Phillip Nguyen
862050241

CS170 Project #1

Objective/Introduction:

This project's purpose is to get experience in using A* search to solve the eight puzzle problem, or sliding block. I implement three different algorithms, and then compare them to each other in solving this puzzle. The algorithms in use are uniform cost search, A* with misplaced tile, and then A* with euclidean distance. This project is coded in Python 3.7 and a full trace is provided at the end of the report**(pg.10)**.

Design:

- Moves in this program are based on where the zero is located.
    - Up means the zero swaps with whatever is above it
    - Right means zero swaps with whatever is to the right of it
- Node class to act like a state of puzzle
    - Each node stores an action and parent
        - Allows us to trace program and print out corresponding moves
- Problem class, in the future allows it to be extended
- Lots of helper functions that are generalized as much as possible to allow extension of the program(such as printing, and grabbing user inputs)
- Keeps a set of explored nodes(graph search)
- Priorityqueue to manage the frontier
    - Had to add tie breaker, since priorityqueue used was not stable
- Set for explored set of nodes
    - Faster accesses

Functionality:

- Solve 8 puzzle problem with 3 different algorithms
- Output a trace/solution which includes:
    - Direction that the zero piece in the puzzle moved for each step
    - The costs associated with the nodes
    - What the puzzle looks like at each step
- Allows you to enter custom puzzle, and will solve it as long as it is valid
- Outputs an estimate running time to solve the puzzle

Algorithm Comparison

Uniform Cost Search

This algorithm is A* with the heuristic hardcoded to 0. The cost of each node would be g(n), so each expanded node has a cost value of 1. In this project, the runtime for this program is too long for the board setup labelled oh boy. It expanded for too long and my system could not solve that particular board setup with this algorithm.

Misplaced Tile

The next algorithm is A* with a heuristic function based on misplaced tiles. I count the number of spots that differ from the goal state except1 the position of the zero. This number factors in the cost of the node. So when we pick a node from the frontier, the amount of misplaced tiles becomes a factor on what is the best choice.

Euclidean Distance Heuristic

The third algorithm is A* with a heuristic function based on euclidean distance. The euclidean distance formula is similar to the Pythagorean theorem. You find the distance from the current tile to the goal tile using the formula, and you sum them all up to get the heuristic value for this algorithm.

Brief Comparison with the Three Algorithms

All three algorithms performed almost the same on the easier puzzles. Uniform cost expanding more states, but with the easier puzzle, the heuristic didn't matter as much. As the initial puzzle state got more difficult, such as with the Oh Boy initial state. My findings saw that the uniform cost didn't solve it, while the other two algorithms with heuristic function managed to find an answer. Below will be data on these algorithms on different initial puzzle states.

Testing Dataset:

Trivial:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

Very Easy:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 0 & 8 \end{bmatrix}$$

Easy:

$$M = \begin{bmatrix} 1 & 2 & 0 \\ 4 & 5 & 3 \\ 7 & 8 & 6 \end{bmatrix}$$

Doable:

$$M = \begin{bmatrix} 0 & 1 & 2 \\ 4 & 5 & 3 \\ 7 & 8 & 6 \end{bmatrix}$$

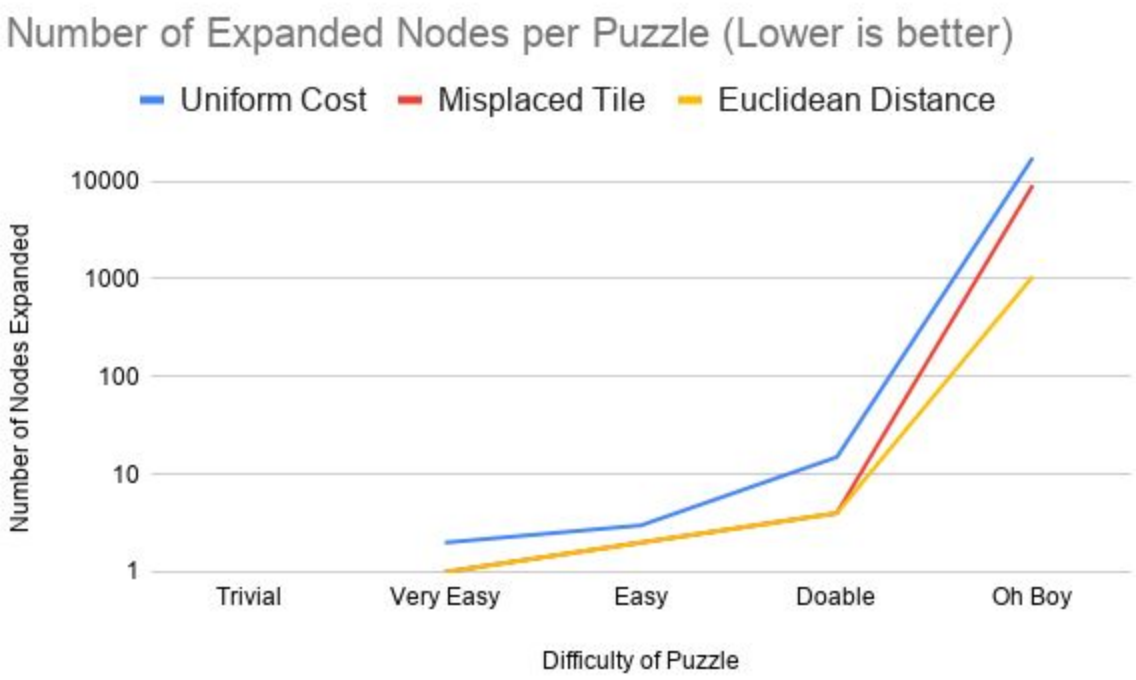
Oh Boy:

$$M = \begin{bmatrix} 8 & 7 & 1 \\ 6 & 0 & 2 \\ 5 & 4 & 3 \end{bmatrix}$$

Data:

Data for Number of Nodes Expanded:

Expanded Nodes	Uniform Cost	Misplaced Tile	Euclidean Distance
Trivial	0	0	0
Very Easy	2	1	1
Easy	3	2	2
Doable	15	4	4
Oh Boy	17432	9103	1058



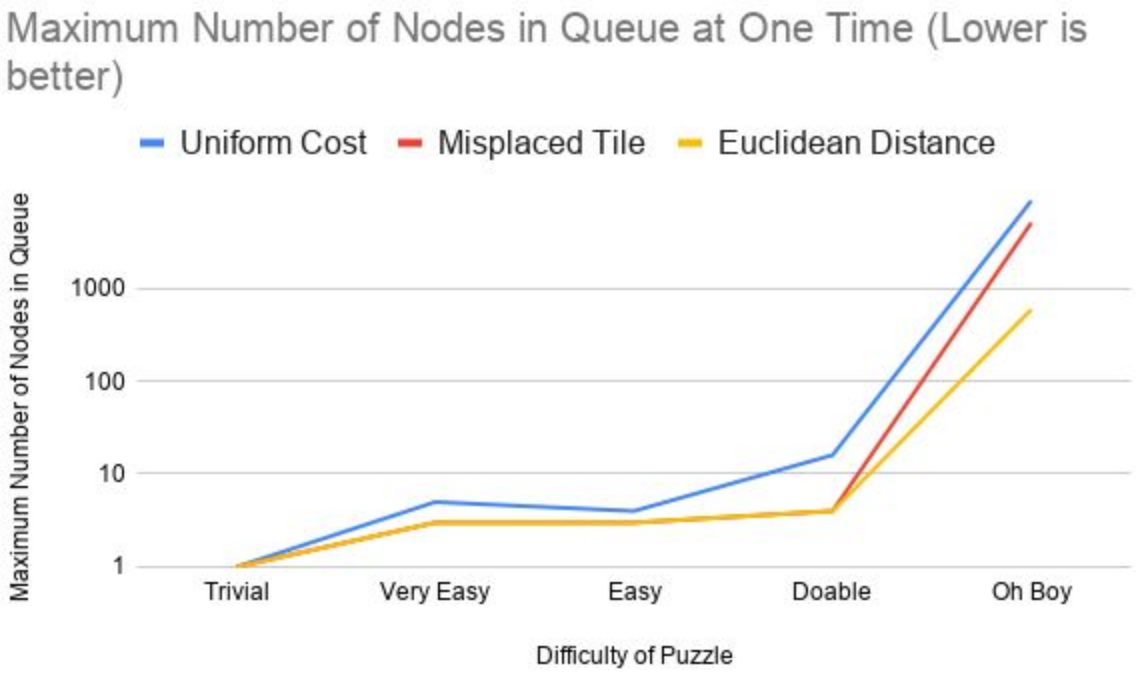
Quick Analysis of the Data Above:

- Uniform Cost did not finish running, max number I got before I had to stop running
- Notice how a good heuristic is much better than a weak heuristic. As the puzzle gets more difficult, it is solved in a fraction of the time. Euclidean distance is much faster than the other two algorithms.

- A weak heuristic is better than nothing, because a the weak heuristic still solved the problem

Data for Maximum Number of Nodes in Queue:

Max Num in Queue	Uniform Cost	Misplaced Tile	Euclidean Distance
Trivial	1	1	1
Very Easy	5	3	3
Easy	4	3	3
Doable	16	4	4
Oh Boy	8738	5051	591



Quick Analysis of the Data Above:

- Uniform Cost did not finish running, max number I got before I had to stop running
- The space for both heuristics stayed the same for easier problems, as soon as the difficulty was hard, the good heuristic saved much more on space
- Easier problems did not cause issues for Uniform Cost Search, but we cannot always rely on having easy problems to solve
- Euclidean Distance saved more space with explored nodes than the other two algorithms

Conclusion:

Analyzing the data from the three algorithms on our testing dataset, we can come to a few conclusions:

- The best algorithm performance is the Euclidean distance followed by Misplaced Tiles and then finally Uniform Cost Search.
- Since Uniform Cost search had the heuristic value hardcoded to zero, its runtime and space complexity was exponential. O(b^d). This meant on the Oh Boy Puzzle, the Uniform Cost search didn't get to solve it.
- With the two heuristic function based algorithms, it is clear that they both improved performance. Especially in the Oh Boy Puzzle, where both of them managed to finish the puzzle.
    - Misplaced Tiles had a weaker heuristic so it finished much slower than Euclidean Distance
- With these observations and data, I can conclude that a weak heuristic(Misplaced Tiles) is better than no heuristic(Uniform Cost Search). But a stronger heuristic(Euclidean Distance), that fits the problem better will perform better than a weak heuristic. This means that heuristics for problems are not equivalent, and some will be better suited for the problem than others. It is important to pick one that tailors to the problem for the best performance.

Challenges Faced:

- Difficulty coming up with great design for project
- Choosing underlying data structures for handling nodes
- Using graph search(keeping set of explored nodes), it took more memory to hold the nodes and time to search through the set

For this assignment, I used:

- Search algorithms provided in the additional readings from iLearn
- Project was coded in Python 3.7, used this Python 3.8 documentation for basic functionalities: https://docs.python.org/3/
- Used this youtube video to get euclidean distance formula: https://www.youtube.com/watch?v=p3HbBlcXDTE
- Small code example tutorials for python was found on https://www.geeksforgeeks.org/
- Priority queue was implemented from priorityqueue library in python
- Explored list was implemented with set()
- Used deepcopy to modify the correct nodes/states

Example to Trace:

$$M = \begin{bmatrix} 1 & 0 & 3 \\ 4 & 2 & 6 \\ 7 & 5 & 8 \end{bmatrix}$$

```
Welcome to 862050241 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 0 3
Enter the second row, use space or tabs between numbers  4 2 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 Euclidean distance heuristic
3

Expanding state
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

The best state to expand with g(n) = 1 h(n) = 2.0 is...
The zero moved down from the previous state.
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Expanding this node

The best state to expand with g(n) = 2 h(n) = 1.0 is...
The zero moved down from the previous state.
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Expanding this node

The best state to expand with g(n) = 3 h(n) = 0.0 is...
The zero moved right from the previous state.
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

You have reached the goal.
GOAL!
To solve this problem, the search algorithm expanded a total of 3 nodes.
The maximum number of nodes in the queue at any one time: 6
A* with the Euclidean distance heuristic 0.0009794235229492188 seconds.

Process finished with exit code 0
```