<u>COP 4516 Spring 2023 Week 5 Individual: DFS/BFS (Solution Sketches)</u>

<u>*Get Out of this Maze!!!*</u>
This problem is a more traditional BFS (than Eight Puzzle). Each of the vertices of the graph are grid squares, and the edges only connect adjacent grid squares up, down, left or right. We can simply create a two-dimensional integer array the same size as the grid to store distances for our BFS, initializing values to -1 (not visited). We want to cut out of our BFS as soon as we reach any border square. Whenever we dequeue a square, we want to process it by enqueuing any unvisited neighbors. For grids, using DX/DY arrays is the best way to iterate through all the neighbors of a square. One example of the DX/DY arrays is shown in the practice problem, Token Game, for the week (http://www.cs.ucf.edu/~dmarino/progcontests/cop4516/ind-contests/dfsbfstopsort/practice/tokengame.java). This reduces the chance of error since you don't have to do bounds checks four times over. (Though, in this problem you can get away without doing bounds checks because the whole border is guaranteed to be tilde characters…) One other key detail about a BFS is that you <u>**have to mark the distance when you enqueue an item, not when you dequeue it.**</u>

<u>*8 Puzzle*</u>
This is a classic game BFS. For each board position, we can calculate the other reachable board positions in one move. Then, for each position, we must store the distance (fewest number of moves to reach it), from the starting board. Instead of storing this distance in an array, it's easier to store it in a HashMap. Two ideas for the HashMap are as follows: (1) Store each board position as a string of 9 characters '0' - '8', where '0' is the empty square. (2) Store each board position as a single integer of 9 digits, again treating the empty square as 0. Thus, it makes sense to have a function that takes in the 2D array board and turns it into the string/integer or vice versa. Or, alternatively, just do all the index math and swap positions in the string or integer.

So, even after doing everything stated above, if you were to submit, you get Time Limit Exceeded. If you do the math, you find out that a BFS to solve one puzzle might take close to 4 x 9! steps. (This is because there are a maximum of 9! possible board positions to reach, and you have to check out 4 possible moves when exploring from each board position.) More accurately, since half of the board positions aren't reachable, it's really close to 2 x 9! steps. This is roughly a half million steps, which is fine for a single case. But, the input format explicitly states the following: "The first line of input will contain an integer N (1 ≤ N ≤ 10000) which represents the number of test cases. So, now, our total run time is 10000 x 2 x 9! ~ 7 x 10^9 steps, which is way too many for a programming contest.

But, one key realization is this: <u>***every move while solving a puzzle is reversible!!!***</u> So, what if, instead of starting at the initial puzzle, we start at a solved board, and try to make moves to all other board positions? Doing so is a single BFS. But in that BFS, we will find how far away each board position is from the <u>***ending board position.***</u> Thus, the key to solving this problem is the following:

Before you read in the input, run a <u>**SINGLE**</u> BFS from the ending board position to all other reachable positions. Store the distances in a HashMap as mentioned previously.

Then, when you read in the input, just provide the answer stored in the HashMap for how far from the solution board each input board is.

This is a critical idea in many shortest distance problems - if you want the shortest distance from many different places to one end place AND the edges are undirected, then flip your search and start from the ending spot and in one fell swoop you'll get all the distances from that place to all possible beginning locations, which is what you want!!!

Same Letters

Many ways to solve this using elementary techniques. The standard way would be to use a frequency array of size 26 keeping track of how many of each letter is in each word and then compare the frequency arrays. An even easier way would be to store each word as a character array, sort them using an API call, and then check if the two resultant strings are equal or not (another API call).

Interstellar Love

Each constellation is simply a connected component with 2 or more vertices. Thus, running either a DFS or BFS on every unvisited vertex will identify the total number of connected components and it's fairly easy to skip over vertices not connected to any other vertex. To determine whether or not a constellation needs to be fixed, note that a tree of v vertices must have v-1 edges. One way to determine if a component has the proper number of edges would simply be to keep a degree count for each vertex as you read it in and store this in an array. Then, when marking a component, add up the degree counts of each vertex in the component. Note that the sum of degree counts in a component is equal to twice the number of edges in the component, since each edge adds 2 total to the sum of degree counts. So, if the sum of these degree counts in a component equals 2v-2, then it does not need fixing. Otherwise, it does. One way to implement this instead of using the type boolean for the visited array, use int and when a vertex is marked, store its component number. Then, this array neatly stores all vertices that are in the same component, if you sweep through it again.