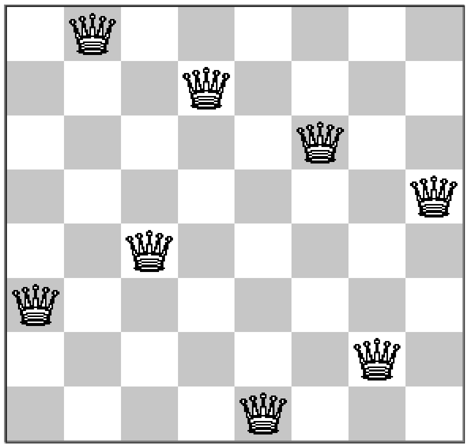
CS 461
Program 2 – Genetic Algorithms

The second programming assignment of the semester is a demonstration of genetic algorithms.

The *n-queens problem* is as follows: In chess, a queen can move forward or backward, left or right, or along either diagonal. Chess is played on an 8x8 board; this problem is a generalization to an *n x n* board, where *n* is any integer > 3. (There is no solution for the 3- or 2-queens problem, and a 1x1 board is trivial.) The problem: Place *n* queens onto the board such that none are attacking any other. An example of a solved 8-queens problem is:



There are other solutions, of course. For this program, you will ask the user for a value for *n*, and find a solution for the *n*-queens problem.

Note that each file (vertical column) has exactly one queen. Thus, we can model any position by a list of *n* integers in the range from 1 to *n* (or 0 to *n*-1, it doesn't matter), corresponding to the rank (row) the queen is on for that file.

Fitness. For a fitness metric, use the number of queens that can attack each other. Or it may be clearer (and easier to use some algorithms) if fitness is an increasing number; thus you may choose to use something based on the size of the problem ($n^2/2$) or an arbitrary constant (such as 100) minus the number of attacking queens.

Reproduction: crossover. For this problem, the merit of a particular rank depends in part on the rank value of adjacent files. That is, the queen being in the top rank of the second column above is a good position only because of the positions of queens on adjacent ranks; there's nothing particularly good or bad about the top row as such. Thus, crossover should preserve at least some adjacency information. For deciding how much each parent should contribute to offspring, choose a position randomly, call it *k*; everything from the beginning to index *k* should come from one parent, everything else from the other. At least one value should come from each parent.

OR, choose each value independently from either parent, with equal probability. Your option.

Reproduction: mutation. After values are chosen from each parent, each value has a 1% chance to be modified, by +/- 1. Mutations in each direction are equally likely (that is, 1% chance of mutation; if there is to be a mutation, mutation up or down are equiprobable). Use modular arithmetic for the mutation, 'wrapping around' to the other side of the board if a mutation would take it off one edge.

Population size. Start with a population of at least 200 randomly-generated arrangements.

Generations. There are two main methods commonly used:

- The *N* members of each generation are ranked by fitness function, and the *N/2* members below the median are discarded. Then pairs of members are chosen at random and new members generated.
    - In one variant, an additional *N/2* members are generated, with half the population being made of members of the previous generation who survived.
    - In another, *N* members are generated, forming a new population, with all members of the old generation being discarded.
- The poorest-performing individual member of the population is removed. Two members of the current population are selected at random and produce a new member. In this variant, a new generation is said to occur when *N* new members have been produced.
- In either case, random selection can either be uniform (each member eligible for reproduction equally likely to be chosen) or weighted by fitness (so that the most fit are more likely to reproduce).

Results. For each generation, report the average fitness score, and the best and worst score. Continue iterating until a member is produced with no captures/attacks possible (fitness of 0). Print this result.

Technical requirements. This program may be implemented in C, C++, C#, Java, or Python. No third-party libraries should be required to run your program.