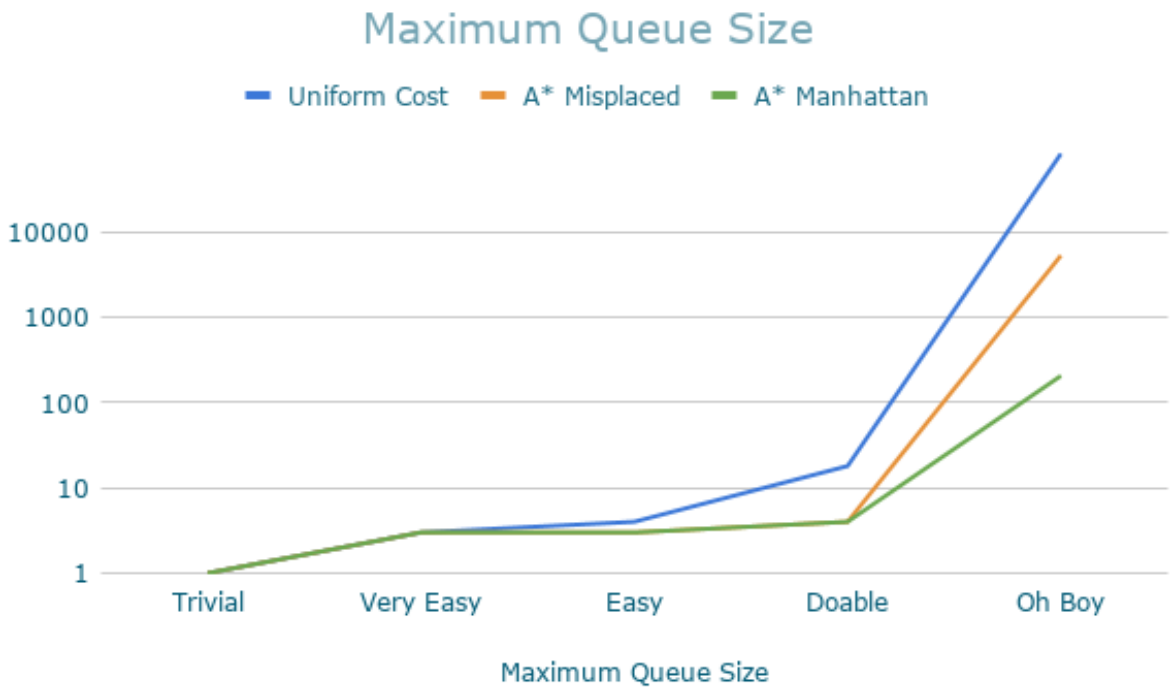
*Oh Boy Uniform Number of Nodes is Estimated, as hardware was unable to finish the run, due to the number of nodes, and runtime.

**Impossible Number of Expanded Nodes based on the Calculation (9!/2) -1 for number of possible Eight Puzzle nodes excluding the starting node.



Maximum Queue Size			
	Uniform Cost	A* Misplaced	A* Manhattan
Trivial	1	1	1
Very Easy	3	3	3
Easy	4	3	3
Doable	18	4	4
Oh Boy	81936	5265	205

*Oh Boy Uniform Maximum Queue Size is Estimated, as hardware was unable to finish the run, due to the number of nodes, and runtime.



Analysis and Conclusion

All three algorithms ran at similar runtimes when the problem was within the Trivial to Doable range. Once we reached Doable, we can see that Uniform ran slower compared to A* Misplaced, and A* Manhattan. With Oh Boy, we can definitely see that, especially with the results, A* Manhattan ran the fastest at about 1-2 minutes, with A* Misplaced coming in second at about 5 minutes, and Uniform being the slowest, running on my machine for 6+ hours.

In conclusion, with the data presented and the time it took for my computer to run the algorithms, we can see that A* Manhattan is the most favorable one, as in every case, it has the fast runtime, compared to all the other Algorithms. Not only that, A* Manhattan utilizes the least amount of memory space with the least number of expanded nodes, and the least number of Maximum Nodes in Queue.

Trace of A* Manhattan Distance Heuristic

```
Welcome to Alic Lien's 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 tab between the three numbers:
1 2 0
Enter the Second row, use space or tab between the three numbers:
4 5 3
Enter the Third row, use space or tab between the three numbers:
7 8 6

Enter your choice of Algorithm:
   1. Uniform Cost Search
   2. A* with Misplaced Tile Heuristic
   3. A* with Manhattan Distance Heuristic
3
Expanding state
[[1 2 0]
 [4 5 3]
 [7 8 6]]
```

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

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


Goal!!!


To solve this problem the search algorithm expanded a total of 4 nodes.
The maximum number of nodes in the queue at any one time was 3 .
The Depth of the goal node was: 2
```

Source Code:

```python
import random
import copy
import heapq as priQ
import numpy as np


#Node stores Array, and uniform, heuristic, and cost values.
class node:
    def __init__(self, data = None):
        self.data = data
        self.uniform = 0
        self.heuristic = 0
        self.cost = 1

    #needed to accomodate heapq
    def __lt__(self, other):
```

```
        return self.cost < other.cost

    #Checks to see if current node is solved (based on Goal Node)
    def solved(self):

    #Checks if current node is repeated (compares with boards in the
visited array)
    def isRepeat(self, dir_node):
        if matched == 9:
            return True
        return False

#Initial Blank Puzzle
INIT = np.array([[0, 0, 0],
                 [0, 0, 0],
                 [0, 0, 0]])

#Counts number of misplaced tile, returns heuristic value
def misplaced(input_node):

#Evaluates the distance between tiles, returns heuristic value
def manhattan(input_node):

#Shifts 0 into desired direction
def shift(curr_node, direction):
    #Deep copy to fully copy the puzzle board
    new_board = copy.deepcopy(curr_node.data)

    #Finds the location of "0" on the board
    zeroloc = np.argwhere(curr_node.data == 0)

#Search Function, takes in function to determine which one to use
def search(problem, function):
    queue = []
    visited = []
    priQ.heapify(queue)

    expand = 0
    max = 0
    depth = 0

    #Selects which search to use
```

```
    if(function == "1"):
        #print("Uniform Cost Search Selected.\n")
        problem.cost = 1
    if(function == "2"):
        #print("A* with Misplaced Tile Heuristic Selected.\n")
        problem.cost = 1
        problem.heuristic = misplaced(problem)
    if(function == "3"):
        #print("A* with Manhattan Distance Heuristic Selected.\n")
        problem.cost = 1
        problem.heuristic = manhattan(problem)

    priQ.heappush(queue, problem)

    while (len(queue) > 0):
        priQ.heapify(queue)
        #expand += 1
        if len(queue) > max:
            max = len(queue)

        curr = priQ.heappop(queue)

        #Use for text formatting
        if (curr.uniform != 0):
            print("The best state to expand with a g(n)
=",curr.uniform,"and h(n) =",curr.heuristic, "is...")
            print(curr.data)
            print("Expanding this node...")
        else:
            print("Expanding state")
            print(curr.data)

        print("")
        if (curr.solved()):         #If answer is found
            depth = curr.uniform
            print("\nGoal!!!\n\n")
            print("To solve this problem the search algorithm expanded a
total of", expand, "nodes.")
            print("The maximum number of nodes in the queue at any one time
was", max,".")        #max queue size
            print("The Depth of the goal node was:", depth)      #number of
moves to solve
```

```
            return;
        else:                       #Continue Search when Answer not found
            visited.append(curr)
            up = node(shift(curr, 1))

            up.uniform = curr.uniform + 1

            #misplaced
            if (function == "2"):
                up.heuristic = misplaced(up)
            #manhattan
            if (function == "3"):
                up.heuristic = manhattan(up)

            #Calculates cost with f(n) = g(n) + h(n)
            up.cost = up.uniform + up.heuristic
            down.cost = down.uniform + down.heuristic
            left.cost = left.uniform + left.heuristic
            right.cost = right.uniform + right.heuristic

    print("\nSearch Failed...\n\n")
    print("The search algorithm expanded a total of", expand,"nodes.\n")
    print("The maximum number of nodes in the queue at any one time was",
max)
    return


#main#
print("Welcome to Alic Lien's 8-Puzzle Solver!\n")
print("Type '1' to use a default puzzle, or '2' to enter your own puzzle")

choice1 = input()

#Initiate the Start and Goal nodes
StartNode = node(INIT)
GoalNode = node(GOAL)

print("\nEnter your choice of Algorithm: ")
print("   1. Uniform Cost Search")
search(StartNode, choice2)
```