Improvement:

Sorted based on how many other spots each move invalidated  
I figured that if I chose a spot that would narrow how many other moves I’d have to analyze in the future, I would be able to solve the puzzle quicker.

Complications:  
I tracked how many states it visited total, and without the heuristic, it saw 8562, but with the original heuristic it had to visit 10603 states. It also ran at 150% of the time it took the untouched one to run. I decided to swap …. Explanation … and I found that when I prioritized the move that left the most spaces open (least number of invalidations) it only had to look at 1855 states, and ran in 20-30% of the time. This makes sense because… (only check about 20% of total states visited when used on size 8, 1/8th of the time)

Tried sorting the next variable based on distance from center, failed in both directions (doubles the number of states visited in both reverse and non-reversed, triples the time needed). (abs((size/2)-var))

Has the same run times and states visited when I go 0 to size and size to 0 for choosing the next variable.

The problem I wanted to solve was determining the row that I should shift the next queen in. I figured that I could do this by ranking the row based on the number of moves that were available in that row. I figured if I chose the row with the most number of legal slots for queens, I would cancel out less options and leave a more flexible board to work with for the next moves, and thus have to backtrack less.

Before I spent too much time doing this however, I tried altering how I determined the next row to move a few times. I first compared going from row 0 to n vs. n to 0, and unsurprisingly got equal results in time and boards visited. I then tried sorting based on distance from the middle (abs((size/2)-row)) and got the about the same results. This makes sense because if you decide the next row to change systematically without taking in to account the location of the queens, you will be guaranteed to have to back track many times, as there is no way to “get lucky” or make a good guess. I then tried determining it randomly, and as you would expect, I got some runtimes that were very fast, and a few that were pretty long. I sorted the value assigning process randomly as well, and it helped.

After enough of that, I started making a real heuristic. In order to sort them, I counted the number of valid moves (cols that the queens can be in without being attacked or attacking others) in each row. This was done just by looping through each col of each row and determining if moving there would cause any problems. I kept the random sorting on the value determining function. I was a bit worried that all the extra time it would take me to calculate the valid moves would negate the time saved by not having to visited as many boards and backtrack as much, but that was not the case. When I first implemented the heuristic, I found that it drastically increased the time and the number of boards I had to visited. I then tried sorting in reverse order, so that the rows with the least number of available moves are assigned values first, and this dropped the number of boards I had to visit to anywhere from ½ to 1/10th of what it was with just random sorting, and the time also dropped about 25% from the average time.

This improvement makes sense because at each move we are assigning values to variables that only have one or two valid variables, so we are less likely to assign it an incorrect value that we will have to change later. If we think of this like the color problem we did in class, prioritizing the variables based on how many variables are allowed in them would be like starting by assigning random colors to all the boxes that don’t touch each other first, and then having to go in and fill the ones in between and inevitably reassign the original boxes to different colors. This would be a terrible way to solve it. Alternatively, if you sort based on the most constricted boxes, you could be almost certain that the color you chose would be correct, and there would thus be a very limited chance that you would ever have to go back and change it.

I next wanted to figure out how to make it faster and more efficient in basically any way I could. I tired sorting the variables based on how many other spaces they would make illegal, but this didn’t help much. I also made my code more efficient, but because this didn’t change the number of boards I checked, my runtimes dropped only slightly. I decided to try “hill climbing” where I would start at a random board state and shift the already places queens around until a valid solution was found. I imagined that this would sort faster because not only would there be a chance that we guess really close to a valid board state, but we could also make each of our moves more informed, because there is more data (already places queens) to work with.