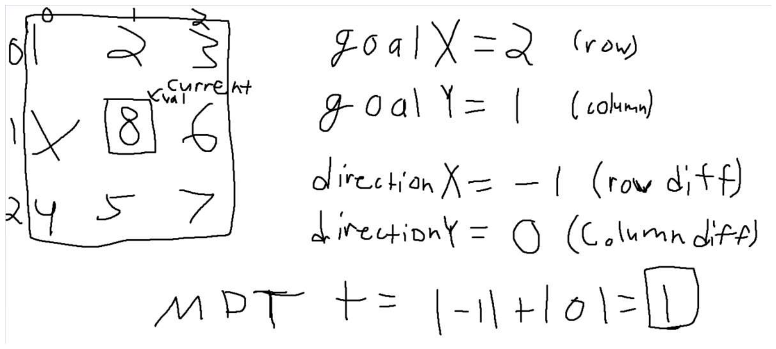
Write-Up

1. The heuristic implemented for my A* search was the Manhattan Distance. I pass in my root node along with the size of the puzzle (used for traversal). I create an int variable to keep track of my total Manhattan distance initialized to 0 each time the method is called.
    - ➔ To calculate the total Manhattan distance, I created two integers ‘goalX’ and ‘goalY’. These variables will get the row and column for the correct location using the current value. Then, created two more variables, ‘directionX’ and ‘directionY’. These variables took care of the distance from the current value to its correct location.

    Go through a nested for loop of size ‘size’ to go through the number of elements in the puzzle. Create an int ‘currentVal’ which will grab the current value located in my root Node. Node class contains a 2-D array which I called.

    - ➔ Now, make sure the current value is not equal(!=) to 0. This is because we do not calculate the 0 value. Once we check this ‘goalX = (currentVal – 1) / size’ will get our row for the current integer’s correct location. ‘goalY = (currentVal – 1) % size’ will get our column for the current integer’s correct location. Afterwards we will get the distances from the current value to the correct ‘goal’ location. ‘directionX = i – goalX’ will get the row distance. ‘directionY = j – goalY’ will get the column distance.

To finish up just set the total Manhattan distance variable += the absolute value of your directionX + absolute value of your directionY. Now you will know the total distance from your current value to the goal spot. Do this for each value in the puzzle and when finished return your total Manhattan distance value. *Ex.*



2.

    BFS -> Create a queue “First-in-First out”, add your root node to the queue to perform BFS. If the que is not empty and we have not found the match(Boolean) checks if we found the solution. Remove the root to a new node, create our children for (up, right, down, left) and add each child that exists into the queue. Then we check to see if our 2D array stored in our node class matched with the correct 2D array. Once it does break out of the while loop.

IDDFS -> Like BFS, but we added a max depth level and used a stack "Last-in First out". If the stack is not empty, pop the root to a new node, create the children and push each into the stack.

A* -> Like BFS, now we switch over to a priority queue. To perform A* I used the formula f = g + h. Where g is the depth level, h is the heuristic, and f is the sum of the depth level and heuristic.

BFS vs IDDFS -> IDDFS is better if you know the exact depth. While BFS is better when you do not know the exact depth. This is because IDDFS will continue to search even after the correct depth. Therefore, memory usage is dependent on these factors. Nonetheless, both BFS and IDDFS crash when using the second 4x4 puzzle.

BFS vs A* -> BFS stores our children in a queue (First-in First out). The A* time and space are very dependent on the heuristic we use. BFS and A* can always find a solution, but you need more memory for BFS. We see this in the second 4x4 puzzle how BFS will crash and A* provides us with the solution.

IDDFS vs A* -> A* always provides the most optimal path. IDDFS does not always find the best path. We know depth-first search is good in memory usage and will be lower than A* 's memory, but it focuses only on the best nodes. Thus, not everywhere is searched.

3. Used: IntelliJ
    - ➔ My MainClass.java takes in the text file Maze.txt, located in: 'src' folder. To run a different puzzle, go to your source folder and click on the Maze.txt file and change the puzzle accordingly. The program should then run after hitting the play(green arrow) button.
    - ➔ IterativeDeepeningDFS.java class has a public int depthLimit = 15. This is my max depth that the algorithm will run until. Change the max depth of 15 to whatever depth you want the program to run until.
    - ➔ If you run BFS and it is out of memory it will just stop and not go to other algorithms. So, just comment out BFS in the MainClass and just run IDDFS or A*. Tested on second 4x4 puzzle.

References:

- ➔ https://www.aaai.org/Papers/AAAI/2002/AAAI02-111.pdf
- ➔ http://theoryofprogramming.com/2018/01/14/iterative-deepening-depth-first-search-iddfs/