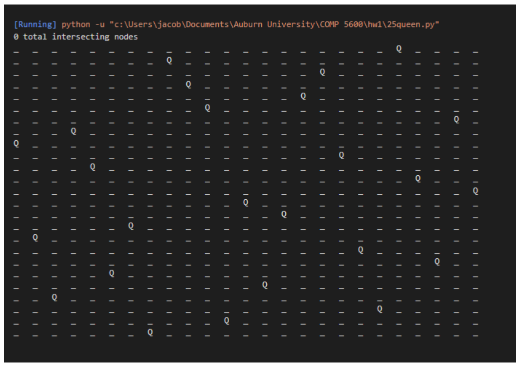
1) See source code in attached file **25queen.py**.

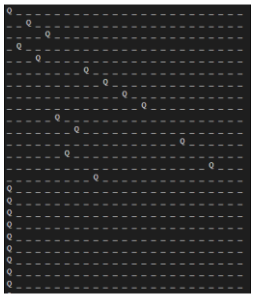
The **score** function is defined in the Queen class. This determines given a queen's spot how many other queens it intersects with horizontally, diagonally, and vertically (note, with this definition, a lower score is better). **Neighbors** are all spaces within a row on the board, therefore when searching for the optimal spot of a queen, it can be placed anywhere along the x-axis, while the y-axis is fixed. The logic behind this is that on an N-sized board with N queens, they should stay within their own lane since otherwise we would be guaranteed a score of 1 at minimum. The **best state** I was able to achieve held **0** total colliding queens. To achieve this state, I took a hill climbing approach with random initial positions, rerun multiple rerun multiple times to collect the optimal result. Upon running the python script, 25 queens are created (each at position (random(0, 24),Y) where Y is the unique Y coordinate, and random(0,24) is a random integer between 0 and 24 inclusive) and mapped (by position) into a two-dimensional array. This map is used to maintain the board structure. From here, each queen, one at a time, evaluates the lowest score placement in its lane, and places the queen into that spot after updating its position. After the hill climbing search is complete, it is re-run again. This is because some of the earlier queens are impacted by the later ones that had not yet been placed. As mentioned, this process is rerun, up to 100 times (with an early exit if a solution with 0 intersecting queens is found).



```
[Running] python -u "c:\Users\jacob\Documents\Auburn University\COMP 5600\hw1\25queen.py"
0 total intersecting nodes
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _
_ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _
_ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _
_ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q
_ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _
_ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _
_ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

The reason this implementation relies on random ordering to solve the problem is due to not identifying an optimal way to solve a tie. For example, in the picture below, if the

4th row queen had determined it should be placed in the 7th column, that would lead to an optimal solution. Unfortunately, both spots resulted in the same score. Random initial ordering mitigates this issue.



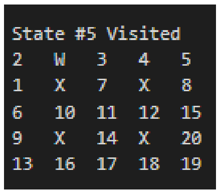
```
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _
_ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Q _ _ _
_ _ _ _ _ _ _ _ _ Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Q _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

2) See source code in attached file **A_star_block_puzzle.py**.

It took **17 steps** to achieve the goal state.

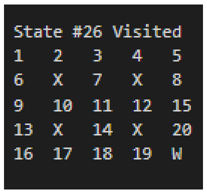
This occurred after going through a total of **30** states.

The 5th state



```
State #5 Visited
2   W   3   4   5
1   X   7   X   8
6   10  11  12  15
9   X   14  X   20
13  16  17  18  19
```

The 5th from last state:



```
State #26 Visited
1   2   3   4   5
6   X   7   X   8
9   10  11  12  15
13  X   14  X   20
16  17  18  19  W
```

(w corresponds to the white space; X's are the black holes)

3) **Theorem**

Given h(s) (the heuristic function) is admissible, A* is optimal

Proof

By definition, for h(s) to be admissible, then the following must be true: h(s) <= h*(s), where h*(s) is the true cost of given node s to the goal.

For A* to be optimal, it means that it must be able to find the best solution. In other words, the solution path from the start node should have cost h*(s), the true least cost to the goal.

For the sake of proof by contradiction, we assume that A* is not optimal, in order to show that in such a world, h(s) could not be admissible.

If A* is not optimal, that means we either exit in error (Line #24), or we do not exit with the optimal goal state

(Line #5). This means either I) No goal was ever reached, or II) The goal that was reached is not optimal.

I) No goal was reached.

If the goal is never reached, then we can assume there is no path achieved from start node S to goal node G. However, we know that some h*(s) path exists. This is a contradiction; therefore, some goal must have been reached, though possibly not an optimal one.

II) The goal was not the optimal solution

In this case, we exit with some goal path T from start node S to goal node G. Recall, we update g(n), the actual cost between each step (Line #16), or we don't need to because it is already minimized (Line # 13). This means that the cost of each node within the goal path can be summed up to determine the total cost from S to G. Note that by definition, this would be the heuristic function h(s).

$$h(s) = \sum_{n \in T \setminus \{G\}} c(n, n')$$

Recall that T is not an optimal path. Therefore, we know that h(s) > h*(s). This is a contradiction; therefore, such a T must be an optimal path.

4) Uniformed-Cost Search and Dijkstra's algorithm are similar in idea, but **Dijkstra's algorithm finds a least-cost path to every node**, while **UCS only finds the least-cost path to a single goal node**.

5) In #2, we used the summed Manhattan distance of all nodes from their goal position as the heuristic function. One way this could be improved would be to **omit the white space square from the heuristic**. There were multiple times in my results where the blank square would backtrack, since in the goal, it is placed at its origin. Therefore, it could be equal (or even more favorable) to turn back the path is had previously come from.