CSC 180-01
Mini-Project 4
Friday November 15

Solving n-queens Problem Using Genetic Algorithm

By
Warren Quattrocchi (Team Leader) ID# 219714665
Janus Kwan ID# 212910387
Andrew Wright ID# 218771424

(1) Problem Statement

This project serves as practice with genetic algorithms by using Distributed Evolutionary Algorithms in Python (DEAP), the most popular Python library for evolutionary computation.

The n-queens problem was first invented in the mid-1800s as a puzzle for people to solve in their spare time, but now serves as a good tool for discussing computer search algorithms. In chess, a queen is the only piece that can attack in any direction. The puzzle is to place a number of queens on a board in such a way that no queen is attacking any other. In this project, DEAP is used to solve the 8 queens puzzle. The 8 queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens attach each other. The eight queens puzzle is an example of the more general N queens problem of placing n non-attacking queens on an $n \times n$ chessboard, for which solutions exist for all natural numbers with the exception of $n = 2$ and $n = 3$. One way to describe the board is to say it has a cost of 0, because there are 0 pairs of queens attacking each other. This can be generalized to say the cost of a given n-queens board is equal to the sum total number of distinct pairs of queens that are in the same row, column, or diagonal.

(2) Methodology

In the first algorithm, each queen is randomly placed on the board, which is designed as an array from 0 to n^2 - 1. The fitness function determines how many attacking pairs exist on the board. The algorithm, written by Warren and Andrew, finds the attacking pairs in rows, columns, and diagonals. It also finds the number of invalid boards, where queens are placed in the same position. The fitness score is the number of attacking pairs, and the goal is to find the lowest score. True solutions to the puzzle are fitness scores of zero. The selection function uses a K-Tournament with a value of 3 to select the best individuals in the population, and this is used in the next population. Mutation is also allowed to occur. The process is run through the genetic algorithm for 100 generations. In the second algorithm, the board is built using the indexes of an array as the rows, and the values of each element as the column. The fitness function still finds attacking pairs, but no row collisions or invalid boards can occur. This algorithm is also run for 100 generations. The average and minimum for each generation are plotted out and the best individual and fitness value are printed as a chessboard.

(3) Experimental Results and Analysis

First Algorithm, Table by Andrew

Trial	Tourn Size	Pop. Size	Cross -over Prob.	Mut. Prob.	#Gens	Avg	Min
1	3	1000	0.5	0.2	100	2.435	1
2	6	1000	0.5	0.2	100	2.804	1
3	1	1000	0.5	0.2	100	36.009	3
4	3	100	0.5	0.2	100	4.83	2
5	3	100	0.5	0.2	1000	3.33	1
6	3	1000	0.5	0.2	1000	1.989	1
7	3	1000	0.5	0.2	100	1.319	0
8	3	1000	0.1	0.2	100	2.193	1
9	3	1000	0.5	0.5	100	4.882	1

Conclusions: Tournament size should be small, but not too small. Larger population size with small # of generations, is very bad. Having a larger mutation probability is also apparently bad. Sometimes running the algorithm more than once with the same parameters can achieve better results.

Second Algorithm, Table by Andrew

Trial	Tourn Size	Pop. Size	Cross -over Prob.	Mut. Prob.	#Gens	Avg	Min
1	3	1000	0.5	0.5	100	0.706	0
2	6	1000	0.5	0.5	100	0.695	0
3	1	1000	0.5	0.5	100	7.968	2
4	3	100	0.5	0.5	100	0.69	0
5	3	100	0.5	0.5	1000	0.59	0
6	3	1000	0.5	0.5	1000	0.817	0
7	3	1000	0.5	0.5	100	0.744	0
8	3	1000	0.1	0.5	100	0.756	0
9	3	1000	0.5	0.1	100	0.153	0

Conclusions: This algorithm consistently achieves better results than the first algorithm. Having a lower population size and a higher # of generations seems to achieve the best results. Especially with a low mutation probability.

(4) Task Division and Project Reflection

For Project 4 we decided to spend most of the time working together to figure out the most efficient algorithm for our fitness function. In order to come up with a solution we brainstormed ideas together and were able to get a working algorithm for the position indexed representation, and from there we adapted it to work with the row indexed representation. After this we modified the 2 algorithms to work with any n > 3, which created much more interesting results for a larger n when run through the genetic algorithm. The main task divisions included Janus with the initial skeleton, Warren and Andrew with the fitness function design, and Andrew with parameter tuning. One thing we learned from this project was that the row index based representation had substantially better results, resulting in a min 0 nearly every time due to the lack of row collisions and invalid boards, while the position indexed representation would tend to quickly produce a min of 1 after around 20 generation, and never find an optimal solution.