CS 7770 Intro To Computational Intelligence

Fall 2024 Project 1

Christian VanMeter

September 28, 2024

Contents

1 Problem Description

Project 1 asked us to solve a problem using an evolutionary algorithm. I chose to solve the N-Queens problem which involves placing a queen on each column of a N by N chess board so that no queens attack each other. I find this problem interesting because as the size of the board increases it would be pretty difficult for a human to solve it but with a computer it can solve the problem pretty quickly depending on the method you use to solve it. The one constraint on the problem is that the board must be a square board (N by N board). I provide comparisons between two selection operators $(\mu + \lambda)$ and $(\mu\ ,\ \lambda)$ with varying population sizes, board sizes, and mutation rates (for graduate student portion).

2 Technical Description

For my implementation I chose to encode the queens position as a chromosome with N elements where the each index corresponds to a column on the board (first index = 1st column of board) and the value stored at the index corresponds to what row the queen is placed on (value of 0 means the queen is on the first row of the board). I chose to implement $(\mu + \lambda)$ and $(\mu\ ,\ \lambda)$ as selection operators. $(\mu + \lambda)$ involves taking both the parents and children and only selecting the best μ (population size) individuals based on their fitness scores for the next generation. $(\mu\ ,\ \lambda)$ involves only the λ children carrying on to the next generation regardless of their fitness scores. Fitness scores are calculated by recording the number of non-attacking queen pairs. The maximum fitness score that can be achieved when the problem is solved is $\frac{N \cdot (N-1)}{2}$. For crossover, two parents will be randomly selected from the population and a single point crossover point will also be randomly selected to create two new children. With mutation a random value will be chosen and if it is lower than the mutation threshold a single random queen position will be changed to a random value. I provide comparisons between two selection operators with varying population sizes, board sizes, and mutation rates (for

graduate student portion) to see how these variations effect the solution speed.

3 Algorithm Design

I designed my algorithm using Python along with Numpy. The flow of how my algorithm runs based on the two selection operators is shown by the following lists below:

$(\mu + \lambda)$ Implementation

1. Create random individuals of size N based on population size (user input)
   (a) Calculate fitness scores
   (b) Sort population based on fitness scores
   (c) Set current generation value to 1
2. Loop that runs until maximum generations value is reached (user input)
   (a) Check if solution is found (max fitness achieved). If a solution is found the function will stop and the solution will be returned to the user.
   (b) If no solution is found we create the next generation
      i. Perform crossover with two random parents from population with a random crossover point to create two children
      ii. Each child will undergo mutation if a random value is selected that is lower than the mutation rate (user input). If mutation occurs a random queen will be changed to a new random row position
      iii. Repeat steps i and ii until children of size N have been created
      iv. Calculate fitness scores of the children
      v. Combine population and children into single list
      vi. Sort list based on fitness scores and return the best N (population size) individuals

(c) Increment current generation value and go back to step 2

(μ , λ) Implementation

1. Create random individuals of size N based on population size (user input)
   (a) Calculate fitness scores
   (b) Sort population based on fitness scores
   (c) Set current generation value to 1
2. Loop that runs until maximum generations value is reached (user input)
   (a) Check if solution is found (max fitness achieved). If a solution is found the function will stop and the solution will be returned to the user.
   (b) If no solution is found we create the next generation
      i. Perform crossover with two random parents from population with a random crossover point to create two children
      ii. Each child will undergo mutation if a random value is selected that is lower than the mutation rate (user input). If mutation occurs a random queen will be changed to a new random row position
      iii. Repeat steps i and ii until children of size N have been created
      iv. Calculate fitness scores of the children
      v. Discard population and only keep children
      vi. Sort children based on fitness scores and return the children list
   (c) Increment current generation value and go back to step 2

The two selection operators were ran with permutations of the values below:

- Number Of Queens: $[5, 8, 10]$
- Population sizes: $[10, 100, 250, 500]$

- Maximum Number of Generations: [5000]
- Mutation Rate: $[0.10, 0.15, 0.20]$

Each permutation was ran 10 times and the generations to solve the problem was averaged over the iterations.

4 Experimental Analysis

Below is two tables showing the average generations required to reach a solution over 10 iterations based on the permutations of values above along with an analysis for each implementation. If the average generations is NaN that means that a solution was not reached in 5000 generations for any of the 10 iterations.

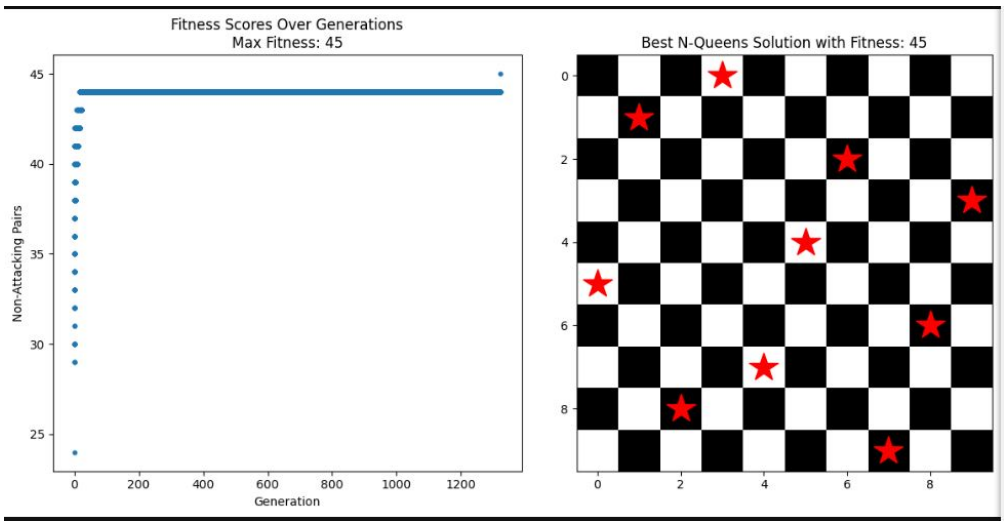
$(\mu + \lambda)$ Implementation

Number of Queens	Population Size	Mutation Rate	Average Generations (10 Iterations)
5	10	0.1	11.67
5	10	0.15	21.33
5	10	0.2	78.3
5	100	0.1	2.5
5	100	0.15	1.4
5	100	0.2	2.2
5	250	0.1	0.6
5	250	0.15	0.7
5	250	0.2	1.0

5	500	0.1	0.1
5	500	0.15	0.4
5	500	0.2	0.2
8	10	0.1	81.0
8	10	0.15	1708.0
8	10	0.2	354.67
8	100	0.1	107.11
8	100	0.15	19.75
8	100	0.2	29.75
8	250	0.1	44.9
8	250	0.15	15.3
8	250	0.2	15.8
8	500	0.1	12.0
8	500	0.15	11.5
8	500	0.2	9.9
10	10	0.1	NaN
10	10	0.15	425.0
10	10	0.2	1638.0
10	100	0.1	177.75
10	100	0.15	222.67
10	100	0.2	47.6

10	250	0.1	384.75
10	250	0.15	78.83
10	250	0.2	115.83
10	500	0.1	26.33
10	500	0.15	126.9
10	500	0.2	64.13

When analyzing the results of $(\mu + \lambda)$ it seems that the main factor in reaching a solution in the lowest amount of generations is the population size. This logically makes the most sense because the larger the population the larger search space we are able to cover. An interesting finding I had while running these tests is that often times the algorithm reaches a state where it is only off from a correct solution by a single pair of attacking queens in a relatively short amount of generations. However, it often takes a large amount of generations to fix this issue which I believe is due to it relying on mutation to fix the one issue because this method is highly exploitative and the population eventually gets to a point where it has little variance in positions. An example of this is provided in the Figure 1:A below. When it comes to varying mutation rates it seems that it sometimes hurts, helps, or causes very little change in some instances to the generations required to reach a solution depending on the population size. Generally for population sizes that are small as mutation rates increase it seems to increase the generations required significantly and it has the opposite effect as population size increases. I believe this is due to the fact that with smaller population sizes the higher mutation rate is causing too much exploration and not allowing the population to converge to a solution by ruining good candidates in the small search space.



$(\mu\ ,\ \lambda)$ Implementation

Number of Queens	Population Size	Mutation Rate	Average Generations(10 Iterations)
5	10	0.1	304.2
5	10	0.15	303.4
5	10	0.2	125.8
5	100	0.1	2.8
5	100	0.15	3.0
5	100	0.2	7.2
5	250	0.1	1.3
5	250	0.15	1.7
5	250	0.2	0.7
5	500	0.1	0.2
5	500	0.15	0.0
5	500	0.2	0.5

8	10	0.1	NaN
8	10	0.15	NaN
8	10	0.2	NaN
8	100	0.1	2427.71
8	100	0.15	2197.63
8	100	0.2	1471.33
8	250	0.1	1362.78
8	250	0.15	706.9
8	250	0.2	871.0
8	500	0.1	494.4
8	500	0.15	501.1
8	500	0.2	471.1
10	10	0.1	NaN
10	10	0.15	NaN
10	10	0.2	NaN
10	100	0.1	NaN
10	100	0.15	NaN
10	100	0.2	NaN
10	250	0.1	794.0
10	250	0.15	1587.0
10	250	0.2	2455.0

10	500	0.1	3290.5
10	500	0.15	3440.0
10	500	0.2	1798.0

When taking a look at the (μ , λ) results it seems that once again the main determining factor for a solution being reached in the least amount of generations is the population size. With this approach it seems that due to its highly exploratory nature it was not able to find a solution over the 10 iterations with these combinations of (number of queens, population size) of (8,10),(10,10), and (10,100). These failures to reach a solution is most likely due to the high exploration of the approach combined with the smaller search space when compared to the complexity of the problem which causes it to struggle to create enough exploration to reach a solution. When analyzing mutation rates on the number of generations required for a solution it seems to have varying effects depending on the number of queens and population size. I believe that more extensive experiments are needed to try and recognize a pattern that mutation rates have on the (μ , λ) implementation.