CIS 421 Artificial Intelligence

Fall 2019 **Dr. Laura Grabowski**

Assignment 3: Genetic Algorithms – *N*-Queens

WARNING: This assignment includes post-experimental analysis. Complete and test your program well in advance of the deadline, so that you have time to finish all the experiments and analysis!

PROGRAM

You will implement the genetic algorithm approach that we discussed in class for solving the *N*-Queens problem (with a couple changes of my own). Remember that this problem requires that *N* chess queens be placed on an *N*N* chessboard in such a way that no queens can attack each other. For your assignment, you will implement a general solution for **N-Queens**. The details of the algorithm design follow. For the sake of reasonableness, you may assume $N <= 100$.

A note on random (pseudo-random) event generation

There is a fair amount of discussion around on how best to model random events. We have enough to deal with in implementing a genetic algorithm, there is no need to borrow more trouble. For all the many different random events that you need in your GA, you need only use the common built-in random value seeds and generators available in the programming language you are using. The GA that you are building does just fine with random sampling from a uniform distribution. No fanciness is needed. I recommend that you establish a simple random event method and use if for your entire program. You can always generate a random value in [0, 1] and use that value for any range by scaling as needed. Again – not fancy.

Representation

The genotype is a permutation of 1, 2, …, N and a genotype $g = <i_1, i_2, \ldots, i_N>$ denotes a unique board configuration where the n^{th} column contains one queen in the $i_n{}^{th}$ row. Put simply, the position in the vector gives the column on the board, and the number in that position gives the row on the board. This scheme automatically eliminates vertical and horizontal constraint violations, so that your program will only have to check for diagonal violations. This is the same representation discussed in your textbook in the simulated annealing chapter, and that we have discussed in class.

Initialization

Seed a **population of size of *N* * 10** with randomly generated genotypes. This population size is pretty arbitrary, so talk to me about adjusting the population size if your program has trouble finding solutions.

Termination

A solution is found, or 1,000 generations have passed.

Fitness

The fitness function is simply the inverse of the number of conflicts (*i.e.,* the number of attacking queen pairs in a configuration). Since a solution will have no conflicts, your fitness calculation will have to ensure that no division by 0 can occur. If *q(p)* represents the phenotype quality, meaning the **number of attacking queen pairs**, then the fitness can be calculated as

$$\frac{1}{q(p)+\varepsilon}$$

where ε is a small constant value (*e.g.,* 0.0001). Be sure that you count each pair only once.

You will have to determine a method to check a given configuration for conflicts, given a genotype (as described above). **Be sure to explain the conflict-checking method you use in your comments.** You may adapt the conflict checking method from the textbook code (for simulated annealing) if you wish. Efficiency is nice, but I won't get bent out of shape if you use a more brute force method.

Parent Selection:

We will use a population method for choosing parents called *tournament selection*. From the population of size *N*, you will select 10% of the individuals in the population as a "mating pool," using *tournament selection:*

```
Set current_member = 1
WHILE (current_member <= N * 10%) {
       Pick 3 individuals randomly, with or without replacement[1]
       Select the best of these 3 by comparing their fitness values
       Denote that chosen individual as i
       Set mating_pool[current_member] = i
       Set current_member = current_member + 1
}
```

Once the *N* * 10% individuals have been selected, you may use any method you want to try for deciding the pairs of parents (pick randomly, rank by fitness, *etc.*). From these *N* * 10% parents, you will create *N* * 10% children. **Note:** Be sure that you have an even number of parents.

Variation Operators

You will use the methods for crossover and mutation for this problem as discussed in class. Crossover will be done as described below with the pairs of parents that you arrived at in the parent selection process. Mutation will be done probabilistically (as described below), so mutation may not always be performed on newly created offspring genotypes.

- **Crossover.**
  - Determine which parents will be paired. You may do this any way you want to. Popular approaches include pairing according to fitness (*i.e.,* the two most fit individuals are paired, then the next 2 most fit, and so on), pairing randomly, and pairing according to the order the parents were chosen. Please indicate in your comments and discussion what method you used.
  - For each set of parents, choose a random position in the genotype as the crossover point. The random position will be chosen for EACH crossover event (*i.e.,* there will probably be different crossover points for different sets of parents in the same generation).
  - Cut both parents into two segments after the randomly chosen position.
  - Copy the first segment from each parent to the first segment of each of the two children (copy first part of parent 1 to child 1, the first part of parent 2 to child 2).

[1] Sampling without replacement is preferred. Using sampling with replacement may lead to choosing more low fitness individuals for the mating pool, and therefore slow down convergence.

  - Scan parent 2 from left to right, filling in the second segment of child 1 with values from parent 2, skipping any duplicate values (*i.e.,* values that are already in the child's genotype). Repeat this process with parent 1 and child 2.
- **Mutation.**
  - First, decide if a mutation will occur. Mutation will be done probabilistically. Mutations will occur 10% of the time.
  - Method: Select 2 positions in the genotype at random and swap the values. The random positions will be chosen for EACH individual that will be mutated.

Survivor Selection / Population Update
After children are created and mutated, your algorithm must evaluate the fitness of the newly created children. Then, you will update the population using either of the two following methods:
1. **Age-biased**: The children replace their parents in the new generation. This method is somewhat easier, but may converge more slowly.
2. **Fitness-biased**: Each child's fitness is evaluated, and all individuals (old population plus newly created offspring) are ranked/sorted by fitness. The N * 10% individuals with the lowest fitness are discarded (or, viewed the other way around, the N – (N * 10%) most fit individuals are retained). In other words, your algorithm will use a constant population size. This method is a bit more complicated to implement, but will probably converge more quickly.

The method you use is up to you, but **be sure to state in your comments which option you are using (age-biased or fitness-biased)**.

PROGRAM OUTPUT AND ANALYSIS
Because the GA is not deterministic, you need to run it a number of times to understand its average behavior. **You will run your completed GA 25 times, using a different random seed for each run, with N = 12**. For each run, print the (1) generation number an individual with the optimum fitness value (*i.e.,* a solution) was found, and (2) and the solution representation vector to an output file (NOTE: be sure your output file is set up to be appended, not to be overwritten). You will be keeping track of all the solutions that your program finds in the 25 different runs. If your program does not find a solution in a run, you will record the maximum generation time as the time a solution was found. In evolutionary computation, that is usually code for "this population failed to find a solution." You may also occasionally generate a solution in the initial population. Record that as solution found at Generation 0. If either one of those things happens a lot, it's possibly indicative of a bug. You may want to talk to me. **Submit the output file from your 25 experimental runs along with your source code.** What you're doing here is supplying me with the raw data from your experiments.

Using the data that you collected, you will perform the following analysis of your program's performance:
- Find the maximum, minimum, and average of these 25 recorded generation values. These calculations will require some post-processing of the output file. You may write a short program to do that yourself, or use a tool such as Matlab or Excel. You do not have to submit your code for the analysis.
- Create a scatterplot of the 25 values with the run number (1-25) on the x-axis and the generation number on the y-axis. **Include the maximum, minimum, and average generation numbers, annotated directly on the plot or shown in a legend for your plot**.

- Report the number of *different* solutions (not *unique* solutions) your program found over the 25 runs. Factoid: for 12-queens, there are 14,200 solutions, 1787 unique solutions if you account for rotations and reflections of board configurations. You can look up numbers of solutions for other *N* online.
- Include with your plot a "sketch" of the design choices that you made for your implementation, including:
  - Basic parameters (whether you changed them or not) – population size, mutation rate, crossover probability, max number of generations, size of mating pool.
  - Parent selection – did you sample with or without replacement in tournament selection?
  - Parent pairing – how did you pair sets of parents?
  - Conflict checking – what method did you use to identify conflicts between queens?
  - Survivor selection – did you use age- or fitness-biased selection?
- Put your plot and other information into a word-processed or typeset file and include it in your zip file with your program source code.

WHAT YOU WILL TURN IN
You will turn in all of the following files**:**
1. Source code.
2. Output file.
3. Discussion (.pdf, .doc, ,docx, formats are acceptable – PDF is preferred), including your plot and the sketch of your program design choices.

IMPORTANT COMPILATION NOTE: All programs must compile under Linux on the CS lab (Dunn 302) server.

Grading breakdown for the assignment:
- 20% -- Program generates results.
- 10% -- Program was run 25 times (with different random seeds) and data were collected.
- 30% -- Program produces reasonable results – Benchmarks:
  - Around 50% (minimum) of the populations find a solution.
  - Different solutions are found by different populations.
- 10% -- Output file with raw data is included.
- 10% -- Analysis and program design sketch are included.
- 20% -- Code is well designed, formatted, and commented. Note that programs with severe violations of the published style and commenting guidelines will receive a 0.