Ann-Marina Miyaguchi
CS170 Sect 2

Project 1 A* Search - Project Report

Challenges with project

1. **Circular Dependency**
   It was challenging to decide on what classes I would use for the project. I started off with the Board and Tree class and once I got that working I started brainstorming ways to get Problem and AStar to interface with them. Since I made so many classes I had a bit of difficulty getting the expand function to work properly. My AStar class contained the frontier queue and explored set and included the Problem class, so having my expand state function in the Problem class created a challenge since having the Problem header also include the file to AStar created a circular dependency. In order to fix this issue, I got rid of any inclusions from AStar in Problem. I also placed the main expand function in AStar so that it would have access to both the frontier queue and explored set as well as the expand up, down, right, and left functions.
2. **Fixing comparisons**
   Beyond this, getting A Star Search to fail was surprisingly difficult. Rather than returning nullptr, my function preferred to keep expanding nodes. After a lot of trial and error, I eventually figured out that my if statements to check if a state was in the explored set or frontier set was wrong. Rather than checking that a specific priority and Board were in the frontier or explored set, I needed to instead only check that a board (the 2d array) was inside.

Optimizations

1. **Creating the frontier set**
   For A Star we have a tree that can have at most 4 children. This is because we can expand a state in at most 4 directions: up, down, right and left. Since we only wanted to expand when a state was NOT in the explored set or frontier queue, I also created a frontier set which kept track of what nodes were in the frontier queue. This way I wouldn't insert a node that was already explored or was already put in the frontier queue. The same board (matrix) could potentially be included twice however, in that case they would have different priorities. For every item pushed into the frontier queue, the same item is inserted into the frontier set. When items are removed from the frontier queue, the same items are removed from the frontier set. In this way it makes searching for if a node is already in the frontier queue or explored set much easier.

Design

There are four classes: Board, Tree, Problem, and AStar. Board is essentially the node class and Tree is what allows us to more easily interface with Board. Problem contains the setup with the initial and goal state as well as 4 functions/operations to expand up, down, right and left that return the 2D array given a Board. AStar has the actual AStar implementation with the heuristic functions and the actual expand state function that uses the 4 expand functions from Problem.

Graph vs Tree Implementations

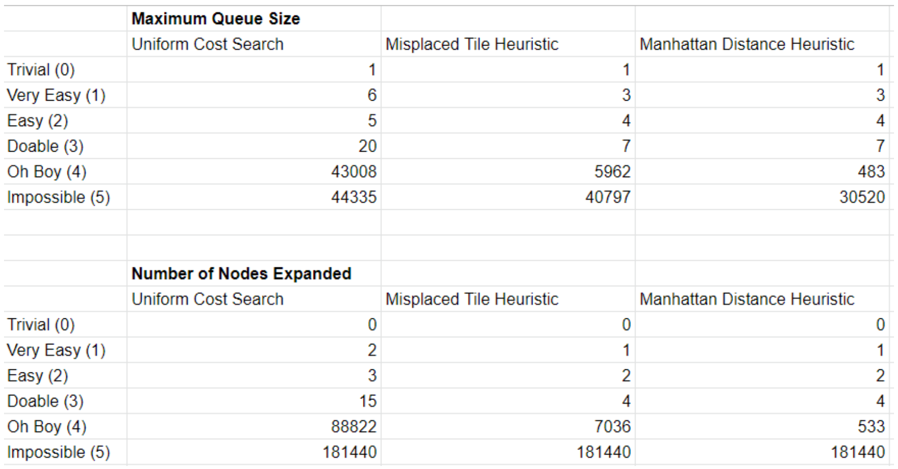
I used a tree implementation to keep track of where each state was in relation to each other. Using the tree implementation I was able to print the path taken by A* search as more easily add nodes to the frontier queue. I used a parent pointer to help with printing the path taken by A* search, in a recursive function print_path. Since my Board class keeps track of the depth, I'm also able to quickly get the depth at a specific node.

Comparing Heuristic Functions

Test Cases and results:

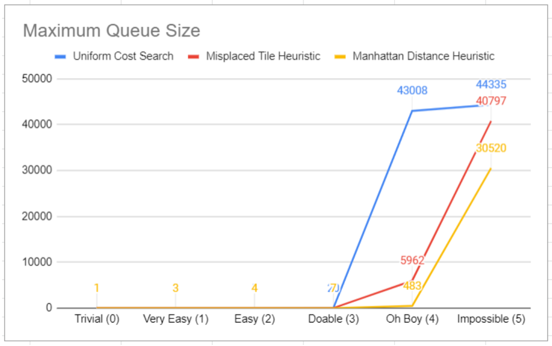
Trivial (0)	Easy (2)	Oh Boy (4)
1 2 3	1 2 0	8 7 1
4 5 6	4 5 3	6 0 2
7 8 0	7 8 6	5 4 3

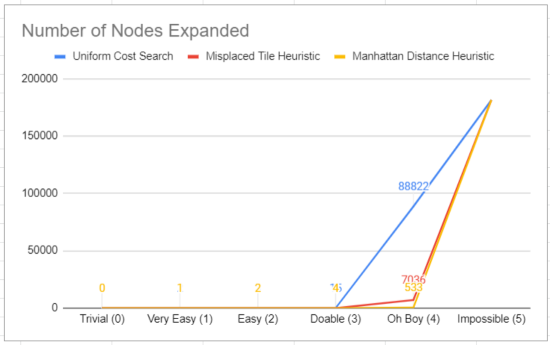
Very Easy (1)	Doable (3)	Impossible (5)
1 2 3	0 1 2	1 2 3
4 5 6	4 5 3	4 5 6
7 0 8	7 8 6	8 7 0

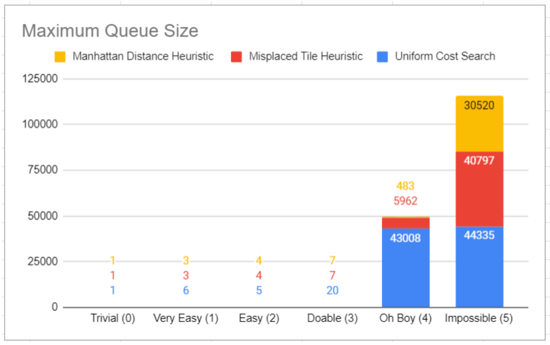


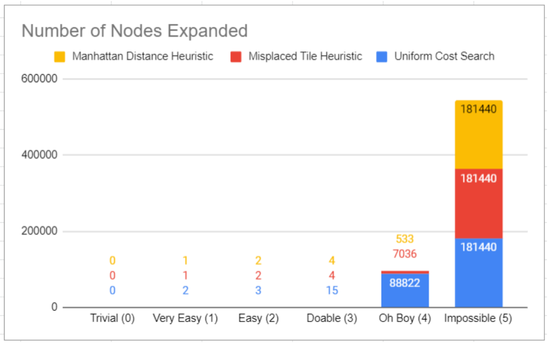
	Maximum Queue Size		
	Uniform Cost Search	Misplaced Tile Heuristic	Manhattan Distance Heuristic
Trivial (0)	1	1	1
Very Easy (1)	6	3	3
Easy (2)	5	4	4
Doable (3)	20	7	7
Oh Boy (4)	43008	5962	483
Impossible (5)	44335	40797	30520

	Number of Nodes Expanded		
	Uniform Cost Search	Misplaced Tile Heuristic	Manhattan Distance Heuristic
Trivial (0)	0	0	0
Very Easy (1)	2	1	1
Easy (2)	3	2	2
Doable (3)	15	4	4
Oh Boy (4)	88822	7036	533
Impossible (5)	181440	181440	181440









Euclidean with more complex solutions/no solutions, ends up expanding far fewer nodes and also has a smaller queue than the other two. In most cases, Uniform Cost Search performs significantly worse than both Euclidean and Misplaced Tile.