8 Queens Genetic Algorithm Report

Name: David Shaulov

ID: 317005403

Chromosome representation:

- I chose to represent the position of the queens as a list of length 8, where $list[i]$ is the row of the queen on the $j'th$ column.
- For example, for $[6, 3, 7, 4, 1, 8, 2, 5]$, the queen on the third column is on row 7.
- $8x8$ matrix representation is inefficient since if two queens occupy the same column it is an invalid solution.

Fitness function implementation:

- The fitness function measures the amount of pairs of queens that threaten one another – queens that are on the same row or diagonal.
- The less queen that threaten one another the higher the fitness.

Selection type:

- I have implemented both roulette and elitism selection.

Crossover types:

- I implemented single-point, two-point and uniform crossover types.

Mutation implementation:

- Given mutation rate p, with probability p the mutation functions swaps the places of two queens.
- Picking a random index and changing it to some random number is inefficient since it may choose a number already present in the list – meaning chooses a row that already has a queen on it, which is an invalid solution.

Experiment method:

- Since there is an element of probability, I ran each experiment 10 times and took the average time to first solution.
- I disregarded solutions generated by chance from initial population generation.
- Each experiment ran for 10,000 generations.
- Other than the experiments mentioned below, I conducted additional experiments, but they did not yield significant results, so I chose to omit them.

Low mutation, low population experiments:

Population Size	Crossover Type	Mutation Rate	Average Time To First Solution (Seconds)
50	Single-point	0.05	Did Not Find

50	Single-point	0.2	Did Not Find
50	Uniform	0.05	Did Not Find
50	Uniform	0.2	Did Not Find
100	Single-point	0.05	6.8
100	Single-point	0.2	15.86
100	Uniform	0.05	27.2
100	Uniform	0.2	25.46

Low mutation, high population experiments:

Population Size	Crossover Type	Mutation Rate	Average Time To First Solution (Seconds)
150	Single-point	0.05	8.26
150	Single-point	0.2	8.09
150	Uniform	0.05	5.99
150	Uniform	0.2	8.26
300	Single-point	0.05	3.59
300	Single-point	0.2	0.82
300	Uniform	0.05	3.28
300	Uniform	0.2	1.34
500	Single-point	0.05	0.53
750	Single-point	0.05	0.55

High mutation, high population experiments:

Population Size	Crossover Type	Mutation Rate	Average Time To First Solution (Seconds)
300	Single-point	0.3	1.61
300	Single-point	0.4	1.41
400	Single-point	0.3	0.73
400	Single-point	0.4	1.22
500	Single-point	0.3	0.73
400	Single-point	0.4	1.22

Brute force experiments:

- For the brute force method, I simply generated random lists of the elements $[1,2,3,4,5,6,7,8]$ until a valid solution was found.
- I ran the brute force algorithm 10 times and got an average time of 0.94 seconds to first solution.

"Smart" brute force experiments:

- Instead of generating random lists of $[1,2,3,4,5,6,7,8]$, I generated permutations – avoiding duplicate elements.
- I ran the "smart" brute force algorithm 10 times and got an average time of 0.005.

Conclusions:

- Best result for the genetic algorithm is 0.53 seconds to first solution, using a population size of 500, a single-point crossover and a mutation rate of 0.05.
- The genetic algorithm performs almost twice as fast as regular brute force, but "smart" brute force is much quicker.