Gwendolyn Kiler
SID: 862208140

CS170 8 Puzzle Report

Sources I Used:
Just python documentation from docs.python.org for basic class functionality. Use of numpy.flatten and arrays was just from prior knowledge.

Challenges in this project:

I didn't get stuck on anything in this project in particular. In the end, I had to do bug fixing for general type errors in my untested code, but that was just from lack of knowledge using classes and object oriented programming in python. The part that took the longest was probably charting out the preliminary design, but the project doc suggested using a node class and a problem class, so I just went from there using the suggested pseudocode step by step in A-Star's coding, with no significant change between A-Star and Uniform Cost as Uniform Cost is just a flat cost, whereas A-Star uses heuristics.

In the end, it was just a loop of initial node -> check possible moves -> check if moves were seen before -> expand non seen moves -> add to queue -> repeat while checking goal state. I also know the algorithms are optimal since the given heuristics are optimistic/overestimate the cost, and the first goal found will be the lowest cost in the priority queue, meaning all other solutions have a higher cost in the heuristic.

Optimization:

I think the two most significant speed ups I used in the design of this project were numpy arrays and hashing the arrays to be used in a set.

Python set checking is O(1) time, so it doesn't slow down with massive tree sizes. To accomplish this, I initially tried to use the method tostr() but the debugger told me tostr() is deprecated and to use tobytes(), so I did and it worked well.

Numpy arrays in general are just more optimized and more efficient than default python vectors, so using 2D arrays for states likely saves memory and time.

Using a priority queue will also efficiently sort by node cost by overloading the less than operator, which speeds up each node retrieval by a lot.

When testing the code, the "Oh Boy" test case only takes a few seconds, and going through all states for "IMPOSSIBLE" to find no solution only takes a few minutes at best, and 10 minutes approx. for uniform cost search at worst.

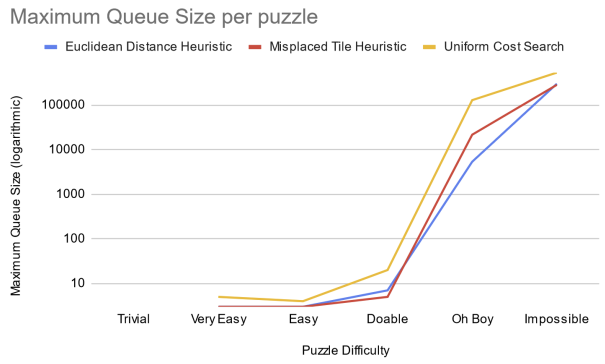
Tree Search

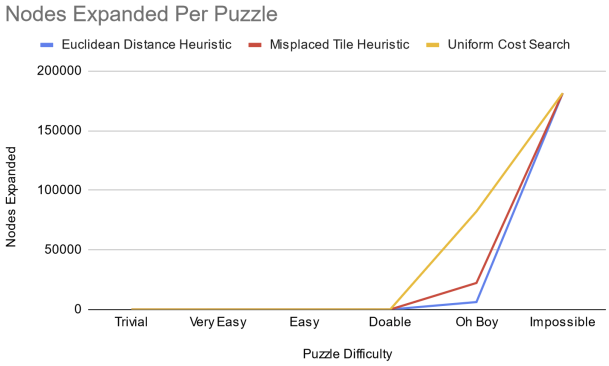
I implemented tree search as it seemed far simpler than graph search. All I needed to implement was parent node tracking for solution tracing, which is just a class variable, save the game state, save the action taken, and use one of the three heuristics added with parent cost to get self cost. I only implemented tree search so I don't have any comparative observations.

Findings

To support my findings, in the following section are graphs and tables for each heuristic and their time/space complexity comparisons. There is little difference between each heuristic at the beginning, with trivial being 0 size/time and very easy only having a difference of 1 or 2 queue size/nodes. As difficulty increases, heuristic is more important as for “Oh Boy”, Euclidean Distance has 4x less space used as max queue size than Misplaced Tile which has 6 times less space than Uniform Cost. Similarly, for time spent as nodes expanded in “Oh Boy”, Euclidean Distance is 4x faster than Misplaced Tile which itself is 4x faster than Uniform Cost Search. For the upper limit of “Impossible”, node expansion is the same as all possible nodes are expanded. For max queue size, Euclidean Distance and Misplaced Tile are about the same, while Uniform Cost is approx 2x more space.

GitHub: Here is proof of version control with my code and commits: https://github.com/gkiler/CS170





Max Queue Size			
Puzzle Difficulty	Euclidean Distance He	Misplaced Tile Heuristic	Uniform Cost Search
Trivial	0	0	0
Very Easy	3	3	5
Easy	3	3	4
Doable	7	5	20
Oh Boy	5355	21587	128405
Impossible	299061	282867	531844

Nodes Expanded			
Puzzle Difficulty	Euclidean Distance He	Misplaced Tile Heuristic	Uniform Cost Search
Trivial	0	0	0
Very Easy	1	1	2
Easy	2	2	3
Doable	6	5	30
Oh Boy	6240	22262	82219
Impossible	181440	181440	181440