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.

For all charts, Bx is the number of boards/states processed for algorithm x, and Tx is the runtime.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size | Ba | Ta | Bb | Tb |
| 10 | 102 | 0.003 | 54 | 0.001 |
| 11 | 52 | 0.002 | 191 | 0.006 |
| 12 | 261 | 0.009 | 172 | 0.007 |
| 13 | 111 | 0.005 | 20 | 0.001 |
| 14 | 1899 | 0.08 | 366 | 0.018 |
| 15 | 1359 | 0.059 | 27 | 0.002 |
| 16 | 10052 | 0.533 | 66 | 0.004 |
| 17 | 5374 | 0.315 | 210 | 0.012 |
| 18 | 41299 | 2.523 | 396 | 0.028 |
| 19 | 2545 | 0.169 | 732 | 0.05 |
| 20 | 199635 | 14.407 | 915 | 0.067 |

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.

a: basic original csp

b: var and val determined randomly

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size | Bb | Tb | Bc | Tc |
| 10 | 132 | 0.004 | 12 | 0.002 |
| 15 | 230 | 0.011 | 18 | 0.006 |
| 20 | 142 | 0.013 | 70 | 0.043 |
| 25 | 4172 | 0.556 | 25 | 0.036 |
| 30 | 1869 | 0.328 | 30 | 0.067 |
| 35 | 13800 | 3.585 | 109 | 0.222 |
| 40 | 977 | 0.238 | 66 | 0.271 |
| 45 | 4766 | 1.487 | 68 | 0.366 |
| 50 | 3221 | 1.144 | 164 | 0.743 |

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.

b: var and val determined randomly

c: valid-moves heuristic on var

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.

d: original hill-climber

e: h-c with valid-moves heuristic on var

f: e starting at a

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| size | Bd | Td | Be | Te | Bf | Tf |
| 10 | 16 | 0.001 | 83 | 0.010 | 22 | 0.002 |
| 15 | 1608 | 0.288 | 187 | 0.049 | 16 | 0.004 |
| 20 | 138 | 0.041 | 25 | 0.013 | 171 | 0.096 |
| 25 | 181 | 0.099 | 63 | 0.044 | 0 | 0.001 |
| 30 | 532 | 0.363 | 126 | 0.123 | 6 | 0.006 |
| 35 | 520 | 0.355 | 60 | 0.072 | 0 | 0.001 |
| 40 | 468 | 0.450 | 60 | 0.092 | 160 | 0.360 |
| 45 | 863 | 0.916 | 55 | 0.109 | 66 | 0.158 |
| 50 | 447 | 0.667 | 53 | 1.303 | 15 | 0.041 |
| 55 | 420 | 0.938 | 113 | 0.468 | 0 | 0.002 |
| 60 | 894 | 2.005 | 88 | 0.354 | 3 | 0.011 |
| 65 | 457 | 0.936 | 66 | 0.297 | 0 | 0.002 |
| 70 | 465 | 1.220 | 98 | 0.419 | 110 | 0.569 |
| 75 | 1497 | 4.609 | 72 | 0.355 | 90 | 0.530 |
| 80 | 2239 | 6.860 | 59 | 0.332 | 2 | 0.021 |
| 85 | 1006 | 4.039 | 180 | 1.079 | 0 | 0.005 |
| 90 | 1953 | 9.529 | 179 | 1.115 | 11 | 0.087 |
| 95 | 1650 | 7.985 | 168 | 1.229 | 0 | 0.004 |
| 100 | 818 | 3.561 | 124 | 1.012 | 168 | 1.677 |

I started with a randomly generated state, and then until the number of conflicts was 0, I would pick a random row and assign it to the col in that row that caused the least collisions. If the new state had less collisions than the old, state, I would set the old state to the new state, and it would recur. Otherwise, I kept with the old state. This only worked when the randomly generated state function was lucky enough to guess a valid state. Crap. I decided to try tracking everything, the number of visited states, the number of conflicts, the current state, ect, and found that because determining the next variable was too rigid of a formula, it would return the same value for the same variable over and over again, even if there were other values with an equal number of conflicts. To combat this, I added a random decimal to the end of each number of conflicts, so that the same value wasn’t chose over and over again. This helped, but my code would then get stuck on the part where I checked if the new state was better than the old states, because when the number of conflicts was the same, it would be determined solely by the randomly generated number, so I ditched that part.

This worked very well. For example, I was able to cut the runtime for a board size 13 from 1 second to .03 seconds, and the number of boards I had to visited dropped from 6707 to 362. This solution makes a lot of sense. Rather than having to fill up a board with limited information (not all the queens on the board) and backtrack whenever it got stuck, with hill-climbing, the program has way more data to make decisions with, and it has to back track less. In fact, I was able to implement a solution that never explicitly backtracked, it just kept moving forward and would correct itself when needed (climb\_3). There was also the chance that the random state method would spit out a board that was only a few moves away from a solution, which made solving it very fast.

After getting a working hill-climber method, I tried to find ways to calculate the randomized values (the initial state and the next row to change) in a better way. I first tried using the next row determiner from my first improvement in this hill climber methods.