Project 1: The 8 Puzzles

Project Collaborators: Ainaz Estiri, Jacob Cunningham, Binh Le, Billy Chau

Language: Python

Challenges: The main challenge was figuring out how to create an initial foundation for the program to build upon and having it play the 8 puzzles game. To solve this challenge, we defined the game state as a 3x3 matrix and each element within that matrix was a tile. This had led to the big problem of figuring out how to implement a logic to check if a move was valid and that a “tile” within the matrix can only be moved to another tile element if it was empty which we represented as 0. How we ended up implementing this task was by creating different move functions that switch the positions of two elements within the matrix only if the value it is trying to switch with is equal to 0. This created the main foundation of the 8 Puzzles program in which we can test different search algorithms on.

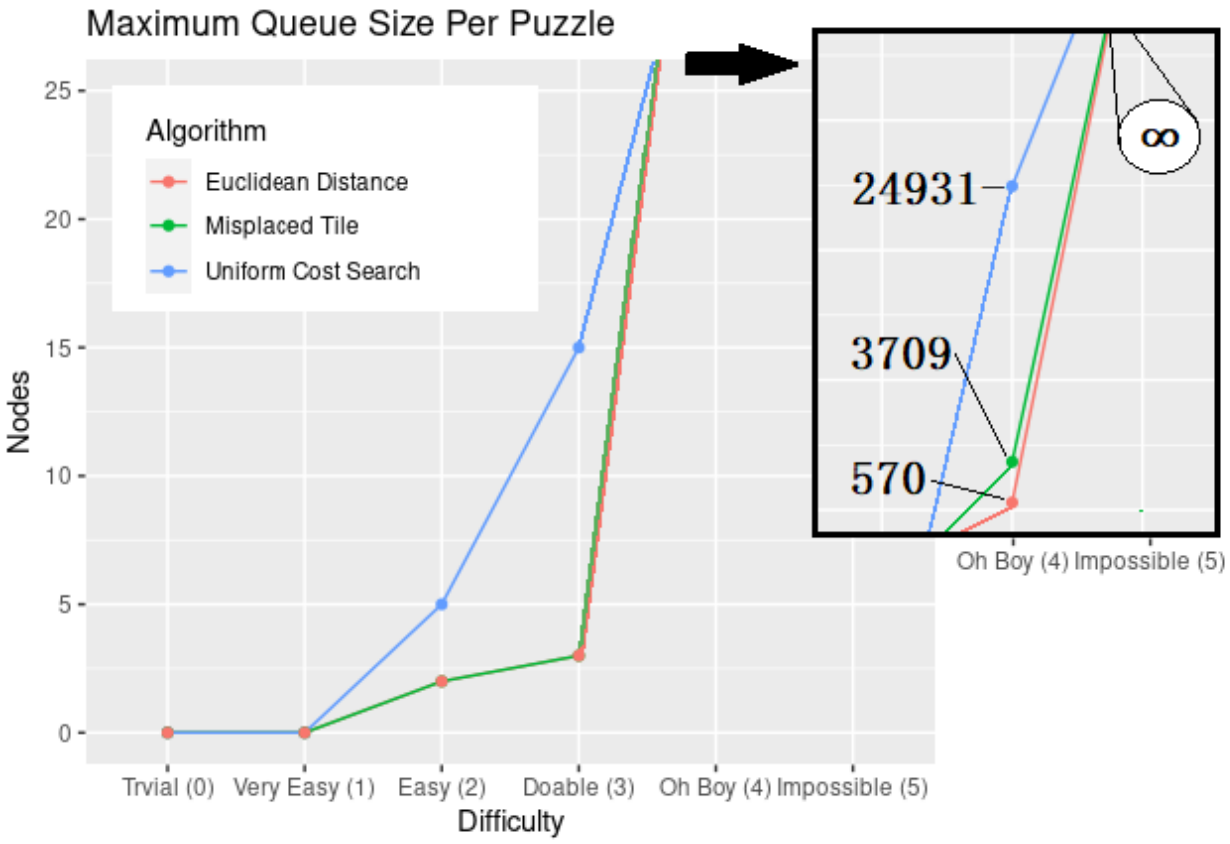
Object Oriented Design: For our design, we created an object oriented program where we had several functions and moving pieces interact together such as:

1. We created a GameState object that represents a universal state of the 8 puzzle board.
    a. GameState() takes a 2 dimensional list using 0 to represent the blank space, and we have a goalTest() method that returns true if we reach the goal state.
    b. We created a GameState object that represents a universal state of the 8 puzzle board.
    c. GameState() takes a 2 dimensional list using 0 to represent the blank space, and we have a goalTest() method that returns true if we reach the goal state.
    d. Used a priority queue for our searching algorithms.
2. We created three heuristic functions that take a GameState as input and return an estimate of the distance to the goal.
3. We created a Search object that takes a GameState starting board and h heuristic function and executes the search.
    a. Used a priority queue for our searching algorithms.
    b. Has a search method that performs the search and returns the number of nodes expanded, maximum number of nodes in the priority queue, and depth of the solution.
4. By separating the search, state, and heuristic we make our code easier to read and update later with different heuristics, search algorithms, and problem types.

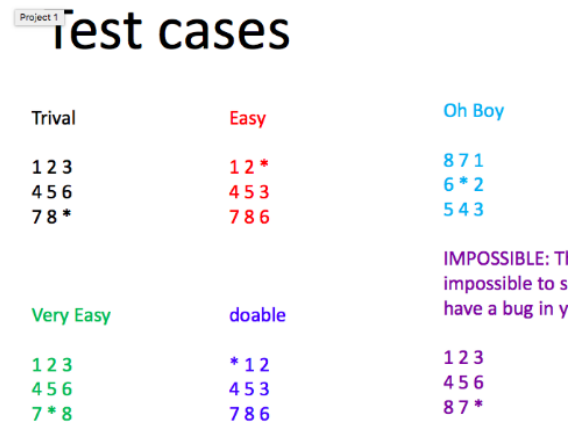
Optimization: We created a class search that represents one universal/possible search, to do the 3 different searching methods

- We chose to use an array to represent the problem state. Since the main operation on the problem state is swapping two cells, the constant time access of an array makes our algorithm efficient.
- The main operations of search are to get the state with the lowest priority from the frontier and to add a new state to the frontier. We use python's priority queue implemented as a binary heap to represent the frontier. This is efficient because binary heaps have a logarithmic insertion time and constant access time.
    - Source: https://www.educative.io/answers/what-is-the-python-priority-queue
- Our operators: moveLeft, moveRight, moveUp, moveDown are universal at each position, we determine which moves are acceptable/possible with the getMoves() function. And we use the 3 algorithm searches (Uniform cost, MissingTile, and Euclidean search) plus h() as the heuristic offset to decide on the best possible move/operator for any puzzle.

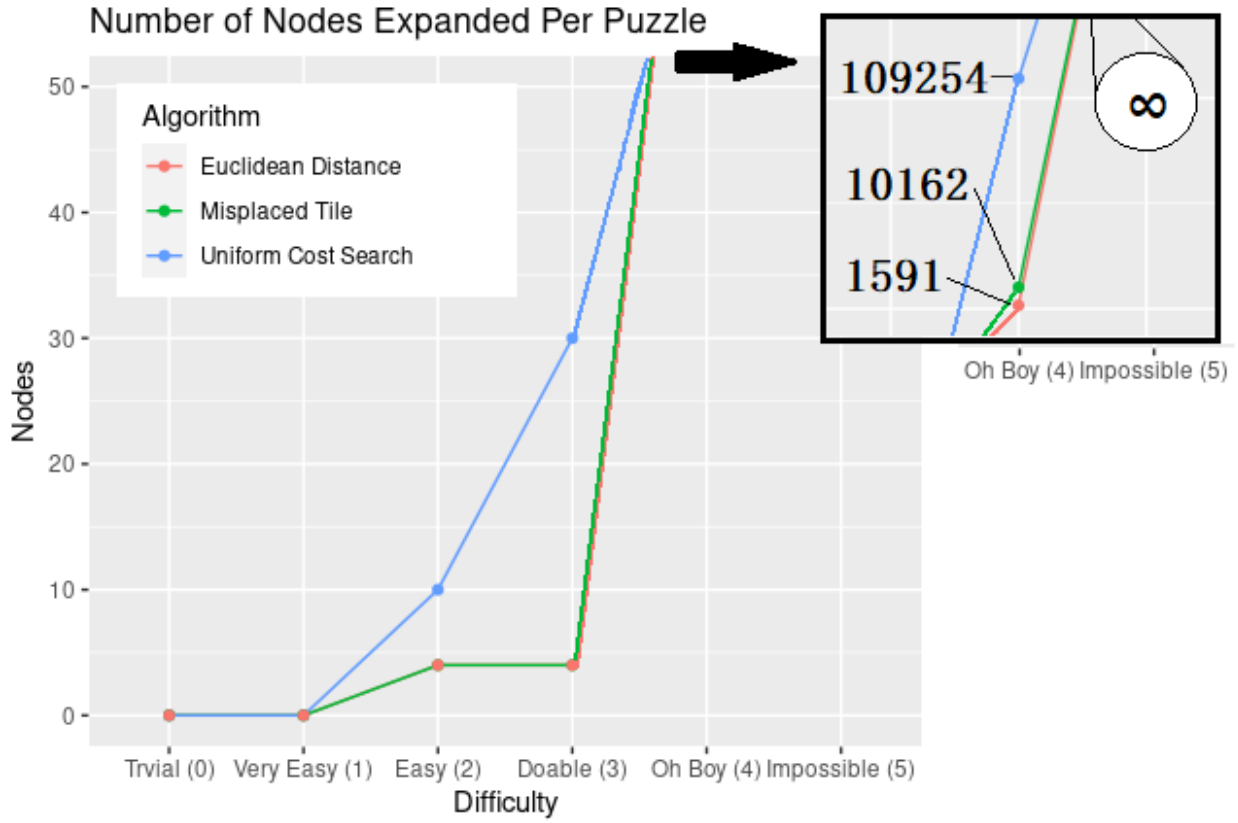
Graph Search: We implemented a graph search in that our program is able to record and keep track of all explored nodes of the selected algorithm as well as the maximum queue size of each puzzle. We compare the aforementioned in the following:



Difficulty <chr>	Uniform Cost Search <chr>	Misplaced Tile <chr>	Euclidean Distance <chr>
Trvial (0)	0	0	0
Very Easy (1)	0	0	0
Easy (2)	5	2	2
Doable (3)	15	3	3
Oh Boy (4)	24931	3709	570
Impossible (5)	∞	∞	∞



We applied the following test cases with each initial state level increasing in difficulty. The following graph shows the maximum queue size of the graphs showing the efficiency of each algorithm. We can clearly see that Uniform Cost Search is the least efficient, and Misplaced Tile and Euclidean Distance are very similar in efficiency up to the point in level 4 where Euclidean Distance will come out on top.



Difficulty <chr>	Uniform Cost Search <chr>	Misplaced Tile <chr>	Euclidean Distance <chr>
Trvial (0)	0	0	0
Very Easy (1)	0	0	0
Easy (2)	2	2	2
Doable (3)	4	4	4
Oh Boy (4)	109254	10162	1591
Impossible (5)	∞	∞	∞

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: Tl
impossible to s
have a bug in y

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

Utilizing the same test cases. We also kept track of the number of nodes expanded for each puzzle and test cases. We can see in the graph the reason for Uniform Cost Search's inefficiency is its tendency to expand more nodes than is necessary to solve the puzzle. Similarly, Misplaced Tile and Euclidean Distance have very close values up to the point of level 4 where Euclidean Distance gains the upper hand in efficiency. This is because the first examples require only zero or one move to solve, so both heuristics return the same value.