Faculty of Engineering and Applied Science
Department of Electrical, Computer and Software Engineering

SOFE 4620U Machine Learning and Data Mining

Mini Project 2

Solving 8-Queen Problem Using Genetic Algorithm

Luke Hruda 100654666

Brittney Desroches 100649514

Adam Wong chew onn 100598499

Table of Contents

1. Problem Specification

The goal of this project is to use a genetic algorithm to solve the 8-Queens problem. The genetic algorithm has to attempt to place 8 queens on a chess board without having any of the queens being able to attack one another. This genetic algorithm must utilise all parts of the standard GA principle including: Chromosome Structure, Fitness Function and Mutation Operator.

2. Chromosome Structure

The initial rule of our genetic algorithm and chessboard set up is that no two Queens will ever be allowed to occupy the same row. To structure the chessboard, 8 sets of chromosomes were used. To be specific, a single chromosome represented a row of the chessboard. The chromosome itself was made up of three genes, this way, when converting the 3 bit chromosome to a decimal number it would yield a number between 0 and 7, which represents the queen's logical position in that row of the chessboard. This means that any possible "solution" would comprise 8 individual chromosomes for a total 24 genes. The image below shows how this would be represented logically and which position each Queen would be in based on the makeup of the chromosome.

	G1	G2	G3	
C1	0	1	0	= 2
C2	1	1	1	= 7
C3	0	0	0	= 0
C4	0	1	1	= 3
C5	1	1	0	= 6
C6	0	0	0	= 0
C7	1	1	0	= 6
C8	0	1	0	= 2

Figure 2.1 Chromosome and Gene Distribution

3. Fitness Function

The fitness function is rather simple for the NQueens problem. Given the precondition from section 2 that no two Queens shall ever exist together in the same row, that means the fitness function only has to check the columns and diagonals. To perform the fitness function, two loops are used, the first loop starts at the first row and goes through to the seventh row (second last row), the second loop starts at one more than the first loops current row ie, if loop one is at row 2, loop two starts at row 3 and continues until row eight (the last row). The second loop will iterate through the rows completely before loop one moves the next row. The second loop will check its current row's queen's position against that of loop one's Queen. If the three Gene values are the same, the function knows the two Queens are in the same column and an attack is possible. A conflict counter is used to track the number of conflicts each chromosome has and is incremented for both of the rows involved. To check for a diagonal attack, the function takes the difference in rows (diff row 3 and row 5 is 2) and checks the offset of the Queen in loop one to the Queen in loop two. If the offset matches the difference in row value, that means a diagonal attack is possible.

Code Snippet:

Check Vertical Attack Code

```
if(this.qPosition[rowOne] == this.qPosition[rowTwo]) {
    this.conflicts[rowOne]++;
   this.conflicts[rowTwo]++;
 }
```

The code above is used to check for Vertical attacks ie, the two Queens are in the same columns. It does this by checking if the position of the queen in row one is equal to the position of the queen in row two. "this" refers to the chessboard and "qPosition" is the array that holds the position of all the queens.

Check Diagonal Attack Code

```
if(Math.abs(this.qPosition[rowTwo]-this.qPosition[rowOne]) == rowTwo-rowOne) {
   this.conflicts[rowOne]++;
   this.conflicts[rowTwo]++;
}
```

The code above is used to check for Diagonal attacks. It does this by taking the absolute value of the position of queen two, subtracting the position of queen one from it and checking if that is equal to the row two number subtracting the row one value. Using the absolute value allows us

to check both diagonals at the same time rather than having to use two if statements one for each diagonal.

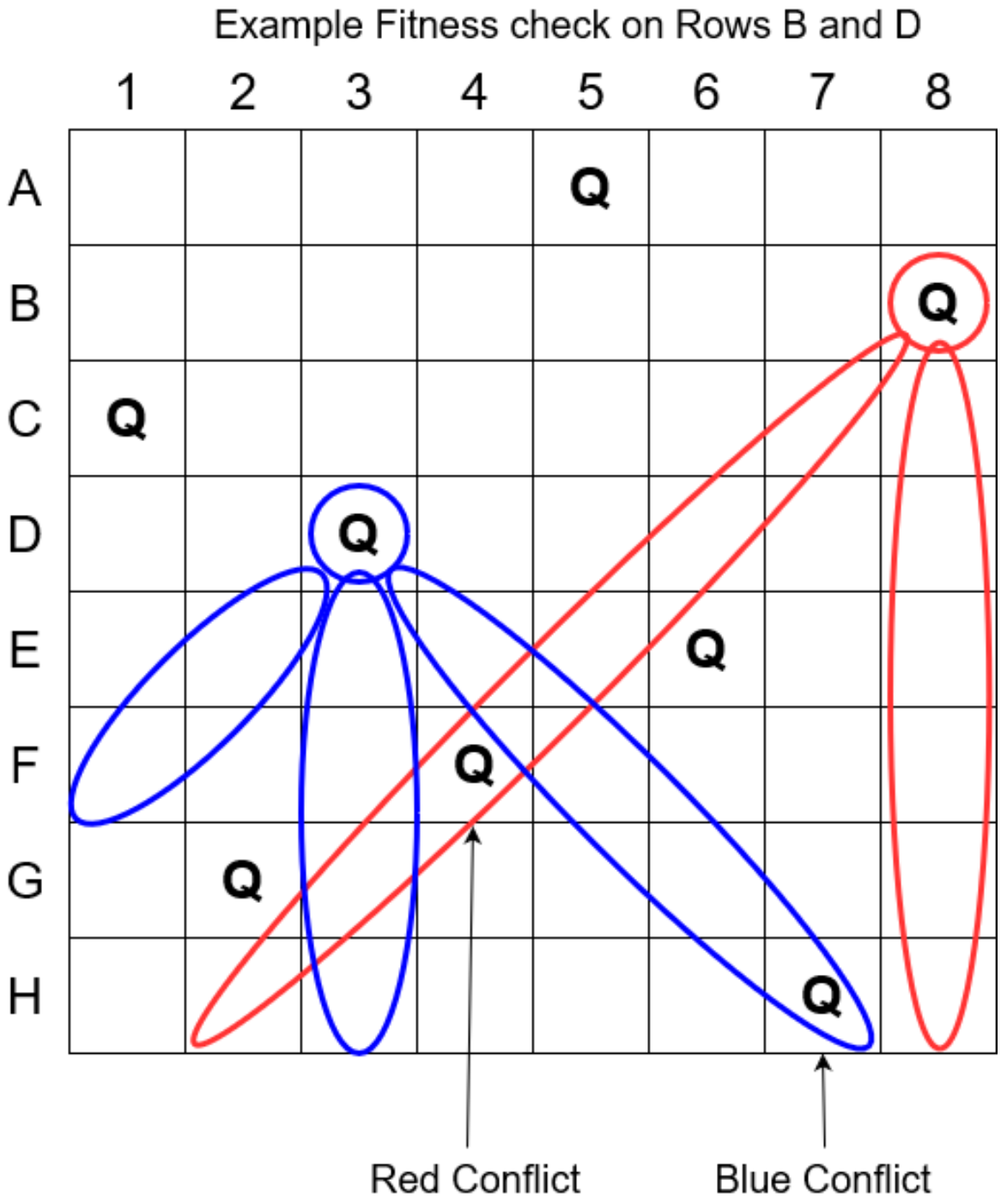

Figure 3.1 Sample Fitness Check Visualisations for Row B (red) and Row D (blue)

4. Genetic Algorithm

This section will discuss the general flow of the algorithm, how the Crossover operation works and how the mutation operation works.

General Algorithm Flow

The genetic algorithm starts by defining an array of chess boards of arbitrary size ‘n’. The boards are all initialised with a random composition of the queens within each row. The fitness algorithm is then run on all ‘n’ boards. This populates the conflicts array for each chess board. One at a time, the sum of the conflicts array is calculated. If the sum of the array is 0, that means it is a solution. The logical positions of the queens are combined to make a unique string for that specific solution ie: “25307461” is the first solution our algorithm found. This string is added to an array list to be used to ensure no duplicates are found. Once the fitness of all ‘n’ chess boards have been determined, the system prints out the Average, Best and Worst fitness of the initial generation along with if a solution has been found. For our algorithm, a lower value (less conflicts) represents a greater fitness. Once this has been completed, the algorithm works on creating the next generation of chess boards. The best half of the ‘n’ chess boards (n/2) are allowed to continue. To simplify we will say that there are 8 chess boards in total. So the best 4 are left alone for the time being. The worst four are removed from the list. To perform a crossover operation, which will be explained in further detail later, the first and second best fit board configurations are randomly mixed together to create the new 5th and 6th chess boards for the next generation. Likewise, the 3rd and 4th best fits are mixed to create the 7th and 8th chess boards of the next generation. Finally, the first, second, third and fourth best fits are allowed to mutate to create the new generation of chess boards.

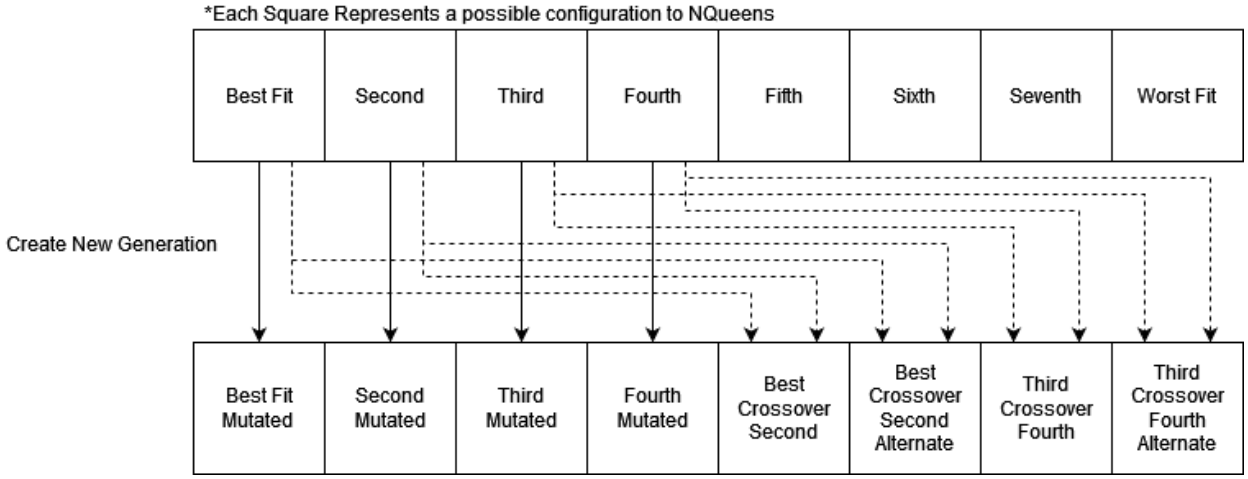

Figure 4.1 Creating New Generations

The reason that the fifth, sixth, seventh and eighth chess boards are not mutated is because their fitness has not been checked. Given that they are a combination of the best fits of their generations, their fitness may be better than their parents. The reason that the best four are mutated immediately is because their fitness has already been checked and checking their fitness again would yield no new results. The mutation operation will be explained in greater detail later. The new generation has the fitness of all its possible solutions checked again. Repeating the procedure if a new unique solution is found. The code provided is set to continue until all 92 solutions to the 8-Queens problem have been found.

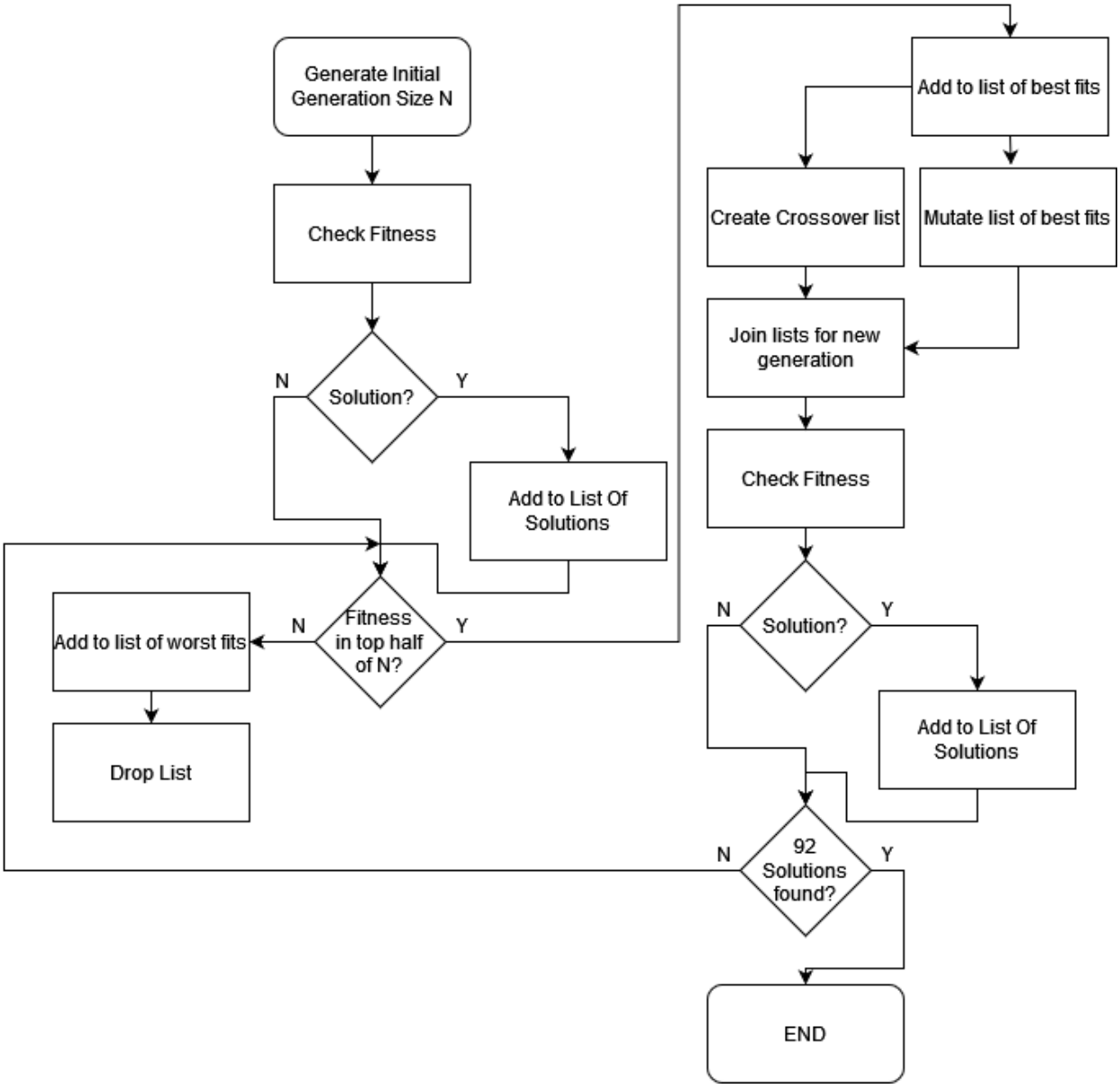

Figure 4.2 General Flow Diagram of the 8-Queens GA

Crossover operation

The crossover operation works by taking two sets of chromosomes and splicing them together to create two new sets of chromosomes, while leaving the parents intact to possibly mutate into better fitting solutions. For the whole generation, the top half are set aside to be crossed over. They are grouped in pairs of two to create two offspring solutions for the 8-Queen problem. That means, for a sample size of 100, the best 50 are spared to be crossed over, where the groups are composed of [1,2], [3,4],[5,6], … , [47,48],[49,50] where each number from 1 to 50 represents its fitness rank. For this example we will focus on just how the Best and Second best fitting solutions are crossed over. To create the two new possible solutions, a 'coin' is flipped to determine whether Child A or Child B receives Parent A's first chromosome. If Child A receives Parent A's chromosome based on the 'coin flip', Child B will receive Parent B's first chromosome. This process is repeated 7 more times for each chromosome. The final result is Parent A and B are intact while Child A and B are composed completely of chromosomes from either parent.

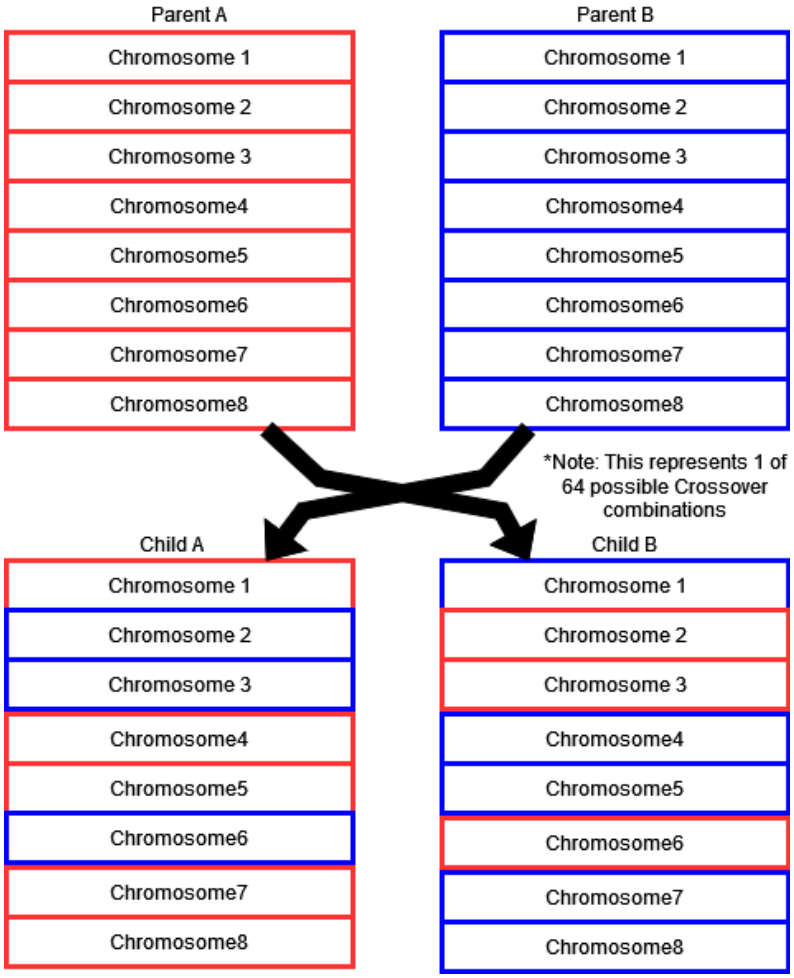

Figure 4.3 Crossover Operation Visualised

Mutation Operation

The mutation operation is the simplest part of this genetic algorithm. The best fit solutions are mutated with each generation to attempt to find a better fit. The algorithm randomly selects between 1 and 4 of the chromosomes to mutate. The mutation will randomly flip one of the genes in the chromosome. Shifting the position of the queen in that row either by 4, 2 or 1 positions based on which Gene is flipped in that chromosome.

5. Results

This genetic algorithm was able to find all 92 solutions to the 8-Queens problem. These have been provided in a sample file "Solutions.txt". Three sample solutions have been provided below.

```
Solution: 1 25307461  Solution: 2 24730615  Solution: 3 53047162
--Q-----              --Q-----              -----Q--
-----Q--              ----Q---              ---Q----
---Q----              -------Q              Q-------
Q-------              ---Q----              ----Q---
-------Q              Q-------              -------Q
----Q---              ------Q-              -Q------
------Q-              -Q------              ------Q-
-Q------              -----Q--              --Q-----
```

Figure 5.1 8-Queens Solutions 1, 2 and 3

During execution of the tests, an initial population of 20 boards was used. Therefore according to the algorithm, the best 10 would be allowed to mutate each generation and a new 10 boards would be made via the crossover algorithm. However this led to a total 3.9 Million generations being required to find all 92 solutions. During testing it was found that an optimal population size was ~800 chess boards per generation, leading to around 130k generations to find all 92 solutions. This also significantly improves the overall runtime of the algorithm, to an average of 3-4 minutes. Population sizes of 4-20 thousand were also tested. While using a sample size of 4 thousand chess boards did drop down to only 40k generations to find all of the solutions, this significantly increased the runtime to 8 minutes and beyond making it less optimal than using an initial size of 800-1000 chess boards.

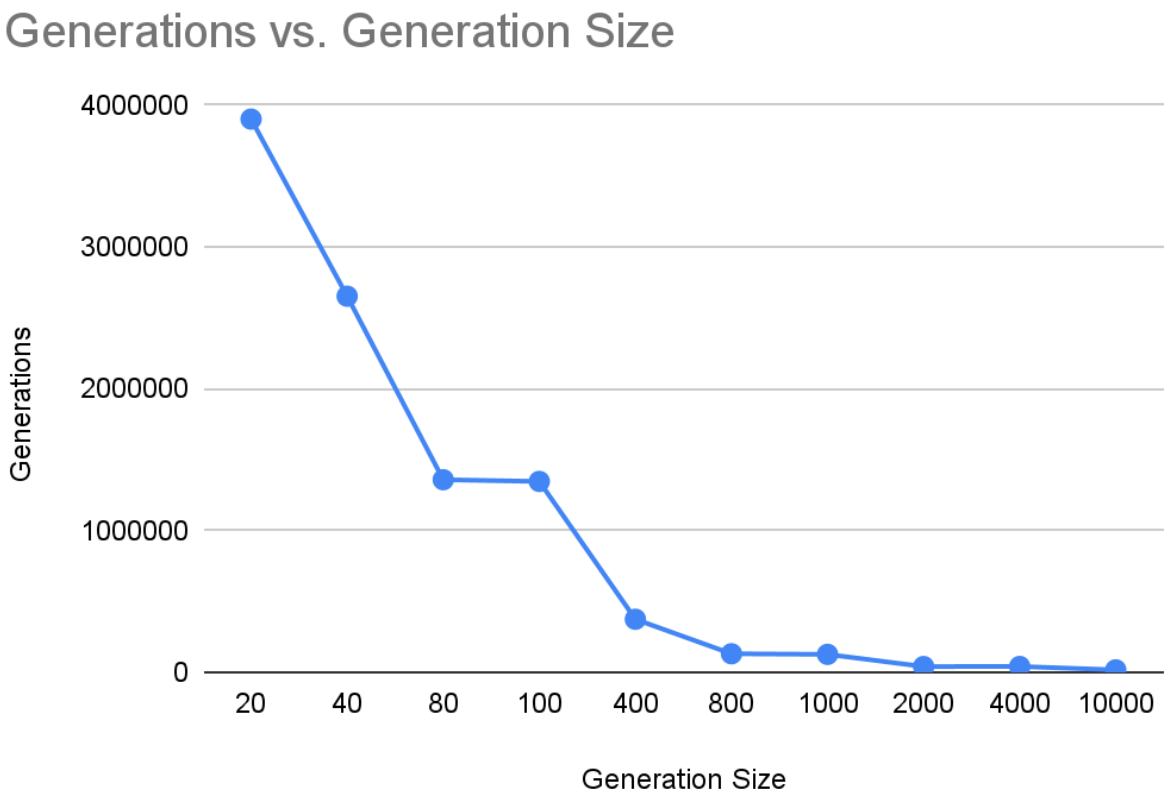

Figure 5.2 Generation Size vs Number of Generations Required to Find All Solutions

Generation Size	Generations
20	3901975
40	2652625
80	1358229
100	1346134
400	373987
800	130587
1000	126279
2000	40432
4000	40822
10000	17002

As can be seen in the figure above or in the chart to the left, the greatest jump in the required number of generations to calculate all the solutions is actually when the population size is doubled from 40 to 80, leading to a decrease of 130 thousand generations required. A similar decrease (100k less) can be seen when moving from a population size of 100 to 400 chess boards per generation. As previously stated, the optimal population size when factoring in both number of generations required and runtime falls around 800 to 1000 chess boards per generation.