Homework 2: Hill Climbing, Genetic Algorithm, and Particle Swarm Optimization

Daniel Wandeler
University of Colorado-Colorado Springs
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80919
dwandele@uccs.edu

Abstract

The N-queens problem requires finding an arrangement of queens on a chess board such that none of the pieces are attacking each other. This problem is common in the programming community to show computational differences in search optimization algorithms. This paper shows the results of testing the hill-climbing algorithm, the genetic algorithm, and the particle swarm optimization algorithm in the setting of the N-queens problem.

Introduction - n-Queens Problem

The n-Queens problem is the problem to find an arrangement of N queens on a chess board, such that no queen is attacking any other queens on the board. Queens can attack as many spaces as they want horizontally, vertically, or diagonally. Looking at this problem in terms of state-space search, the initial state, actions, transition model, and goal test need to be defined. Because each queen is placed on a distinct row and column, solutions can be represented as an array of length N with the values representing the column number the queens is placed in and the indexes showing which row the queen is in. There should be $N \times (N-1)/2$ total pairs of non-attacking queens for a board of size N (Thada and Dhaka 2014).

The initial state is a list representing where the queens are located of length 0 since no queens have been placed yet. The set of actions available to the agent are to add the row number of the newly placed queen to that list that does not conflict. The transition model for the agent is adding a queen. Adding a queen will lengthen the list of our state. The goal test or final state is when all N queens are placed on the board without attacking one another, but specifically, any list of length N should be accepted since checking insures no false positives reach the end.

Assignment

Hill Climbing

The hill climbing algorithm showed how difficult it is to compute a solution to the N-queens problem. It took about 35 runs of the algorithm for it to find a neighbor with a perfect fitness score. However, this algorithm is by far the fastest and uses the least memory of the three algorithms tested in this paper.

The algorithm begins by starting at a randomly initialized solution state. The states are represented as one-dimensional arrays, where the index is the row number, and the value is the column number of where the queens is placed on the board. The algorithm then scans through all possible neighbors of the current board state. It then jumps to the neighbor with the best fitness score. In the case of the N-queens problem, the fitness score of 0 is ideal, and scores worsen as the number increases. This optimal neighbor-jumping step is repeated until one of these conditions is met:

1. There are no neighbors with a better fitness score that exist.
2. There are neighbors with the same fitness score that exist.
3. The current state's fitness score is 0.

In case 1, the algorithm breaks and returns the current state and its fitness score. If this occurs, it is assumed that the algorithm has reached a local maximum and will not find a better solution if given more time to execute. In case 2, the algorithm jumps to a random neighbor of the same score to avoid converging on a local optimum. In case 3, the global optimum has been met, since that state is a perfect solution to the N-queens problem.

The first issue encountered was solving how to write the neighbor-jumping function. Initially, I designed the neighbor-jumping function to find the first collision of queens in a state, then randomly take one of the two queens, and move it to a random column in its row. I thought this solution would eventually get me to the correct solution since eventually, all possible iterations would be tested. I quickly realized that this solution loops into itself in some cases, which could cause the algorithm to run forever. I solved this by using multiple for loops to iteratively check each possible move that could be made from the start state. It calculates the number of collisions at each possible move, then moves to the best scoring move. This process then loops until the algorithm reaches a local optimum or a perfect solution. Secondly, I encountered an issue where the algorithm assumed that moving all the queens into the same row was the best solution. The best solution returned would either put all the

queens in the first or last column, stating that no better neighbors could be found. I solved this issue by fixing a bug in the fitness function that wasn't properly recalculating the fitness after a move from the [0,0,0,0,0,0,0,0] and [7,7,7,7,7,7,7,7] positions. Finally, I encountered an issue where the number of boards checked was not being properly calculated. I solved this issue by changing where the breakpoints in the algorithm were located. Fixing this bug also helped make the code cleaner. Test runs and their results can be found in the Results section of this paper.

Genetic Algorithm (GA)

The genetic algorithm was fairly easy to implement in this case since I am also using a GA for the semester project. I adapted the code from the project that accepts a string of input to accept a list instead. The solutions to the N-queens problem are represented as arrays, where the index is the row number, and the value is the column number. Each gene represents a value in that array, and the chromosome is the full array. The genes are randomly initialized integers between 0 and 7, to represent the column numbers. Each chromosome is made up of eight randomly picked genes. In this implementation, each population has 100 individuals, or chromosomes. Between generations, the new chromosomes are created by taking genes from two of the better half of the performing chromosomes in the previous generations. For each index in the solution array, parent 1 will pass down its gene 45 percent of the time, parent 2 will pass down its gene 45 percent of the time, and 10 percent of the time, a randomly initialized gene will be passed to the new chromosome. In the initial implementation, each parent passed down their gene half of the time, but this resulted in the algorithm getting stuck and not finding a perfect solution in some cases. Otherwise, since I was able to mostly take my implementation from another project, I encountered few issues with this implementation. Multiple test runs and their results can be found in the Results section of this paper.

Particle Swarm Optimization (PSO)

The particle swarm optimization algorithm was the toughest to implement personally. In my implementation, each particle is a class object that has a position, velocity, error, best previous position, and best previous error variables. Each particle's velocity and position are randomly initialized. The algorithm also contains a function to update the velocity, update the position of the particles, evaluate the fitness of the particles, and creating the swarm. In the velocity updating function, the inertia constant was set to 0.728894, the cognitive term is set to 0.5, and the social term is set to 1. In this implementation, 100 particles are made and are adjusted over 500 iterations.

Firstly, I ran into an issue with the states not copying properly, so the particles would move sporadically between iterations. However, even once the states were copying correctly between iterations, I had an issue where the velocity was so low, that with rounding, it would not change the integer value. I fixed this by adjusting the cognitive and social constants to give more meaningful shifts between iterations. The results of multiple runs of this algorithm can be see in the Results section of this paper.

Evaluation Metrics

The assignment specifies requiring two separate evaluation functions to show results from each of the different methods to solve the n-Queens problem. The first function developed is an execution time checker. This will compare each method, show the total execution time, and the corresponding N value.

The second evaluation function developed is to show the number of unique boards that the algorithm evaluates for completeness before finally reaching an acceptable solution.

Results

Table 1 is the results of multiple runs of the hill-climbing algorithm against the N-queens problem.

N=8	Boards Checked	Execution Time	Final Fitness
Run 1	6	10.97 ms	0.0
Run 2	4	7.98 ms	1.0
Run 3	4	6.98 ms	2.0
Run 4	4	8.97 ms	1.0
Run 5	6	11.97 ms	0.0

Table 1: Hill Climbing Algorithm

Table 2 is the results of multiple runs of the genetic algorithm against the N-queens problem.

N=8	Generations	Execution Time	Final Fitness
Run 1	40	133.66 ms	0.0
Run 2	758	2508.85 ms	0.0
Run 3	359	1171.16 ms	0.0
Run 4	22	75.80 ms	0.0
Run 5	20	112.70 ms	0.0

Table 2: Genetic Algorithm

Table 3 is the results of multiple runs of the particle swarm optimization algorithm against the N-queens problem.

N=8	Iterations	Execution Time	Final Fitness
Run 1	500	2189.15 ms	2.0
Run 2	12	51.86 ms	0.0
Run 3	500	2053.59 ms	1.0
Run 4	500	2410.86 ms	3.0
Run 5	500	2058.74 ms	2.0

Table 3: Particle Swarm Optimization

The time complexity of solving the N queens problem through hill climbing shows that it is more efficient to start over and try to find a perfect solution from a different starting point than it is to backtrack and continue searching for a new solution. In this implementation, the hill-climbing algorithm just runs until it doesn't find any better neighbors. The

results will likely be different if given tries at different starting points per execution. The genetic algorithm very successfully found a solution every time it was run, but in some cases, it took a few hundred generations to get to that point. The particle swarm optimization only reached a perfect solution once within 500 iterations, but the time complexity is close to the same across all execution since the same number of iterations is performed each execution.

Conclusion

This assignment shows the difference in performance of different optimization algorithms in the N-queens problem. The hill-climbing algorithm works quickly and does not use a lot of memory, but it quickly falls into traps of local optimums, and shown through testing, struggles to reach a perfect solution. The genetic algorithm works less quickly than the hill-climbing algorithm, but almost always reaches a perfect solution within one thousand generations. Finally, the particle swarm optimization algorithm works quickly and with little memory, but it falls into the traps of local optimums and struggles to find a perfect solution if the parameters are not set optimally. The N-queens problem is a great way to show the differentiation in performance of search optimization algorithms.

References

Thada, V.; and Dhaka, S. 2014. Performance Analysis of N-Queen Problem using Backtracking and Genetic Algorithm Techniques. *International journal of computer applications* 102(7): 26–29.