

1. **Variable Heuristic (Choosing a Column)**

My variable heuristic is to choose the “most constrained” column. Whenever I pick a place on the board for a new Queen, I update the column coordinate set by removing coordinates, so that the only coordinates in it are ones where I could place the next queen. Thus, the column with the shortest set of the coordinates is the “  
most constrained”—i.e., you have the least options for where to put a queen. I sort my columns based on least to greatest set of available coordinates length, plus a random number between 0 and 1 to break ties. I then pick the first one in the list, which is the most constrained.

1. **Value Heuristic (Choosing a Row)**

I choose a row based on an (approximately) inside out heuristic. Based on the column—representing the x-coordinate--I chose (above), I sort the list of corresponding y (row) values for the column. I used a lambda sort, and the score was the absolute value of the coordinate minus the board size divided by two. For example, on a 4x4 board, row 2 would have a score of 0. Although ties are possible (e.g, both 3 and 1 would have a score of 1 on a 4X4), I didn’t encounter any because the code didn’t ever reach the ends, and adding a random integer didn’t help at all.

1. **Data Structures**

The board was represented by a tuple, with the first object being a list that represented where the queens were based on rows, with the list index being the column, and the value for that index being the row they were in. The list was initialized with all values being -1, so I could know later on to only look at columns where there wasn’t a queen (the value was still -1) when picking the next place to place a queen. The second object in the board tuple was a dictionary, with the keys being column indices, and the values being sets of available rows in that column (usage of these sets is explained in number 1). The sets were initialized with every possible row. My one global variable was a node counter.

For Fun:

It did 800 in 25 seconds but I didn’t have time to test higher or wait that long to run it all the way. Note: The node count actually is almost linear, although this is a log scale because I realized too late. It solved 500 in 500 nodes!

