Project 2

N-Queen

CS 420 - 01

Kristin Adachi

knadachi@cpp.edu

August 23, 2017

Project Specification:

This project focused on using local search algorithms to solve n-queen problems. We were required to implement the straight-forward steepest-ascent hill climbing algorithm as well as another algorithm of our choice. For my project, I chose to implement the genetic algorithm. My program is able to use both algorithms to generate and solve n-queen problems. For the steepest-ascent hill climbing algorithm, my program displays the original board and the solution if it is able to find one. For the genetic algorithm, my program only displays the solution. For analysis, I give the user the option to generate 200 n-queens instances and it attempts to solve all of them. However, I only did this for the steepest-ascent hill climbing algorithm because the genetic algorithm took too long to solve a large number of instances.

My Approach

As mentioned before, the two algorithms I implemented for my program were the straight-forward steepest-ascent hill climbing algorithm and the genetic algorithm. In order to do so, I used six classes: `Board`, `HillClimbing`, `Genetic`, `GenCompare`, `UI`, and `NQueen`. An n-queen board is represented as an array of integers. The `HillClimbing` and `Genetic` classes handle each of the local search algorithms. `GenCompare` is used for the genetic algorithm to order the population by their fitness. Finally, the `UI` class handles all the user interface for my program and the `NQueen` class contains the main method.

I implemented the straight-forward steepest-ascent hill climbing algorithm in the following manner:

1. Find all possible successor boards formed by moving any queen within its column.
2. Find the number of pairs of attacking queens for the current board.
3. If none of the successors are better than the current, the algorithm quits.
4. If a better board is found, the current best board is updated and the algorithm repeats.

Due to the way this algorithm is implemented, it is likely that an n-queen solution cannot be found. As mentioned earlier, the program prints the original board and the board of the solution if it is able to find one. It also prints how long it takes to attempt to find a solution as well as its cost (the number of boards generated).

The genetic algorithm involves a population, a fitness function, random selection, and the ideas of crossing over and mutation. The population size is determined by the user, and I used the number of pairs of non-attacking queens to represent the fitness value of an n-queen board. I implemented the genetic algorithm in the following manner:

1. Find the fitness value for all members in the population.
2. Select two random members to be parents for the next generation.
3. Find and remove the 25% of the population considered to be "least fit."
4. Populate the successor generation by crossing parents with remaining members.
5. Apply random mutations to the 25% of the successor generation considered to be "least fit."
6. Repeat the algorithm until the ideally fit member is found.

Due to the implementation of this algorithm, it is guaranteed that a solution will be found. For this one, my program only prints the solution, but also includes the time it took to solve the n-queen problem and the cost (number of generations generated).

Analysis and Findings

To analyze the straight-forward steepest-ascent hill climbing algorithm, I ran 200 n-queen instances and attempted to solve them. I had my program generate 200 boards of size n. It kept track of how many solutions were found, the average time it took to run each algorithm, and the average cost of running each algorithm (number of boards generated). We were told that this algorithm can only solve about 14% of n-problems for n = 8 so I tested that first with my program. As I ran several tests of 200 8-queen instances, I noticed that my algorithm was able to solve about 11-15% of the problems which was similar to the information we were given. Then, I ran it several times for n = 22. This time, I noticed that the algorithm only solved 1-4% of the problems given. It also took a little longer to run the algorithm and the number of boards generated increased significantly.

The steepest-ascent hill climbing algorithm can solve so little n-queen instances because it can easily get stuck at a local minimum/maximum or at a plateau. If it reaches any of these points, the algorithm simply quits and a solution is not found. As mentioned in class, the

performance of this algorithm could be improved by allowing sideways moves. This improvement would allow the algorithm to continue on a plateau and hope that it eventually hits a shoulder. As a result, this would lower the chances of the steepest-ascent hill climbing algorithm becoming stuck at some point.

For the genetic algorithm, I originally tried to run 200 n-queen instances but it became too slow when the population size was too big or if n became too big. Because of this, I omitted running a large number of instances and ran single instances at a time. As I ran several instances of n = 22, I noticed that the genetic algorithm is always able to find a solution. This is due to the fact that the algorithm does not terminate until a solution is found. I also noticed that if you increased the population size, the time it took to run the algorithm increased as well as the generation count.

After testing both algorithms, I found that each had its pros and cons. The steepest-ascent hill climbing algorithm seemed to be fast in general, however, it was more likely not to find a solution. On the contrary, the genetic algorithm always finds a solution, but it seems to run slower. I think this is most likely due to the fact that it does more work in order to guarantee a solution. In conclusion, I still believe that the genetic algorithm would be the better choice of the two because it guarantees a solution.