This project was completed by Ben Walls. In the search class, there is a search function that performs either UCS or A*. I chose to only use one search function because of how similar the search algorithms are. A boolean parameter determines whether or not heuristics are considered in the priority queue, and everything else is the same.

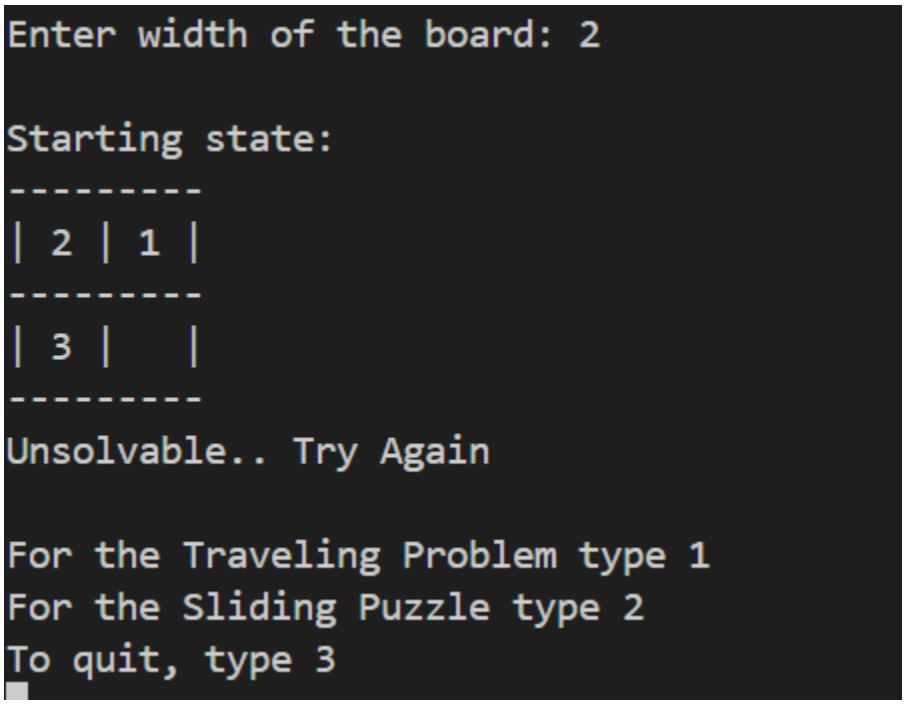
The user starts by running python Search.py. They can type either 1 (for traveling problem), 2 (for sliding puzzle), or 3 (to quit). Upon choosing one of the two problems, the user is prompted by the initialize() function of the respective problem class. After an initial and goal state is generated, the search class runs both UCS and A*.

Output for the traveling problem (Arad -> Bucharest):

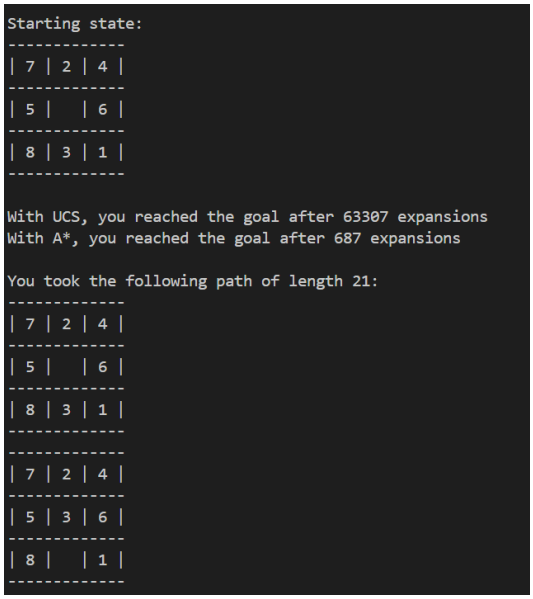
```
Welcome to the searcher
For the Traveling Problem type 1
For the Sliding Puzzle type 2
To quit, type 3
1
Where would you like to start? Arad
Where would you like to go? Bucharest


Starting state:
Arad

With UCS, you reached the goal after 13 expansions
With A*, you reached the goal after 10 expansions

You took the following path of length 4:
Arad
Sibiu
Fagaras
Bucharest
------------
For the Traveling Problem type 1
For the Sliding Puzzle type 2
To quit, type 3
```

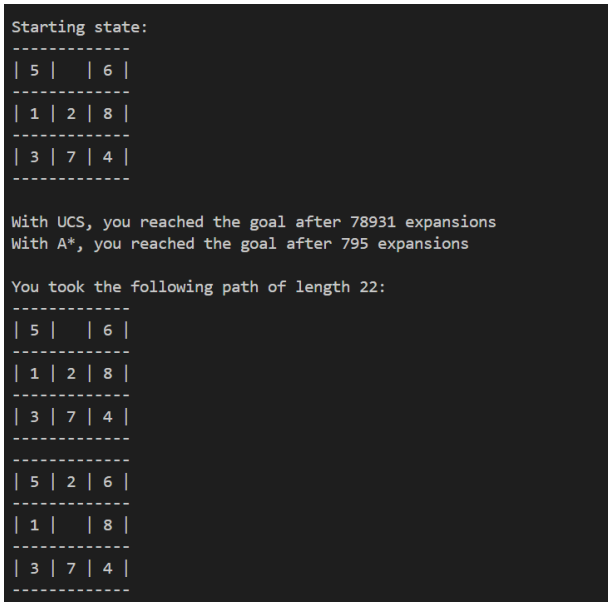
The program reports the starting state, number of expansions for both algorithms, the length of the path, and the path traveled (the path is the same for UCS and A* - this was ensured during testing). Afterwards, the user is continuously prompted with the option to either perform the traveling problem, sliding puzzle, or quit.

For the sliding puzzle, boards are randomly generated, and sometimes these boards are unsolvable. This occurs if the frontier becomes empty before finding the goal state. If an unsolvable board is encountered, the program returns this.



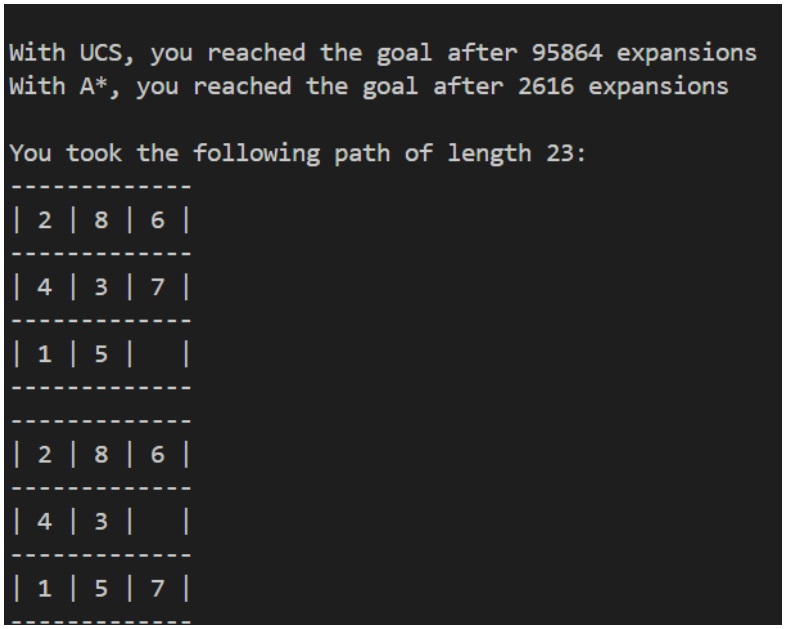
Typically, the program prompts the user for the width of the board, and randomly generates a starting state. Here are two example boards and their results (temporarily hardcoded the examples from the assignment sheet).





Similar to the traveling problem, the sliding puzzle outputs the path traveled, using a function called pretty_print() that prints the current state of the board.

For the above examples, I used the manhattan distance heuristic function in A* search. I also created my own heuristic function. It looks at the current place of the missing tile, and disregards the tiles in the same row and same column. This is because those tiles can be moved, whereas tiles that can't currently be moved probably require more moves to reach their goal location. For these tiles, I simply added the distance away vertically and horizontally from their goal location to the heuristic. Here are the results:



So, using the heuristic significantly decreased the number of expansions needed to reach the goal state.

For the 8-Puzzle, I was surprised at the number of expansions that were made just to find the solution. Also, it was nice to see the use of a heuristic that significantly decreased these expansions. Either way, some of these 3x3 puzzles are almost impossible for a human to solve. This made me realize how powerful these search algorithms can be (and surprisingly fast). All in all, my search algorithm took 1-2 hours to implement. But the entire project took around 6 hours.