Charles Hong

Project 1: 8 Puzzle Solver with A* Algorithm

Challenges

I had some challenges figuring out how to store the board without making the code seem too messy. I decided to go with a one dimensional size 9 vector, instead of a 3x3 two dimensional vector, since the code may look too clumped by declaring a 2d vector. The challenging part is when I needed to figure out how to calculate the Manhattan/Euclidean distance. I spent quite some time figuring out how to properly extract the x and y coordinates of the current state and the goal state. The 0-indexed vectors added some more confusion to my debugging. This became extremely difficult when I had to figure out the operation of moving the blank around. I needed to check the bounds so that the blank doesn't go out of bounds, and if it is within range, it needs to move to the correct spot.

Another challenge that I had is to properly add the appropriate g(n) and h(n) to the right node, and make sure that I can add the correct g and h values. I didn't realize that I had to add the g(n) after traversing to deeper nodes. I thought that since every cost is 1, so I didn't include it. However, I fixed it by running the code a few times, and figured out that I needed to keep track of a g(n) value.

Design

In my design, I have two classes. Node and Problem.

Node:

vector<int> board: A vector of size 9 that stores the state of the game board.

Constructor: creates the board from input parameter, and uses the heuristic calculation helper function

calcHeur(): Based on the option of the heuristic choice, it returns the estimated number of steps to reach the goal state.

Problem:

Node* init,goal,curr: initial state, goal state, and the traversing state.

vector<pair<int,int>> move: a size 4 vector that helps writing cleaner code for the operation of moving the blank 0 around.

set<pair<double,pair<int,Node*>>> pq; //a minimum priority queue frontier that uses the std::set data structure. It uses the h(n)+g(n) value to order its Nodes. The pair inside represents g(n) and the Node that is linked to g(n).

set<vector<int>> visited: keeps track of the expanded nodes.

Constructor: initializes the class variables.

void expand(): create and insert the current node's neighbors into the frontier if those neighbors have not been expanded yet.

bool findGoal(): It runs the A* algorithm and returns whether the solution has been found or not. Utilizes the expand function.

Optimization

I used a std::set as my frontier. Even though the insertion and deletion cost is both O(logn), the std::priority_queue probably would be a better choice. However, I used a set because it is easier to implement as a minimum frontier, since priority_queue by default is a max heap, I didn't want to implement a custom operator. It is safer this way.

I also used std::set to store the nodes that I have already expanded. Checking time is O(logn), which is faster than linearly searching through a list of expanded nodes.

I also passed the Nodes as pointers so that it doesn't create a copy of it every time a function uses it. It can just dereference quickly with the help of pointers.

Comparisons and Findings

The heuristic functions worked better than I thought they would. Here are the graphed results. I used the test cases given in the guidelines. Disregarding the Manhattan distance heuristic, the Euclidean distance heuristic seems to be the best choice for this assignment. It has the least visited nodes and the smallest frontier size. In the impossible case, none of the algorithm worked, which resulted in expanding all possibilities: 181440, which is equal to 9!/2, which is really cool since there are 9 spots on the board, so there is a possibility of 9! combinations. Exactly half of it will have one solution, and the other half of 9! Will have a different combination.

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

Very Easy

1 2 3
4 5 6
7 * 8

doable

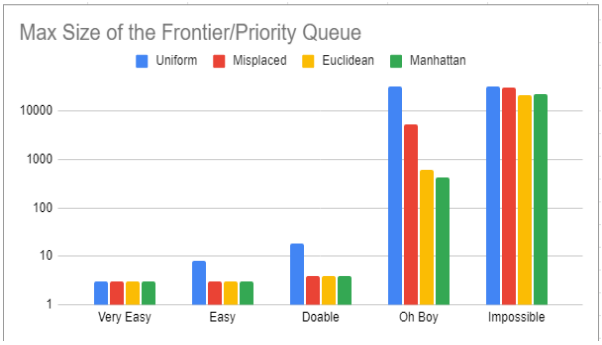
* 1 2
4 5 3
7 8 6

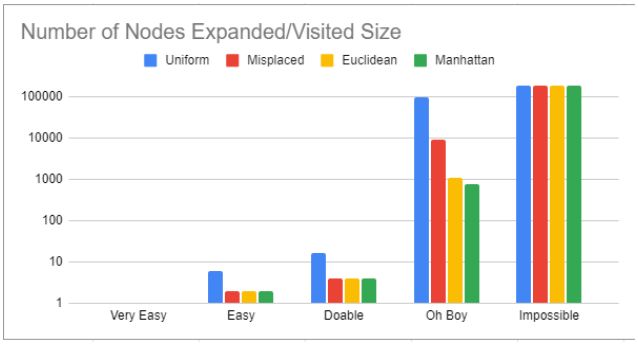
IMPOSSIBLE: impossible to have a bug in

1 2 3
4 5 6
8 7 *

Max Size of the Frontier/Priority Queue				
	Uniform	Misplaced	Euclidean	Manhattan
Very Easy	3	3	3	3
Easy	8	3	3	3
Doable	18	4	4	4
Oh Boy	32271	5336	605	418
Impossible	32757	30165	21689	23131

Number of Nodes Expanded/Visited Size				
	Uniform	Misplaced	Euclidean	Manhattan
Very Easy	1	1	1	1
Easy	6	2	2	2
Doable	16	4	4	4
Oh Boy	98857	9121	1067	741
Impossible	181440	181440	181440	181440





Code: https://github.com/Charles-Hong520/CS170Project1