Dan Scarafoni

Scheme Weeks 3-4

Overview

The purpose of this assignment was to implement, in Scheme, various methods for solving the n queens problem. This problem is to place N number of queens on a chessboard of NxN size in such a way that no queens can attack each other.

Methods

Algorithms

Two major algorithms were implemented in this project- backtracking and min-Conflicts

Backtracking

this algorithm uses two nested recursive loops to run through each row in each column, and each column in the board, and places the queen in the first spot that causes no conflicts. Because there are n columns and it is impossible to have two queens in the same column or row legally, the program must place exactly one queen in every column and row. Because the second recursive loop runs through every row index in a given column, it provides an intuitive means of knowing when an impossible situation has been reached: if the end or a column is reached, and no piece is placed, then the current placement of pieces must be wrong in some way. The program backtracks to the previous column, at the row that it placed the last piece off, and continues from this point in the loops. The program terminates as soon as it places the last queen.

The recursive iterations generate a solution tree where each node has n children, which expands and explores different solutions for the problem. A solution is guaranteed, however, the solution is extremely inefficient. Worst case, the program will go through all possible branching combinations, which results in exponential run time. The following diagram illustrates the algorithm.

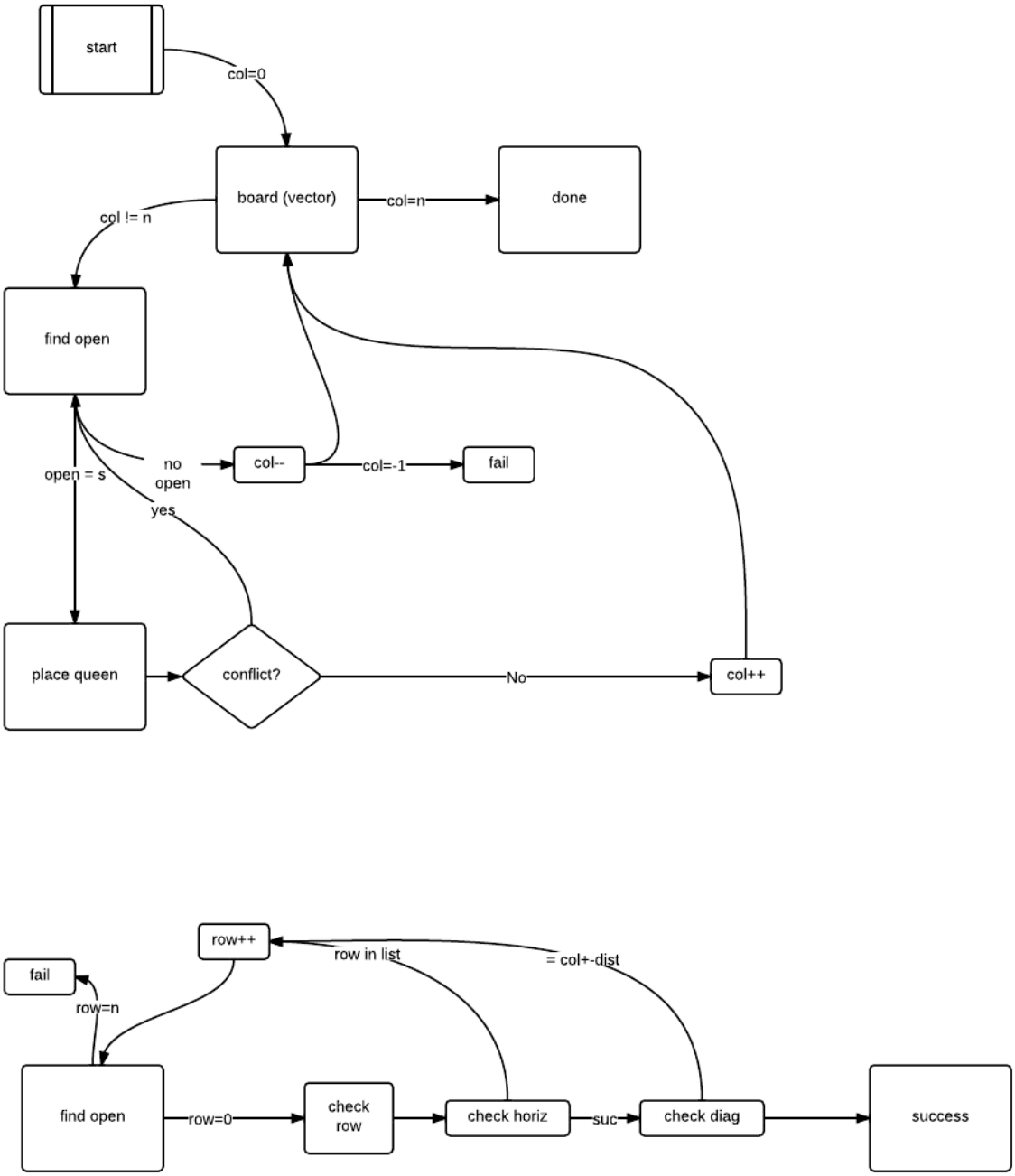


Figure 1- the algorithm for backtracking

Min-Conflicts

This algorithm implemented hill-climbing. The formula began by generating a board with each column containing one randomly placed piece. The formula then recursively iterated through the board, finiding the row index in each column where there were the least amount of conflicts, and moved the queen into that space. This repositioning continued until the loop could be completed without finding any conflicts, at which point the program was finished and the problem was solved.

One major issue with this algorithm was its penchant for becoming stuck at certain configurations. The board would shift each piece into a location where all pieces were at their positions of least conflict, but the board would not be solved. No piece's location could be further optimized, and the program would become “stuck.” In cases such as this, the program would wipe the board clean and generate another board of randomly placed pieces of the appropriate size. The following diagram illustrates this algorithm.

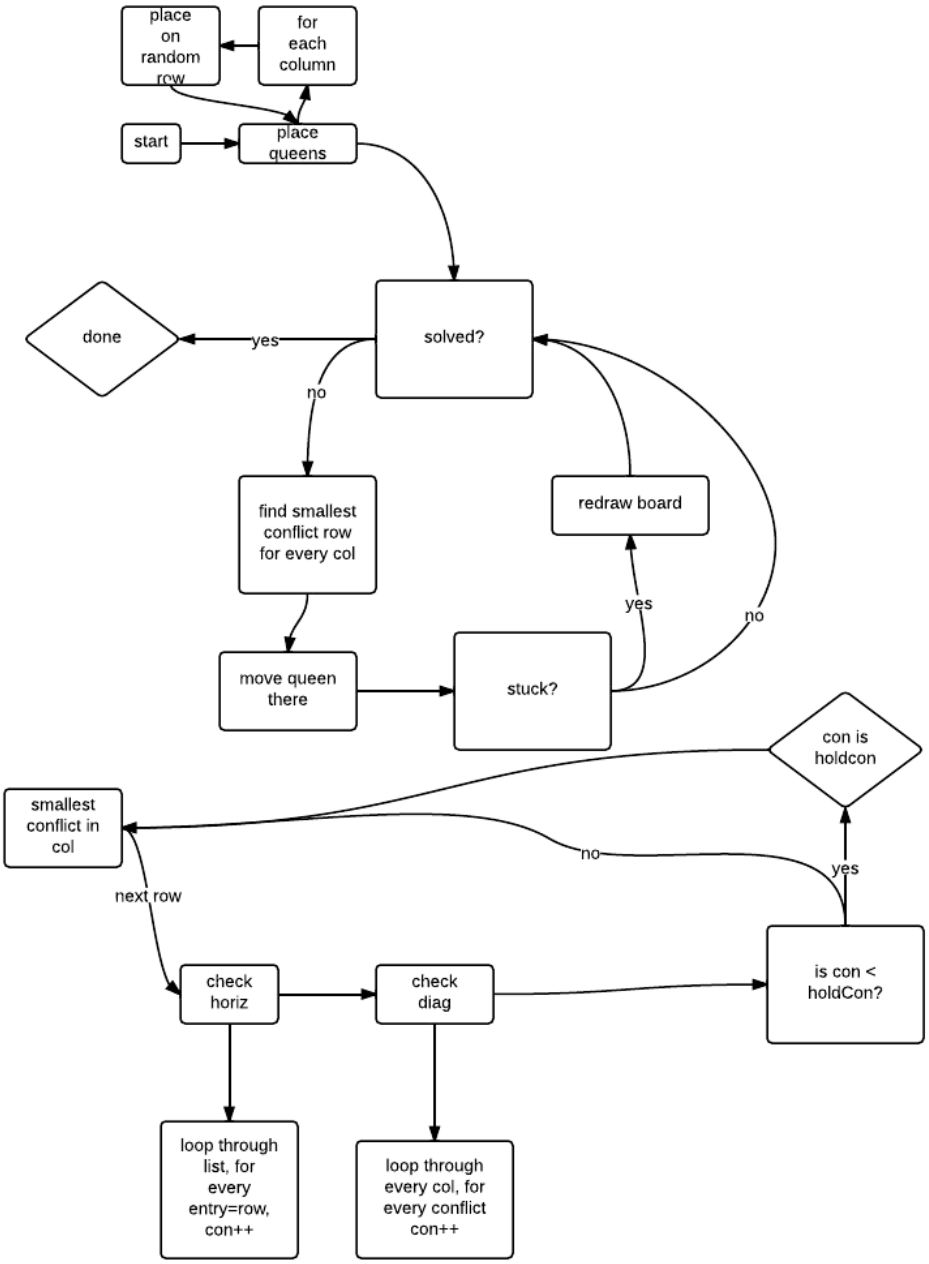


Figure 2- the algorithm for min-conflicts

Data Structures

The following data structures were implemented for the programs

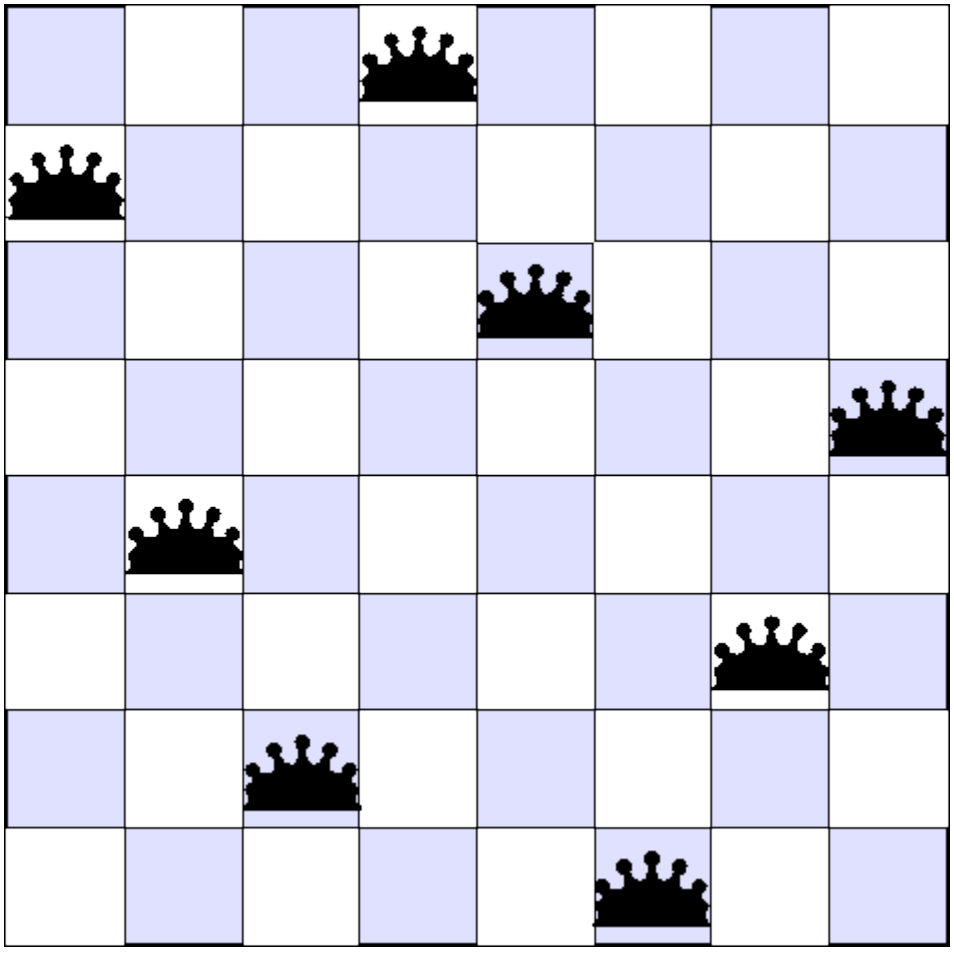
The board was, in both cases, represented by a one dimensional vector of length N. Each index in the vector represented the row in which the queen in that column was placed. This was preferable to a NxN two dimensional vector or a list because it greatly lessened the size of the board, and simplified the information stored therein. Vectors were also preferable over lists as they are random access, which would greatly increase the efficiency and ease with which elements in the board could be accessed.

The states in Backtracking were not tracked. The board did not explicitly make a tree of possible board states, because the sheer overwhelming branching of this tree (as mentioned above) would have been so great that it would have been impossible to keep track of all states encountered so far. Instead, the board simply bounced from one state to another (changing columns and rows as was dictated by the algorithm, without any real tracking as to how it ended up in that particular state.

In min-conflicts, states were handled similarly. However, instead of switching from one row/column to another, it went from the leftmost columns to the rightmost, repeating the loop (and restarting the board) when necessary.

Results

Then n-queens problem is solvable for any n larger than 3. The results of this experiment seem to back this up, as results were found for boards of size 4-20 (any larger and the program failed to terminate in reasonable time). The following from Smith College's computer science department illustrates one of the solutions to the problem for n = 8.



For backtracking, the problem was attempted for boards of sizes 4-20. The number of checks for conflicts and queen placements were plotted against the size of the board and number of queens (N). The results are as follows. Please note the logarithmic scale on the vertical axes for all graphs

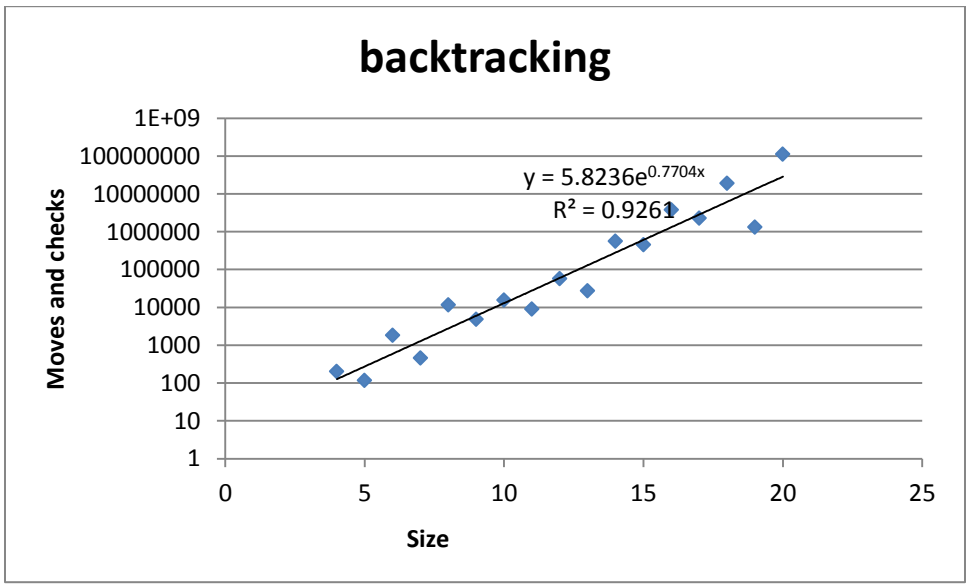


Figure 3- back-tracking's results

This chart indicates that the back-tracking algorithm runs in polynomial time, though still in Big O of e^(x). The median and average was not necessary for this algorithm, as there were not randomizations in the board, and the board always reached the same end point for every N.

Some alterations were made to the back-tracking algorithm, and a separate file was created that run the algorithm by starting in the middle of the board and working out. The same test as above was run and the results were as follows.

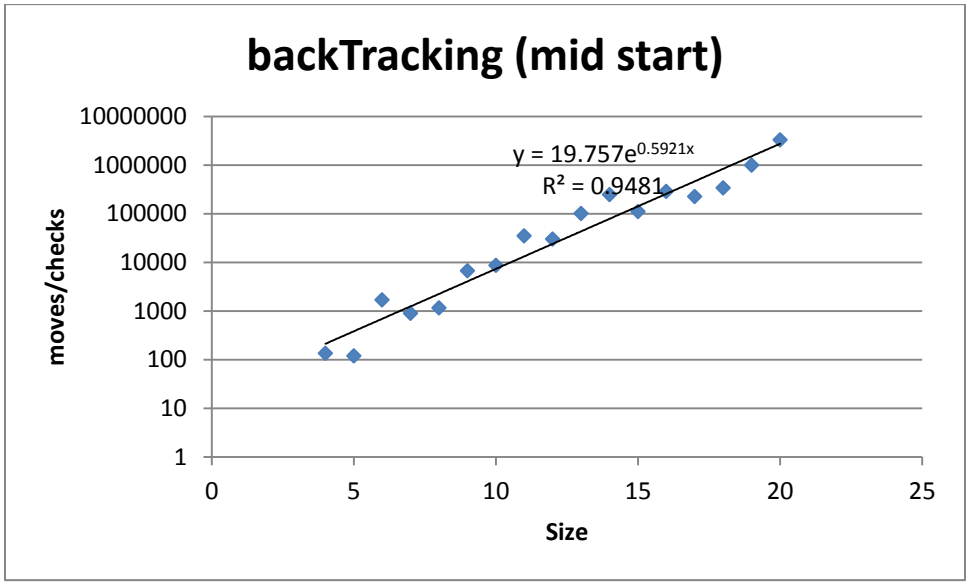


Figure 4- back-tracking with a middle start

As is clearly visible, the mid start algorithm worked slightly faster than the left to right algorithm, although still running in exponential time. In this case, it ran in big O of (e^(x)) time.

For min-conflicts, the data was computed using slightly different methods. Because each time the program was run, the initial queen placement was random, the number of checks for a given problem size could vary considerably. As such, for every board size, one hundred tests were run, and the mean of these runs was reported. The following chart illustrates the data.

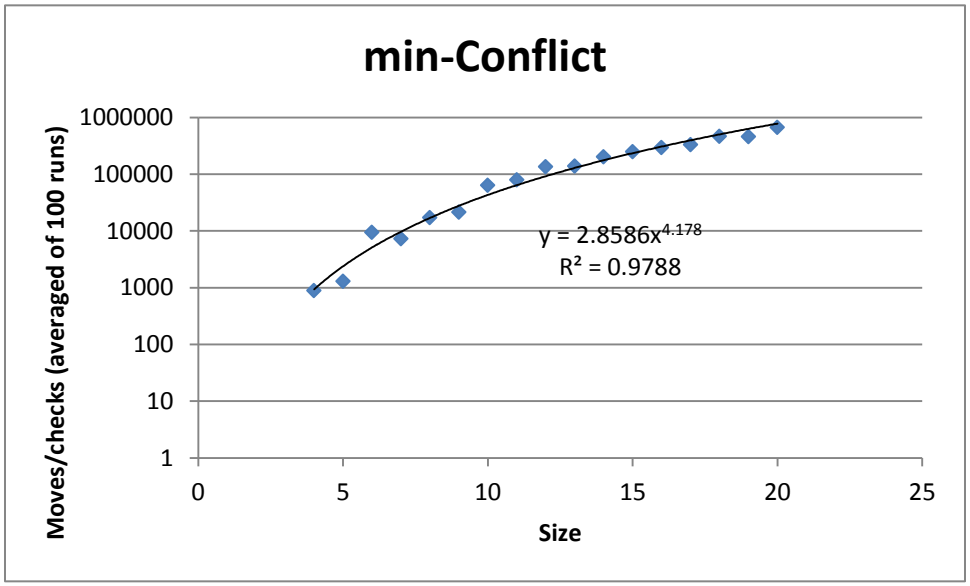


Figure 5- min-conflicts results

The results above indicate that min-conflicts is much more efficient that back-tracking, running in exponential time in Big O of (x^5).

A variation of min-conflicts was also run. This formula was similar to the above, except that the initial board was not generated randomly. Instead, the initial board had all the queens in a diagonal line, with the queen in column one being in row one, the piece in column two at row two, etc… The following graph shows the data.

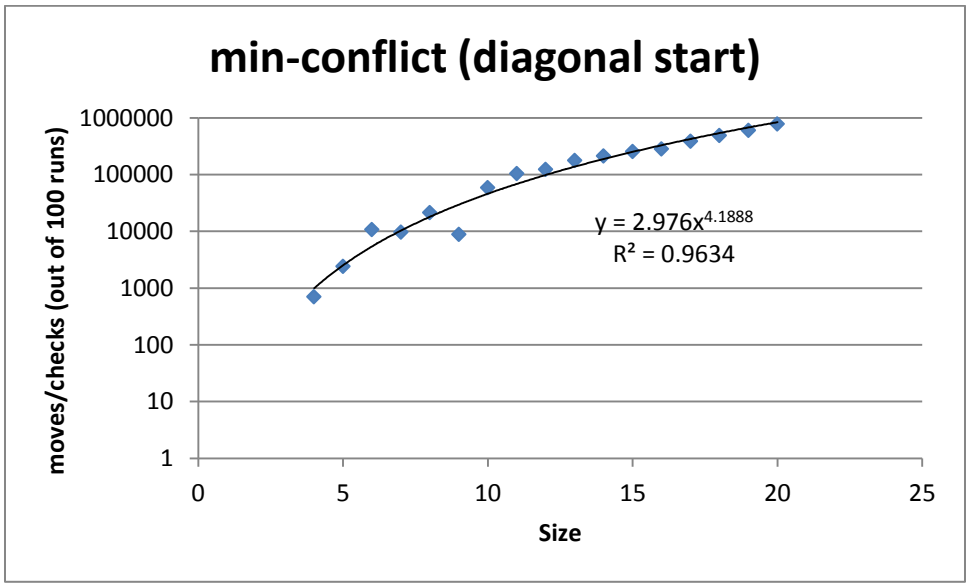


Figure 6- min-conflicts with fixed starting positions

When compared to the previous graph, no real difference in efficiency can be deduced. Both algorithms run in virtually the same time, both having executing in big O of (x^5).

Conclusions

Based on the above data, we can conclude safely that back-tracking is a significantly less efficient algorithm than min-conflicts, although there can be more variation in min-conflicts runtime (as a result of the random nature of some of its functions). Further, we can also surmise that random initial map generation does not have a noticeable effect on runtime in min-conflicts, as there was no significantly large difference between the diagonal start algorithm and the random initial placement algorithm. Further research could be done to investigate this point, however. The placement of queens in other patterns (for instance, all in the same row) could noticeably impact runtime. Because there are thousands of ways in which a board could be initiated, a large number of experiments would have to be done to determine which (if any) improve efficiency.