Thomas Carr

ITCS 6150

03/06/2022

N-Queens Local Search Algorithm

The problem being solved is a classic search algorithm problem known as N-Queens. The goal is to move a piece around a board and ensure that the pieces don’t violate a condition. To understand the condition, we must look at chess. The piece we are looking at, as the problem name implies, is the queen piece. The queen piece can move as far as it wants in any horizontal, vertical, or diagonal direction. The condition that we are attempting to solve is that no queen pieces conflict with each other. The size of the board and the number of queens comes from the name of the problem: N. The board is N by N, and there is N queens.

My solution for this problem was utilizing a local search algorithm called hill-climbing. Essentially, the algorithm will repeatedly choose the best move available until a solution is found. The best move is determined by a functioned called the heuristic function. The heuristic function I used for this problem is called minimum conflict heuristic. It looks at one of the queens and counts how many queens it can reach horizontally, vertically, or diagonally. The lowest count is the one that it goes with. One modification I made was the addition of a list of previously moves to ensure that it doesn’t get stuck in a loop of moving a queen between two spots.  
 Diving into the code, let’s look at the coding structure. The code is written in Node.js, which is my choice in quick scripting languages. The file structure is simply two JavaScript files. One contains all the functions used for the Hill Climbing algorithm while the other contains the main logic of the N-Queens problem. There were four functions used for Hill Climbing:

* generateMoves
  + This function takes in the current board and generates all possible moves that all of the pieces can make. Unfortunately, this has a horrible time complexity, but for the sake of being thorough, it has to get all moves for all pieces.
* evaluateMoves
  + This function takes the output from generateMoves (an array of all moves), iterates through all of them calculating the heuristic value of the move, and returning the best move.
* calculateHeuristic
  + This function takes in a board, looks at every piece, and counts all of its conflicts. Since each piece is iterated through, each conflict will count as 2, so before returning the heuristic value, it is divided by 2.
* logBoard
  + Not as much for Hill Climbing, but it takes and prints out the board. This is used for visualizing the starting and solution boards.

Now let’s look at the index file. The first tunable is N, which will change the board size and queen count. The second tunable is the verbosity: 0 only shows start/solution, 1 shows all steps. With those tunable set, lets dive in and go through the pseudocode:

* Generating the NxN board
* Placing N queens randomly
* Hill Climb Loop
  + Generate the moves
  + Evaluate the generated moves
  + Update the current board with the best move
  + Print the board (if enabled)
  + Check for solved
    - I reuse the calculateHeuristic function since a value of 0 will mean that there are no conflicts, thus the goal was found.
* Log the information

In the end, the Hill Climbing algorithm was very successful and was able to complete whatever size board I gave it. I was able to find solutions in minimal steps because it would avoid loops and would continuously choose the best move available without looking too far ahead. Below are some of the results I achieved:

* N=4, 0.007s, 2 steps
* N=5, 0.011s, 5 steps
* N=6, 0.037s, 33 steps
* N=7, 0.043s, 20 steps
* N=8, 0.026s, 6 steps
* N=12, 0.1s, 15 steps
* N=24, 6.524s, 83 steps
* N=45, 98.502s, 84 steps
* N=50, 79.862s, 42 steps
  + Text

    Description automatically generated
* N=100, 4505.202s (1.25 hours), 66 steps
  + Text

    Description automatically generated

I attached proof for the larger ones (especially 100) because it took so long to execute. Additionally, I had to run the N=100 with a higher ram allowance because just storing potential moves of 100 pieces requires an incredible amount of space (Pieces \* N \* [Directions – 1]), or 990,000 arrays per iteration. Outside of the exponential complexity in relation to N, I believe that the results were excellent.