8 Queens Puzzle: Evolutionary Algorithm Attempt

Seth Cram
University of Idaho - College of Engineering
Evolutionary Computation - CS572
Moscow, Idaho, United States of America
cram1479@vandals.uidaho.edu

***Abstract*—This document is a project for CS572 Evolutionary Computation at the University of Idaho and serves to lay out the problem, representation used, each algorithm in detail, and explains what the collected data means.**

***Index Terms*—individual, population, trait, queen, chessboard(board)**

I. Introduction

First, a brief introduction to the problem. The 8-queens puzzle is trying to place 8 queens on an 8 by 8 chessboard without any of the queens threatening one another. Threatening one another is defined as a chess piece being able to attack another in a single move. Increasing the complexity of this puzzle is how queens are "able to move any number of squares vertically, horizontally, or diagonally" [3]. The 8 queens puzzle is a special case of how to place n queens on an n by n chessboard such that no queens threaten one another.

A succinct history of the puzzle is included for completeness. The 8 queens puzzle was first proposed in 1848, and two years later, the first solution was published, and it was expanded to the n queens problem. A mathematical solution was proposed in 1874 and later refined. Much later in 1974, the puzzle was used to showcase "structured programming" and the "depth-first backtracking algorithm" [1]. Both structured programming and the depth-first search algorithm are important concepts taught early on in the Computer Science discipline.

Some stipulations laid out for the students include a population of 100 random individuals and evolving the population for 1000 iterations without increasing its size using a steady state algorithm.

The 8-queens puzzle is an excellent candidate for an evolutionary algorithm since the fitness of each individual is easily quantifiable as the number of queens threatening one another. The puzzle is also easy to benchmark with its 4,426,165,368 possible queen placements and 92 different solutions [1]. The incredible size of the search space has posed a large hurtle to past participants in solving the puzzle, which makes an evolutionary algorithm a likely candidate for uncovering solutions with their immense computational power. In my representation, an individual is an 8 by 8 chess board with 8 queens placed on it. Each queen is represented by a tuple of two integers storing its x and y position from 0 to 7. My evolutionary algorithm attempts to solve the puzzle through progressively mating two fit individuals to create two children who inherit from their parents, possibly mutate these children, replace the two worst individuals in the population with the children, and iterating on this process 1000 times. My algorithm also specifies high fitness as bad and low fitness as desireable. Finally, during the course of this paper, each queen's position may be referred to as a trait of the board it resides on.

II. Methods

Representation	tuple of 2 integers per Queen
Recombination	uniform crossover
Mutation	integer incrementation/decrementation with fixed probability
Parent Selection	fitness-proportionate

A. Creation Functions

Although not explicitly required, I split the creation of the population and the creation of an individual into separate functions. Creating the population involves specifying the desired population size, and the upper x, y bounds of the board and fielding the resultant population. The individual creation function takes in the desired number of traits for the individual and the board x, y bounds. It uses a uniform distribution to randomly decide the x, y coordinates of each of the eight queens and returns the individual as an array of tuples. Crafting the individual also ensures none of their queens occupy the same space through iterative comparison and regeneration of more random coordinates if required. A simple manual test verifying no queens share the same position and no queen's position it outside the board coordinates vetted this function.

B. Fitness Functions, Class and Implementation

The "EvalFitness()" function takes in an individual as an array of tuples and returns their fitness value. It calculates the fitness value through comparing every queen on the board to every other queen on the board. This comparison involves taking the difference between the queens' positions and if it's 0 in y or x, increasing the fitness score. Otherwise, take the slope between the two queens and if it's absolute value is one, that means the two queens are diagonal of one another, so once again increase the fitness score. Although, eight Booleans are necessary to ensure that the evaluated Queen doesn't mistakenly threaten another Queen hiding behind a threatened Queen. In other words, queens can't move through other queens. Therefore, the fitness score will only be an odd number if some queens share the same space, since for a queen to be threatening another queen, that same second queen must

also be threatening the first queen. The fitness function was approved of by creating several individuals through manually assigning their queens, figuring out the expected fitness score through pen and paper, and comparing the function's output fitness score.

In order to pair the individuals with their fitness values after evaluation, I created the data class "PopulationFitness" with the individual array of tuples and fitness integer value. Although not an optimized approach, every new iteration I evaluate each individual's fitness and add the individual along with their fitness value to an array of PopulationFitness values. This organization allows me to sort the individuals in ascending order according to their fitness. The sorting of these individuals uses the builtin list.sort() with my "getFitness()" function as a key to compare each individual through their fitness score. This saves me the work and time of manually sorting the list myself. The population needed to be sorted in this fashion to select parents properly.

I was able to gather the max and min of the population-fitness pairings list utilizing the builtin max() and min() functions and once again using getFitness() as the key for both. Unfortunately, the mean function I tried using didn't function in the same way since it wasn't builtin, so I summed the fitness values manually and divided them by the population size to get the arithmetic average. I casted the average fitness to an integer merely out of preference. Each iteration, these three statistical values of max, min and mean would be appended to their respective arrays for plotting after the evolution iterations capped out.

C. Breeding Functions

Parents were selected using my "BreedSelection()" function. It takes in the population as an array and optionally a Boolean value to display the distribution being used to select the parents. The function expects the population to be sorted in ascending fitness order (low/good to high/bad). In order to select parents of higher fitness more often, I used a half-normal distribution bounded from 0 to 99. I discovered this approach through researching manipulation of normal distributions and happened upon a Stack Overflow answer which helped immensely. That answer is linked at the end of the paper as a reference. A histogram of the resultant half-normal distribution the parents are drawn from is shown below.

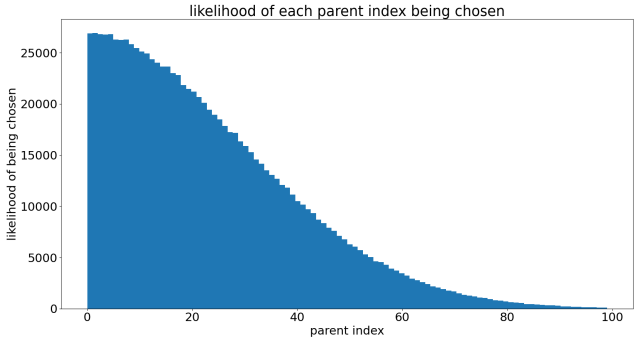

Fig. 1. Half-normal distribution used to choose the parent indices

As seen above, the y-axis is scaled incredibly high. This is done so the chance of choosing one of the values closer to 100 is visible. Although the y-axis is scaled up, these bins in relation to one another are accurate. Despite how it's slower, I decided to use a distribution instead of just choosing the two most desireable individuals, since the instructions stated that the selection "should be slightly weighted towards more fit individuals" so I reasoned that just choosing the two most fit boards to breed didn't satisfy that. I could have used tournament style selection, but decided that I preferred the smooth distribution that I knew more about compared to the newer concept of taking x amount of individuals and choosing the best fit one for each parent.

To create two new individuals, my function "Crossover-Breed()" accepts two parents and returns two children. It assumes both parents have the same number of traits, and randomly assigns queen positions from each parent, regardless of the trait's or board's fitness. This approach of randomly assigning each parent trait to a child, with the other child receiving the other parent's trait, is called Uniform Crossover. I could have used a 1 to n point selection, but decided that the more random approach I followed would take longer but ultimately lead to a more diverse population.

At this point in my development process is where I realized that always ensuring the next queen placed on the board is in a unique position is more difficult than it's worth. At first, I attempted to verify each child's queen didn't conflict with another and if it did I overwrote the conflicting trait with the other parent's trait that wasn't taken for this child. I soon realized that in some scenarios, both parent's traits could be duplicates of other queens' positions. The most feasible way out seemed to apply a 100% occurring mutation for the conflicting trait. I decided that applying the mutation to avoid queens clashing didn't satisfy the project requirements to, "Create a crossover function... and create two children, which should inherit from the parents" since mutating the conflicting child's trait means not all children traits would be inherited from the parents. So, I decided to leave mutating in a separate step and allow the crossover of conflicting queens vying for the same space. Selective pressure was already being applied for queens to not be near or on top of one another.

D. Mutation Function

I created a ”Mutate()” function that took in a child and returned the Boolean of whether a mutation had occurred or not. It used a uniform distribution to generate several random numbers. There was a fixed 20% chance that a mutation happened. A large amount of mutation is generally thought of as detrimental in the world of Evolutionary Computation, so I decided my percentage of mutation should be relatively low. A mutation happening corresponded to one of the randomly chosen queen's positions receiving an increment or decrement in its x or y coordinate. These mutated coordinates are pulled from another ”getMutatedCoord()” function where the tuple trait being mutated is passed in. The segmentation of this process into two functions was to repeatedly call getMutated-Coord() to re-mutate until a mutated coordinate was received that represented a queen placed in a unique position that was actually on the board.

Unfortunately, it's possible that a mutation in either direction can land on a queen, so the check for queen position uniqueness was scrapped. The deciding of whether an x or y coordinate got changed and whether it was an increment or decrement was left up to a uniform random number generator with 50/50 odds for either. Since only a single x or y coordinate of a single queen of the mutated child is changed by only a small margin of 1, the mutation is less likely to be harmful.

E. Replacement Function

Finally, the ”SurvivalReplacement()” function took the population as an array and the desired children to input and returned nothing since the population array would be changed instead. Since the project instructions specified that the function ”removes the two worst individuals” it was a simple matter to use array slicing and replace the last 2 individuals (most fit/worst) in my ascending fitness ordered data with the children.

III. Results

Unfortunately, I was unable to consistently solve the 8 Queens Puzzle. Most of the time, my evolutionary algorithm would find a best fitness individual of 4 or 2, but every once in a while, I'd find solutions with zero fitness. I used graphs taken from a best fitness run of 2 since that was the most common result.

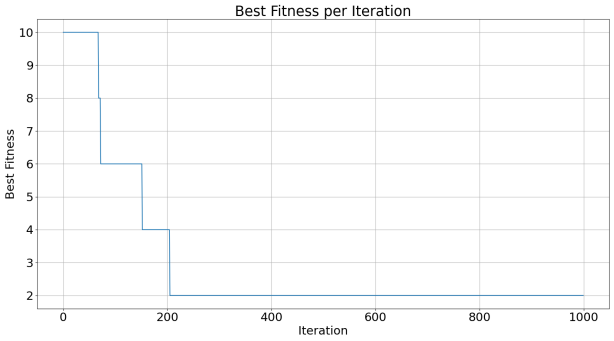

Fig. 2. Best fitness over 1000 iterations

As shown above, the evolutionary algorithm method fields progressively better results. The best fitness per iteration graph has much less common drops in fitness when compared to the other graphs. These drops in fitness represent breakthrough points where a new child is created via crossover and mutation that has a better fitness than any of its predecessors. Due to how the survival function only ever replaces the worst individuals, the best fitness for a future iteration will never be less than a previous one. This is great for finding a local optima, but could pose a problem when attempting to scour the entire fitness landscape.

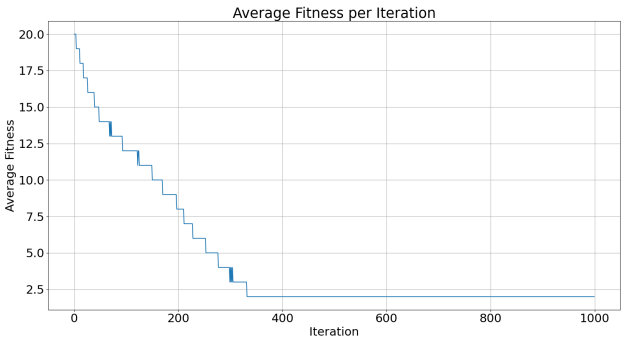

Fig. 3. Average fitness over 1000 iterations

The rapid drops and rises of the graph are because of my decision to round each iteration's average fitness to the nearest whole number. Peaks stem from the worst fitness per iteration occasionally increasing and worsening the average since one of the new children is an outlier due to a poor mutation, crossover, and/or parent selection.

Fig. 4. Worst fitness over 1000 iterations

As seen above, the worst fitness per iteration graph follows the general shape of the other graphs, but is permeated with far more jagged peaks. I attribute the shape of the graph to poor mutations, poor crossover, and/or poor parent selecting. A mutation doesn't always occur in a positive direction since it's inherently random. Just because a child's parents have high fitness doesn't mean the child will have high fitness since they're not a clone of a parent. Due to the half-normal distribution I applied to randomly selecting the parents, the new child won't always get the most fit parents either.

When timing my algorithm without displaying the graphs, it took around 3 seconds on my machine. Since I was curious, I decided to increase the number of iterations to see how long it'd take my algorithm to find the answer. I got up to 100,000 iterations which took around 3 minute before stopping.

Repeatedly running my evolutionary algorithm from scratch in an iterative loop until a solution is found is much more efficient than increasing the iterations on a single run. The first time I did so, it took around 45 runs to find a solution, the next time only 27 runs, the 3rd time 2 runs, and the 4th time 14 runs. Expanding this approach to other issues only works if the developer is aware of the goal the evolutionary algorithm is trying to reach. When a solution was found, most other members of the population quickly converged to it. So, if the desired goal was to find all solutions to the 8 queens puzzle, some new functionality such as blacklisting know solutions would need implementing. Another aspect may also need adding to allow individuals to spread more thinly over the fitness landscape without converging near a blacklisted solution.

IV. Conclusion

In conclusion, my evolutionary algorithm failed 95% of the time with the specified stipulations, but it shows promising progress, since it succeeds the other 5% of the time. So, a genetic algorithm seems useful when trying to solve easily quantifiable problems with finite numbers of gene/trait combinations. Rerunning an evolutionary algorithm until an end-goal is achieved is a viable approach but only if the end-goal is easily checked for and time is nonessential. Increasing the number of iterations isn't the answer to finding a solution, one of the pre-existing stipulations, approaches, or representations needs to be changed. If I were to try and better my algorithm, I'd change the representation from using tuples of integers to single integers to constitute one queen per column's placement. This would ultimately reduce the search space drastically from 4,426,165,36 possible queen placements to merely 8! (40,320), requiring far less computational power and time [1].

References

[1] En.wikipedia.org. 2022. Eight queens puzzle. [online] Available at: https://en.wikipedia.org/wiki/Eight_queens_puzzle [Accessed 6 September 2022].

[2] Stack Overflow. 2022. How to generate a random normal distribution of integers. [online] Available at: https://stackoverflow.com/a/37412692/13046931 [Accessed 1 September 2022].

[3] En.wikipedia.org. n.d. Queen (chess). [online] Available at: https://en.wikipedia.org/wiki/Queen_(chess) [Accessed 1 September 2022].