NQueen (MinConflict) Project Report

Dylan Chung

CS 4200.01 Spring 2020

Dominick Atanasio

March 13, 2020

My Approach

For the most part, I followed the given pseudocode, however rather than generate all the conflicted variables, and selecting one from it, my approach involved selecting any random "variable" and then check if it is conflicted or not. If it's not conflicted, my program kept picking random variables until it settled onto one that had a conflict. I felt this approach would be cheaper overall than to generate all the conflicted variables at each iteration. In the first method, I would have had to check for conflicts for all 25 variables and then pick a random conflicted one at each iteration. In my "pick and check if it's conflicted or not" approach, each iteration involves randomly checking, and at first it should be fairly quick at finding conflicted variables as almost all of them are conflicted. As it gets less conflicted however, it may take longer, but I believe it is overall faster than the former method.

From the selected conflicted variable, I move the variable or Queen through each position in the column, calculating the amount of conflicts that would occur if the Queen were to be moved to that row. At the same time, I'm keeping track of which position had the lowest amount of conflicts. At the end, I will move the Queen to the lowest index, however, without taking some extra precautions, this may lead to a local minima in which the fitness is at two and there are only two variables in conflict with one another. This results in the same calculations and moves being done over and over again until the max iteration count is hit.

This issue occurs when there are multiple lowest number of conflict indexes, and your program always decides to pick the same one at each iteration. To counter this, I ensured that when there were multiple lowest conflict indexes, I would choose a lowest conflict index at random. This ensured that even if it came down to a fitness of two, when the calculation runs, it would eventually choose a different minimal conflict index to move to at each run. Another approach would have been to do something similar to simulated annealing and randomly mutate the variable when this condition occurs to pick a worse move in hopes for eventually settling on better.

Data

A 25N Queen Solution (Genetic) at 400 Instances with a generational limit of 500 generations.
Avg Solved : 95.0%
Avg Generations : 167.5375 generations
AvgRunTime : 0.024055004 seconds
TotalRunTime : 9.622002 seconds

A 25N Queen Solution (Annealing) at 400 Instances with Initial Temp: 1000 and CoolDown Rate at 0.95

Avg Solved : 100.0%
Avg Generations : 520.45 generations
AvgRunTime : 0.0056025027 seconds
TotalRunTime : 2.2410011 seconds

A 25N Queen Solution (MinConflict) at 400 Instances with Limit 1000 Iterations
Avg Solved : 100.0%
Avg Generations : 72.89 generations
AvgRunTime : 5.625006E-4 seconds
TotalRunTime : 0.22500025 seconds

Analysis & Findings

As shown by the data above, MinConflict is significantly better than Genetic and Simulated Annealing in terms of both the amount of generations needed and the average run time. MinConflict also solves 100% of the time on average because on average, a 25N Queen problem is solved within 72 loops, and I have a cutoff of 1000 as suggested in class. Timing is extremely quick because arrays are contiguous in memory, and iterating through them is fast enough. Although it was recommended to use a HashMap and HashSet for improved performance, I felt it was not necessary, and as shown by the timing, it really wasn't. In fact, MinConflict is faster than Annealing and Genetic by multiple seconds in the TotalRunTime.

But aside from accessing stuff from data structures being a cause of slow down, it makes sense that MinConflict is faster because Genetic requires remaking the entire population at each generation, sorting them by fitness, and selecting two of the best randomly each time to create a new generation of the same population size over and over again until one of them has a fitness of 0. As one would imagine, that would be fairly time consuming and expensive, and on average, my algorithm did this 167 times with a population of 100! This meant 167 x 100 Board Objects were created.

Annealing involves constantly creating successors by altering the position of one queen on the board until one is found to be acceptable or the algorithm accepts it because it's within range of the acceptance. Since annealing tends to get more picky as it cools down, the amount of random movements made and checked against would result in numerous successors being generated and rejected towards the end. In some cases, since my implementation did not account for this, it was possible to randomly generate a previously checked board and have it get rejected again.

MinConflict on the other hand does not randomly make boards by moving at random until something works, but instead it has a fixed location to go to at each row, that being the

lowest conflict index. Each loop is thereby fairly quick as there is less random work involved and a decision is found fairly quickly each time because this position can be calculated.