Taylor Thurlow

12 October 2018

CS 4200 Fall 2018

Analysis of Solving 21-Queen

I chose to represent the N-Queen problem in a basic object-oriented format. The board itself is represented as a semi-stateless object, such that it only stores information about its original queen positions and size of the board (in this case fixed to 21). This way it is easier to apply different solution algorithms to the same board instance without worrying about the internal state of the board object.

In order to test both algorithms, 500 random boards were generated. Each generated board was solved using both Simulated Annealing and Genetics. Due to the nature of local searches, the solutions to the same board are not guaranteed to be the same, and in fact are highly likely to be different. Solution time of each solution was tracked and logged, as well as solution cost for both. In the case of Simulated Annealing, cost increases by 1 for every child state generated. For the Genetics algorithm, cost increases by 1 for every generation of new child states.

Implementation of Simulated Annealing was pretty typical, but did take some tweaking. I ended up with a ‘schedule’ in which the temperature is multiplied by 0.95 for each iteration. However, if the temperature is below 0.01, a fixed temperature value of 0.01 is used. Searching stops if the algorithm reaches a maximum cost of 100,000. In my testing, every board was solved so this limit was never reached. Additionally, my initial temperature is based on board size - N^2 - where N is the board size, in this case, 21. This results in a starting temperature of 441 for 21-Queen. Tweaking the values was the hardest part of this algorithm but I am very satisfied with where it has ended up.

Implementation of the Genetics algorithm seemed easier initially, but in my experimental results turned out to be far worse at solving 21-Queen. I chose a maximum number of generations of 5000, a mutation probability of 80%, and a population size of 1000. The population was stored in a priority queue sorted by the number of attacking queen pairs. Because the Java PriorityQueue treats lower values as higher priority, this allowed me to easily select the most fit members of the population. Each generation involved selecting the 1000 most fit members of the population, and choosing 1000 random pairs to reproduce. The resulting 1000 children are added to the population and the loop repeats until either we have reached the maximum number of generations or a solution to the problem has been found. Implementation of Genetics was in general easier, but in the long run turned out to be far worse at solving 21-Queen. I am not sure if this is a result of a problem with my implementation, values chosen, or just a result of Genetics being a poor way of solving boards of this size.

Data

SA Solution %	100.00%
G Solution %	95.20%
SA Runtime Average of Solved Problems	2.83
SA Runtime Average of All Problems	2.83
G Runtime Average of Solved Problems	2863.03
G Runtime Average of All Problems	4782.73
SA Cost Average of Solved Problems	5066.29
SA Cost Average of All Problems	5066.29
G Cost Average of Solved Problems	298.80
G Cost Average of All Problems	524.45

As is evident from the shown data, the Simulated Annealing solution is far, far better at solving 21-Queen. It solved all 500 instances with an average solution time of 2.83 milliseconds. Genetics only solved just around 95% of problems, and took more than 1000 times as long in cases where it was even able to find a solution. The average of all problems is not super useful as a data point because those unsolved problems are time limited due to hitting the iteration maximum.