David Fantin

Dr. Kuttal

CS-2003-02

March 7, 2019

Project 2 Report

The logic behind my code is as follows: first, place a queen in the A1 square and mark the down ward diagonal and row with the value of the column the queen is in plus one. Then move to the next column and scan down the rows until an empty square is found and place a second queen there. After the queen is placed mark all empty squares (squares with the value of zero) in the same row as well as the upward and downward diagonals that emanate from the queen, marking the value again as the value of the column the queen is in plus one. Repeat these steps until it comes to a column with no empty squares. When that happens, you backtrack. To do this the computer goes to the previous column, removes the queen and all subsequent marks that queen had on the board (places the queen attacked) and moves down to the rows to the next empty square and places another queen, repeating step 2. If there are no empty squares however, the computer must backtrack again and repeat the previous step. This all happens over and over again until a solution is finally found and no queens are attacking eachother.

The way I graphed the number of “Backtracks,” “isAttacked” calls and “placeQueens” calls was a tedious process: for both scenarios I ran the code, plugging in values 4 through 25 and copied the results onto a sheet of paper. Then I started up excel, and recopied the results into three separate columns. Then, by use of Excel’s graphing function I graphed the results onto a single graph. From the results of this project, I can conclude that backtracking becomes increasingly more inefficient the larger the value of N gets, as shown when I tested numbers larger than thirty and it took a very long time to actually compile. But from what I can gather from the graph, I seems as though even numbers take longer to compile than odd most of the time, as shown by spikes in all three graphs at some of the higher even numbers (especially 22).