Project 1: The Eight Puzzle

1. Group Members:

Student Name: Aarav Sane
SID: 862308345
Lecture Section 001

Student Name: Aadil Umar
SID: 862270822
Lecture Section 001

Student Name: Angel Ibanez
SID: 862177056
Lecture Section 001

Student Name: Jacob Rozenkrants
SID: 862350684
Lecture Section 001

2. Contributions:

Aarav Sane:

- Set up Github Repository
- Prepared Code Design
- Uniform Cost

Aadil Umar:

- Created Euclidean
- Worked on Uniform Cost

Angel Ibanez:

- Created Interface in main.py
- Worked on Uniform Cost

- Report

Jacob Rozenkrants:

- Created Misplaced Tile
- Polished uniform cost

3. Challenges:

- When implementing euclidean distance a challenge we came across was that it was difficult to find which index a number belonged to.
- Another challenge we encountered was designing our program as generic as possible in order to easily change to a 15 or 25 sized puzzle.
- We also had issues while keeping track of the explored states, as we represented states by a list of lists and since lists are unhashable in Python, we had to convert them to tuples to hash them. We also tried sets, but they wouldn't work as a set would just sort the elements and thus it wouldn't correctly keep track of the state.

4. Code Design:

Github Repository:
https://github.com/skyguy-netizen/CS170-Project

main.py:
This follows the format and prompts the user to enter the puzzle and the program creates the goal state. Next, the program prompts the user to decide which algorithm to use, with the options being the uniform cost search, the A* with misplaced tile heuristic, and the euclidean distance heuristic; finally the user can also choose to see the trace of the algorithm.

Node.py:
The initialization of the node includes the state (current puzzle configuration), the parent, the operator (whatever move that was made to reach this state), the current depth of the node, and whatever heuristic is being used. There are two overloaded comparison operators: the equality check and the less than check which simply compare the nodes. The find function is a helper

function that locates a target element in order to determine possible moves. There are also print functions which include printing the current state of the puzzle and printing a description of the move that led to the current state.

operators.py:
The operators describe the possible actions that can be taken when trying to make new puzzle states. In this project, there was up, down, left, and right to describe how we could move the puzzle pieces to generate new states. This simply moves the blank puzzle piece into a new position, and the branching factor depends on whether the tile is in a corner, an edge piece, in the middle, or if it will be a repeat. A deep copy is made and checks to see if the move will be valid. The neighborsNode function uses the find function to locate the blank tile and will call the actual operators to generate the neighbor nodes to create new states.

Problem.py:
This contains the implementation of the solution to the problem. The Problem class keeps track of the initial state, the goal state, whether a heuristic is being used, and the trace. The search algorithm is the uniform cost search algorithm and it is implemented such that it doesn't have to make use of the heuristic unless the user has specified that they want to see the misplaced tile heuristic or the euclidean distance heuristic. The search algorithm uses a queue to manage the frontier and uses a dictionary to keep track of the explored states to avoid revisiting them. Furthermore, the algorithm continues until it has reached the goal state or exhausted all states. If the goal state is found, there is a print method which will reconstruct and print the path from the initial to the goal state.

graph.py:
This file simply allowed us to graph our work so that our work could be in a presentable format for this report, and we made use of the plt() function.

5. Questions:

- Did you try optimizing your code by using special data structures?

A. The special data structure we used to try and optimize our code was

using a map in order to not expand repeated nodes.

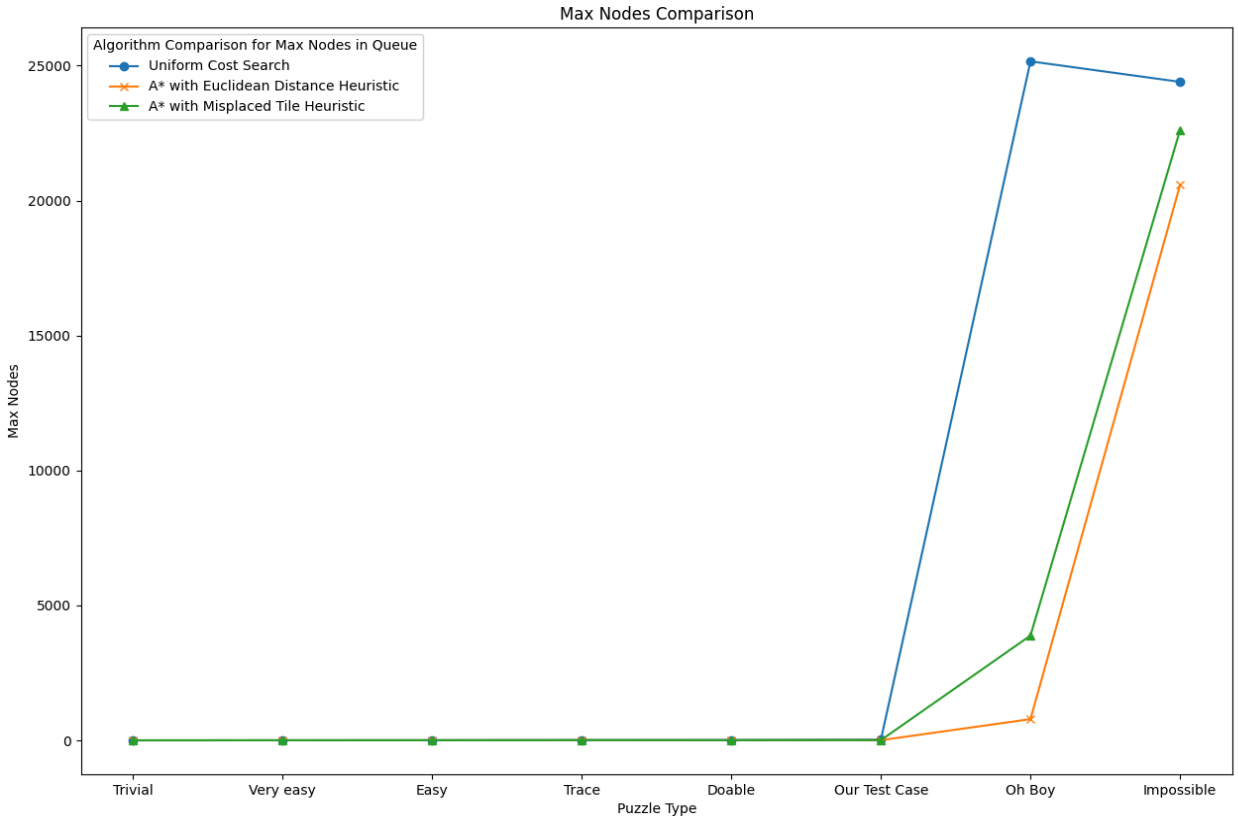
- Did you implement a graph search or a tree search?

A. We implemented a graph search to keep track of our explored nodes for our program.

6. Comparing Heuristics:

- Max Nodes in Queue comparison (Space Complexity)

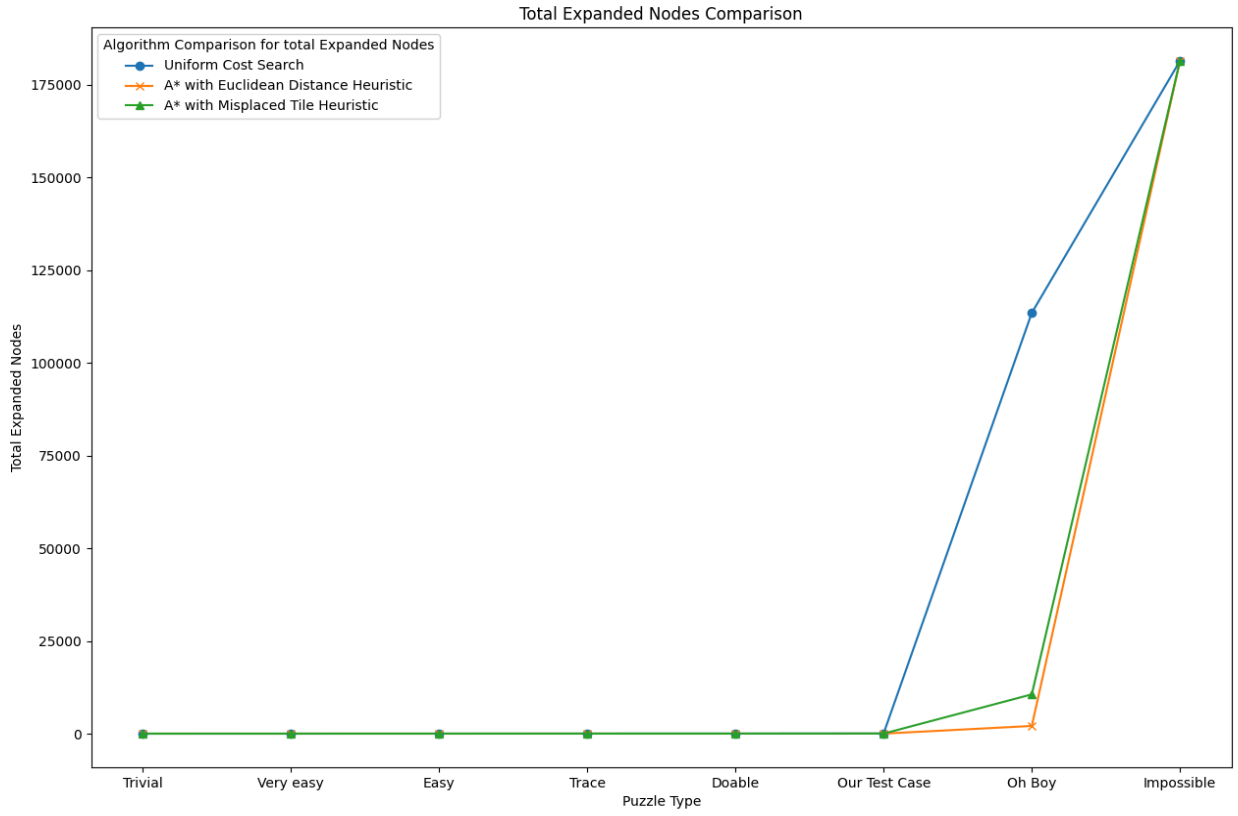
	Uniform Cost Search	A* Misplaced	A* Euclidean
Trivial	1	1	1
Very Easy	3	3	3
Easy	8	3	3
Doable	16	8	4
Trace	14	6	6
Our Test Case	28	9	7
Oh Boy	25159	3884	787
Impossible	24399	22611	20577



- Number of Total expanded Nodes (Time complexity)

	Uniform Cost Search	A* Euclidean	A* Misplaced
Trivial	1	1	1
Very Easy	4	4	4
Easy	13	5	5
Doable	32	4	8
Trace	27	9	9
Our Test Case	61	15	12

Oh Boy	113494	10595	2076
Impossible	181440	181440	181440



For the **Uniform Cost** heuristic:

- It performs adequately for simpler puzzles like "Trivial" and "Very easy," with low maximum queue sizes and explored nodes.
- However, for more complex puzzles like "Trace" and "Oh Boy," it struggles significantly, with high maximum queue sizes and explored nodes.

For the **A* Misplaced Tile Heuristic**:

- It performs better than Uniform Cost, with generally lower maximum queue sizes and explored nodes across all puzzle difficulties.
- It shows improved performance for all puzzles, including "Trace" and "Oh Boy"

For the **A* Euclidean Heuristic**:

- Similar to A* Misplaced, it outperforms Uniform Cost, especially for more challenging puzzles.
- It generally has lower maximum queue sizes and explored nodes compared to A* Misplaced, indicating slightly better performance.

In summary:

- The **Uniform Cost** heuristic is simple but struggles with more complex puzzles.
- **A* Misplaced** and ***A* Euclidean*** heuristics show significant improvements, especially for challenging puzzles, with **A* Euclidean** having an advantage over **A* Misplaced**

8. Trace:

Initial State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

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

The best state to expand with g(n) = 1 and h(n) = 2 is...
Move 2 up
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

The best state to expand with g(n) = 2 and h(n) = 1 is...
Move 5 up
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

The best state to expand with g(n) = 3 and h(n) = 0 is...
Move 8 left
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Solution found at depth 5
Printing solution
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

Move 2 up
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Move 5 up
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Move 8 left

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

Max queue size: 6
Nodes expanded: 9