CS 170: Assignment 1

Kevin Prada
SID: 862311940
Due: October 31st, 2022

The Eight-Puzzle

An exploration of search algorithms

In completing this assignment I consulted:

- The search lecture slides from Professor Keogh's CS 170 class and my own notes from lecture.
- Python 3.11.0 Documentation, sourced at https://docs.python.org/3/.
- Clipart and UC Riverside logo found through Google Docs Image Search.
- GeeksforGeeks.org for concepts such as using lambda for sort() functions. (https://www.geeksforgeeks.org/python-lambda/). I also consulted GeeksforGeeks for overloading operators in conjunction with the sort() function and in() functions. (https://www.geeksforgeeks.org/operator-overloading-in-python/). Finally, I also consulted it for using the map() function at (geeksforgeeks.com/python-map-function()).
- I also took heavy format and structure inspiration from the sample report given in Professor Keogh's Project #1, done by Sue Mee.

All important code is original and done solely by myself. Unimportant libraries and methods that are not original and have been imported are:

- All subroutines used from **time**, to handle the algorithm efficiency and analysis.
- All subroutines used from **copy**, to deepcopy and modify Puzzle objects.

Outline of this report:

- Cover Page (Page 1)
- Report (Pages 2-5)
- Sample Traces (Pages 6-8)
- My Code Github Link (Page 9)
- Snippet of my code (Pages 9-12)

Introduction

Sliding tile puzzles are combination puzzles where the objective is to slide flat uniquely identified pieces around until a solution is found. They are played in n x n grids, or squares, where one of the pieces is left out. For example, in a 4 x 4 tile puzzle, you have 15 possible pieces with one open space. Because you have an empty square in the grid, any piece surrounding that empty spot can be moved into it, thus leaving the square it previously occupied now empty. A tile may be moved up, down, left, and right depending on where this empty spot is located.

Figure 1: A clipart picture of a 15-tile sliding puzzle for reference.

This first project is for Dr. Eamonn Keogh's CS 170: Introduction to Artificial Intelligence. It is based in the University of California, Riverside's Bourns College of Engineering during the quarter of Fall 2022. This report is to detail my work as well as show my mastery of the concepts learned in search and utility of heuristics. Specifically, I'll be using Uniform Cost Search, as well as Misplaced Tile and Manhattan Distance Heuristics applied to the A* algorithm. I'll be utilizing the Python (Version 3) programming language to complete my project, and all original and cited code is included below.

Algorithm Explanation

Overall, we will be using the General Search Algorithm, which is a search algorithm with the property of being "a single algorithm to do many kinds of search" [2] (Keogh, pre-blind search). Because it is a single universal algorithm, the single difference is "in how the nodes are placed in the queue"[2] (Keogh, blind search.The algorithm allows for two parameters. The first being the problem we are trying to solve, the next being the type of queueing function we will use in order to solve it.

Context

The General Search algorithm will involve continuous instructions, such as checking for empty search tree failures, goal state arrival, and node expansions until either a solution is reached or no solution is found possible. We will be moving from a current state to the next possible state using a particular method, and will be calculating the cost function G(n), which is the cost to get to a state n from the initial state. We will also have the heuristic function H(n), which returns a number that is “an estimate of the merit of the state, with respect to the goal” [2] (Keogh - heuristic search). We will then use these together for the A* methods.

Uniform Cost Search

As explained within Professor Keogh’s slides, Uniform Cost Search is a simple algorithm which “enqueues nodes in order of (cumulative) cost, the G(n)” (Keogh, Uniform Cost Search). We are essentially expanding the current cheapest path node wherever we are with the current state. No heuristics are used that can help the search, which means we are doing a type of blind search. This is the least efficient algorithm of the three we are using within this project.

Misplaced Tile Heuristic (A-Star)

With A-Star, we now use a particular heuristic that will help guide our search to be more efficient. In this case, our heuristic is the Misplaced Tile Heuristic. This method checks the number of tiles that are not in their proper place. Since we are using A-Star, our new cost function will be F(n) = G(n) + H(n), where the Misplaced Tile Heuristic = H(n). Now, the node expanded will feature the lowest sum of the depth and MTH.

Manhattan Distance Heuristic (A-Star)

Another A-Star algorithm, our heuristic now features the misplaced tiles, but now with significance on how far away they are from their proper places. It starts by seeing which tiles are misplaced, then calculates how far each of those tiles are, or their distance to their respective proper tile placement on the puzzle. Similarly, our cost function will be F(n) = G(n) + H(n), where

the Manhattan Distance Heuristic = H(n). The node next to be expanded will feature the lowest sum of the depth and MDH.

Analysis and Comparison

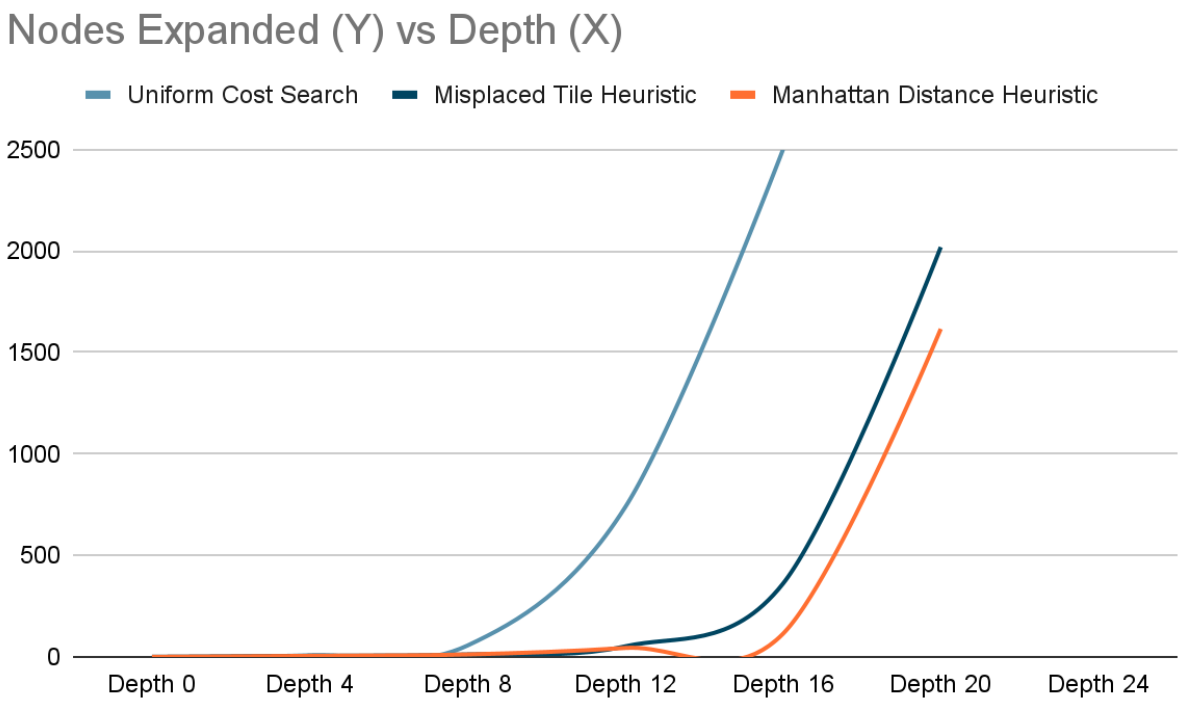

Figure 3: Graph comparing node expansions at certain depths of each algorithm.

As we can tell in Figure 3 above and Figure 4 below, utilizing a blind search approach starts to get extremely inefficient after depth 8. Because it is without heuristic support, it is not able to be performed comfortably with puzzles that are more complex. On the other hand, with the heuristic searches, we can see that both Misplaced Tile and Manhattan Distance heuristics do well onto depth 20. We can also observe that while both algorithms are doing well, Manhattan

Distance is the most efficient algorithm as it expands the least nodes of all three, as well as boasts the best run-time complexity of all..

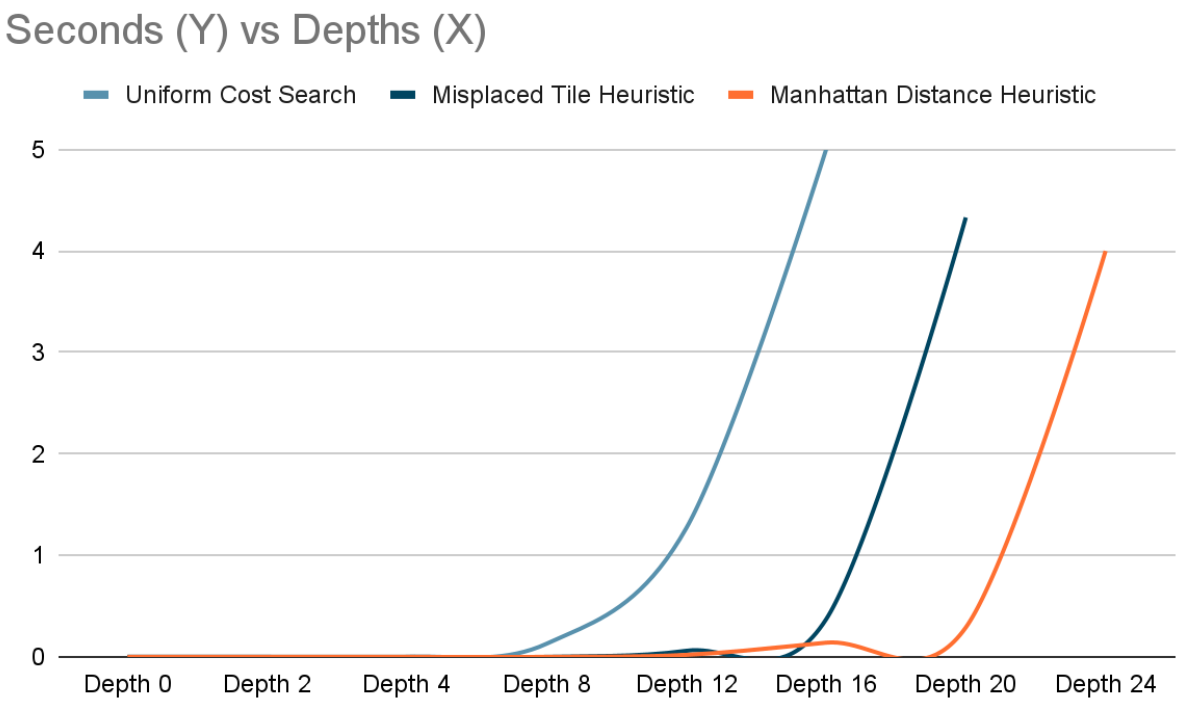

Figure 4: Graph comparing algorithm run-time at certain depths of each algorithm.

We can perform an analysis and compare efficiency at a depth such as 16. At that depth, the misplaced tile heuristic takes 0.37 seconds to complete the puzzle, while the manhattan distance heuristic takes 0.14 seconds. This shows a 164% increase in efficiency when using the manhattan distance heuristic. At less complex depths, these heuristics tend to have similar efficiency. However, as the depths become deeper, we can see that the manhattan distance heuristic reigns king.

Conclusion

Throughout the project, we can observe explicit differences between our algorithms that we utilize within a search problem.

The following is a traceback of an **easy** puzzle:

Thanks for choosing to play the Sliding Puzzle Solver!

Select 1 to create your own puzzle, 2 to choose a ready-made 8-puzzle! 2

Default puzzles are 3x3.

Choose a difficulty from 1-8: 2

Puzzle has been successfully selected.

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

The best state to expand -

G(n) = 0

Misplaced Tile H(n) = 2

Manhattan Distance H(n) = 4

[1, 2, 3]

[4, 5, 6]

[0, 7, 8]

The best state to expand -

G(n) = 1

Misplaced Tile H(n) = 1

Manhattan Distance H(n) = 2

[1, 2, 3]

[4, 5, 6]

[7, 0, 8]

The best state to expand -

G(n) = 2

Misplaced Tile H(n) = 0

Manhattan Distance H(n) = 0

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

Goal State!

Solution depth was 2

Number of nodes expanded: 5

Max queue size: 3

The following is a traceback of a **difficult** puzzle:

Thanks for choosing to play the Sliding Puzzle Solver!

Select 1 to create your own puzzle, 2 to choose a ready-made 8-puzzle! 1

Puzzles are N x N. Input your dimension: 3

Enter your puzzle, using a zero to represent the open tile.Enter your numbers separated by a space.

Input numbers for row 1

0 5 3

Input numbers for row 2

1 7 4

Input numbers for row 3

2 6 8

Here's the solution for this grid:

======

1 2 3

4 5 6

7 8 0

======

Puzzle has been successfully selected.

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

The best state to expand -

G(n) = 0

Misplaced Tile H(n) = 7

Manhattan Distance H(n) = 16

[0, 5, 3]

[1, 7, 4]

[2, 6, 8]

The best state to expand -
G(n) = 1
Misplaced Tile H(n) = 6
Manhattan Distance H(n) = 14
[1, 5, 3]
[0, 7, 4]
[2, 6, 8]

// Deleting pages of expansions to save space

The best state to expand -
G(n) = 23
Misplaced Tile H(n) = 1
Manhattan Distance H(n) = 2
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]
The best state to expand -
G(n) = 24
Misplaced Tile H(n) = 0
Manhattan Distance H(n) = 0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Goal State!
Solution depth was 24
Number of nodes expanded: 8838
Max queue size: 1836
3.18 seconds elapsed

Github Link

https://github.com/pradakev/AIPuzzleSearch

Snippet of my code

```
from puzzle import Puzzle

class AStar:
    """
    ASTAR -
    A class representing the queueing function that will be used
    for the General Search Algorithm. Functions will include the
    Uniform Cost Search, Misplaced Tile, and Manhattan Distance algorithms.
    """
    def __init__(self, heuristic):
        self.heuristic = heuristic


    """
    UNIFORM COST SEARCH -
    Nodes are sorted by depth cost only. Done by taking the
    current nodes on the queue, nodes expanded, and sorting
    by lowest depth using the lambda function.
    """
```

```
def UniformCostSearch(self, nodes, nodeExpansion, visited):
    self.nodes = nodes
    self.nodeExpansion = nodeExpansion
    self.visited = visited
    self.checkRepetitions()
    # UCS Should return the list with the lowest value
    # to the right of the the queue. (Descending order)
    newNodesList = self.nodeExpansion + self.nodes
    # Cite key=lambda sort here
    # Lambda is used here to sort by Puzzle object's depth variable.
    # x stores the variable. Since we need descending, we set reverse=True.
    newNodesList = sorted(newNodesList, key=lambda x: x.depth, reverse=True)
    DEBUG = False
    if DEBUG:
        print("&" * 8)
        for i in range(len(newNodesList)):
            newNodesList[i].printGrid()
        print("&" * 8)
    return newNodesList

"""
```

```
ASTAR WITH MISPLACTED TILE HEURISTIC -
Nodes are sorted by depth cost summed with the misplaced tile h(n).
"""
def AStarMTH(self, nodes, nodeExpansion, visited):
    self.nodes = nodes
    self.nodeExpansion = nodeExpansion
    self.visited = visited
    self.checkRepetitions()
    # MTH Should return the list with the lowest value
    # to the right of the the queue.
    newNodesList = self.nodeExpansion + self.nodes
    # Cite key=lambda sort here
    # Lambda is used here to sort by Puzzle object's depth variable summed
    # with Puzzle's misplaced tile Heuristic.
    # x stores the variable. Since we need descending, we set reverse=True.
    newNodesList = sorted(newNodesList, key=lambda x:
    x.depth + x.misplacedTileH, reverse=True)
    DEBUG = False
    if DEBUG:
        print("&" * 8)
        for i in range(len(newNodesList)):
```

```
            newNodesList[i].printGrid()
        print("&" * 8)
    return newNodesList
```