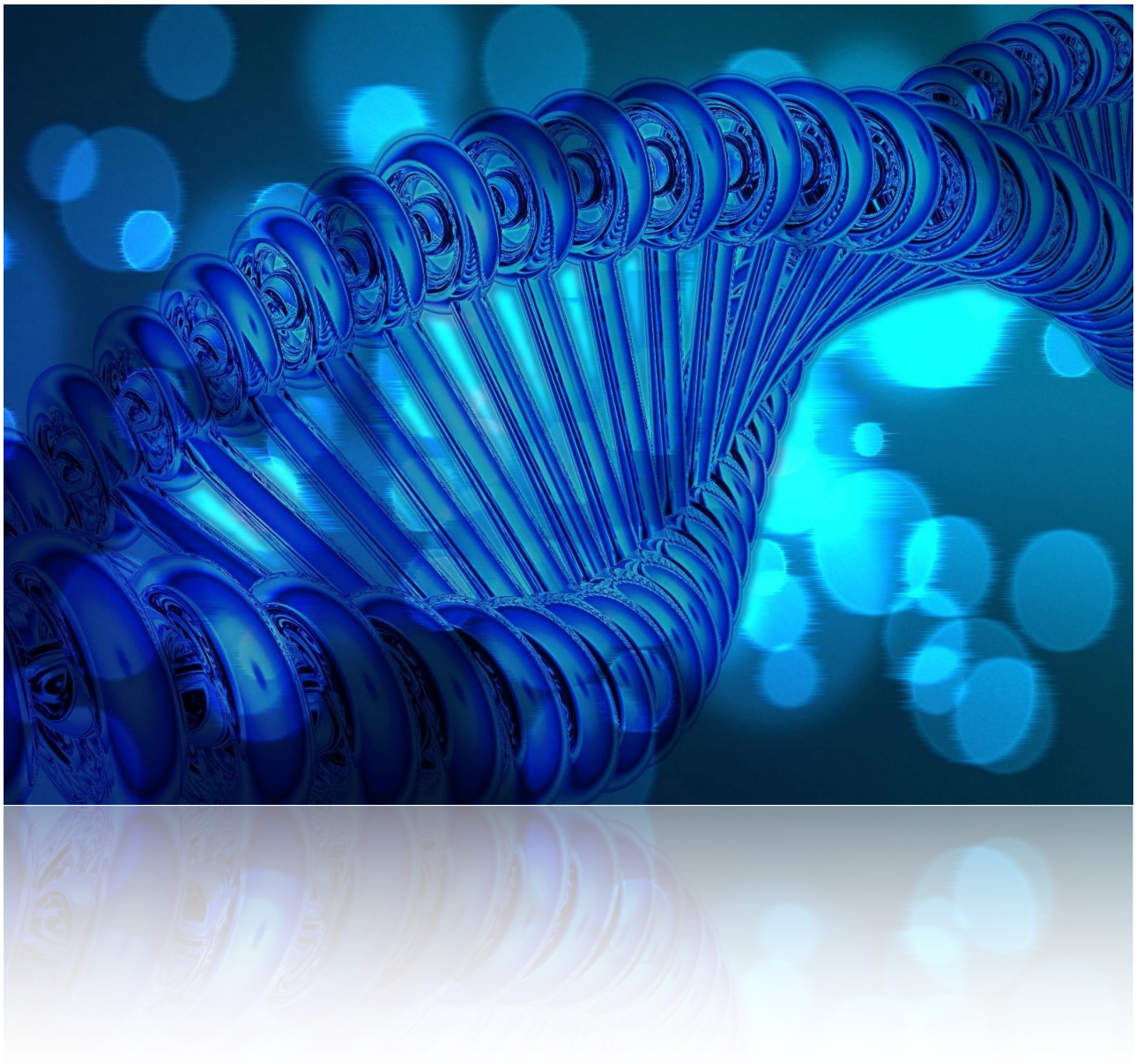
Assignment 2

COSC343 - Artificial Intelligence

Adam Sherlaw - 1935911



Genetic Algorithm : Design

Class : *NQueensGA*

A population of boards is generated, evaluated and then the population is used to produce offspring which are then returned into the population for future reproduction. The main method handles the setup for the GA by interpreting the command line arguments, setting the class variables and then executing the *geneticAlgorithm()* for the run count. *geneticAlgorithm()* from here on, handles the entire algorithm process.

Method : *geneticAlgorithm()*

The *geneticAlgorithm()* method handles the execution of all elements that are required for the algorithm to run successfully. The method first initialises the population by calling the *initPop()* method which handles the creation and evaluating of each board. The population is then sorted by cost, where the lowest cost boards are placed at the head of the population. The population is now in a position that is ready to produce new offspring.

A while loop is run until a solution is found (i.e. a board has a vector with cost 0). Inside this loop several operations occur; the *reproduce()* method is called to generate new offspring for the population; the *sort()* method is then called to sort the population once again so new offspring are positioned correctly for the next round of reproduction. Some counters are incremented and a conditional is in place to ensure we have not exceeded the maximum number of generations.

Method : *initPop()*

When executed, specifically at the beginning of the algorithm, *initPop()* handles the creation of the entire population. For every member of the new population, a new board is created with n-queen pieces placed at random positions on the board. The board is then evaluated to calculate it's cost which is used to determine the quality of the solution vector. The new board is then added to the population to be used in future operations.

Method : *reproduce()*

For all members of the population that fall outside of the parent population (at an index greater than the parent_size variable), we generate new offspring to take the place of these *old* members.
Reproduction is achieved by choosing two parents (using the selected parent selection type), crossing the two selected parents over with each other (using the selected crossover type) to create new offspring that are then mutated and finally the produced offspring are inserted into the population.
Parent crossover is broken into cases, where the opted selection type dictates how the parents will be crossed over. The 4 crossover types are; single fixed-point, single random-point, double fixed-point and double random-point crossover.

Single fixed-point, uses the *crossOver()* method with a single predefined point value for the crossover point. In our algorithm, we use the board size divided by two to give us a split size down the centre of the two parents.
Single random-point, uses the *crossOver()* method with a randomly generated number in the range of 0 to 7 exclusive as the crossover point. This range is to ensure that we get some crossover in the resulting vector, as if the split point was at 0, the entire resulting offspring is a copy of the second parent, and vide versa for split point of 7.
Double fixed-point crossover uses two predefined split values. In our implementation, we use point 2 and point 5. This gives us a reasonably equal split of the parents.
Double random-point crossover uses two randomly generated numbers. The two values are in the range of 0 to 7 exclusive, as stated before, this eliminates a direct copy of one of the parents. The smaller of the two values are used for the first split point and the larger is used for the second split point.

Method : *genParent()*

Cases are once again, used to choose which type of parent selection is used. The 4 parent-selection cases implemented are; Superior parent selection by binary tournament, random parent, alpha-male and parent pairs.
Superior parent selection uses binary tournament to select the better parent of the two randomly chosen parents from the parent population (parent with lowest cost).
Random parent selection uses a randomly generated number to give an index value of a parent in the parent population. This parent is then returned.
Alpha-male parent selection will mate the member of the parent population that has the best cost with all other members in the parent population.
Parent-pair selection pairs the members in the parent population together so that they only mate with their paired partner. e.g. Member 1 will mate with member 2, member 3 will mate with member 4 and so on.

Method : *Sort()*

Sort implements selection sort, which sorts the population based on their cost. The population is sorted so that costs are arranged into ascending order. This allows us to maintain the best solutions at the head (beginning) of the population.

Class : *Board*

Board class is used as a container to represent one board that contains a vector of queen positions. Boards are represented as arrays and the cost of a board is stored in a integer variable. The queen piece positions are represented in the array, with the index denoting the x axis and the value at that index representing the y axis value. Several functions are implemented in the board class such as; an evaluation of cost function, that is used to calculate the cost (fitness) of a board with regards to completeness of a solution, a cross board function, that is used to crossover two parents at given points, a mutate board function, that is used to select random positions in the board vector and assign them a new random queen. Some accessor and mutator functions are implemented to access the vector and cost data fields.

Method : *Board()*

This is the default constructor of the class, that initialises a new board to a vector containing queens at random locations.

Method : *evalBoard()*

Board vectors are given a cost based on the number of attacking queens. This is calculated by iterating over each queen in the vector array and checking if any other queens are on the same axis, be it horizontal or on either of the diagonal axes. If their is a collision, then the cost is incremented. This is done for all queens on the board.

By only having one queen per position in the vector, we can cancel out the need to check vertically for collisions. We also only check forward of the queen in the vector array as if we were to check both forwards and backwards, then we would count collisions multiple times. The final cost count is set to the cost of the board.

Method : *crossBoard()*

Cross board takes three parameters; board to crossover with the current board, and two integers; cross point 01, cross point 02. These two points denote the first and second split points.

To cross over the two boards, we first take the current board's vector from 0 to the first split point (-1) and assign it to the child's vector. We then take the input boards vector from the first split point to the second split point (-1) and add the selection to the child's vector. We finally take the current board's vector from the second split point to the end of the vector and add it to the child's vector. we finish by returning the child vector.

Method : mutateBoard()

For all positions in the vector; Generate a random number between 0 and 1 inclusive and compare it to the mutation rate. If the number is less than the mutation rate, generate a random queen value. Otherwise, move onto the next position.

Genetic Algorithm : Experimentation

In order to allow our genetic algorithm to run at its optimum, the values and the methods of selection that are used need to be experimented with to allow us to give it the best possible chance of efficiently finding a solution to the problem.
The values that I shall be experimenting with here are:

Population size: The size of the population in which to run our Genetic Algorithm on.
Number of parents: The size of the population that is dedicated to the generation of new offspring.
Mutation Rate: The probability of a random queen in a vector being replaced by a new randomly placed queen.
Number of generations: The maximum number of generations allowed to be run
Parent selection type: The different types of parent selection
Crossover type: The different types of crossover

To observe the effect that these variables have on the number of generations required to generate a complete solution, I shall run the Genetic algorithm for 40 runs and take the average epoch count as the result of evaluating each test. I shall also keep track of the number of runs that reach the epoch count as a measure also.

Parent and Population Size:

The table (A) below shows the average number of epochs generated for different population and parent combinations. As the population is decreased, we can observe an increase in number of generations that are required to generate a solution. This increase in number of generations required can be seen most clearly in the number of times that the epoch count limit is reached.
As the limit was set at 5000 for our testing, in the larger population (1000), this limit was was generally large enough to allow the algorithm enough epochs to return a solution, as shown by an average of 7 out of 30 runs encountered the epoch limit. It is to be expected that we can sometimes hit a local minimum and not be able to break away from under it, and therefore the epoch count would be reached.
This can be compared to the smaller population of 10, which on average, had 34 out of 40 runs that encountered the epoch limit. This is a much larger percentage of runs exceeding the epoch limit. If we were to continue using this population size, we would increase the number of epochs that the algorithm is allowed to run to offer the algorithm a better chance to find a solution.

The size of the parent population also has a great influence on the number of epochs required to calculate a solution. A parent size of around 50 percent of the total population has a far lower number of epochs which is in some cases, cab be more than half of the epochs that a parent size of around 10 percent generates.

This means that we need to keep our parent population around 50 percent to have a good chance at a more desirable offspring. This would make sense, as the more variance we have in out breeding population, the greater chance we have of missing the local minimum and heading for the global minimum.

Pop. Size	No. Parents	Average Epochs	EPOCH Limit
1000	100	200	15
	200	160	4
	500	52	3
500	50	125	13
	100	321	12
	250	231	9
100	10	213	31
	20	160	28
	50	43	30
50	6	12	35
	12	61	34
	24	86	33
10	2	293	34
	4	212	33
	6	143	36

(A)Table showing population and parent size in relation to the number of generations required to calculate a solution. (Double Random-point crossover, Pair-Parent selection, Mutation value of 0.001, epoch limit of 5000 used)

Mutation Rate:
The mutation rate is important to ensure that the population retains the ability to continue to diverge away from the local minimum and not a local minimum. The mutation rate needs to be set to a value that allows the population to be sufficiently able to produce offspring that pull the population out of a local minimum.
The table (B) below shows different mutation rates and the effect that they have on the number of epochs required to generate a solution.

Mutation Rate	Epocs	EPOCH Limit
0.0001	158	24
0.001	91	17
0.01	253	16
0.1	287	16
0.5	69	0
0.8	492	0
1	712	0

(B) Table showing mutation rate in relation to the number of generations required to generate solution. (Single Random-point crossover, Random parent selection, epoch limit of 5000 used)

A very small mutation value of very nearly zero requires the population to rely on the crossover of parents to not put them into a local minimum. The converging effect of crossover is greater than the smaller mutation rate so within several generations the effect will be unnoticeable. On the other hand, if the mutation rate is so large, the population will be made up of nearly all randomly generated members. This will allow us to find the global minimum, eventually, but too much mutation can lead to missing the global minimum as the algorithm will never get a chance to progress towards a solution. This can be seen when the mutation rate is set to a value near 1. The number of epochs required to generate a solution is greater than if the mutation rate was set to 0. The advantage of having this large mutation value is that you will eventually find the local minimum and never become stuck in a local minimum.
Here, a value of 0.5 would work well as the number of epochs required is low and the ability to jump out of local maxima is of good benefit.

Number of Generations:
The number of generations that the algorithm is allowed to run is simply a measure that in the event of becoming trapped in a local minimum, will stop the algorithm from trying forever. The number of generations needs to be set such that the algorithm has enough room to find a solution. For experimenting, due to time constraints, I have limited my algorithm to 5000 generations.

Parent Selection and Crossover Type:
Choosing a suitable parent is a critical part of any Genetic algorithm. If we choose parents that produce off spring that move us away from a possible solution, then we have a problem. Another issue that has to be addressed is the crossing over of two parents. We must cross parents over in such manner that the resulting offspring are unique to their parents so that we do not end up with a population filled with identical boards.

Below is a table (C) displaying the epochs that result from the different combinations of parent selection and crossover positioning.

	Superior	Limit	Random	Limit	Alpha	Limit	Pairs	Limit
Single Point Fixed	158	20	168	15	155	24	68	32
Single Point Random	51	3	**14**	0	136	17	21	18
Double Point Fixed	213	16	301	19	55	27	61	28
Double Point Random	142	6	209	1	206	18	314	17

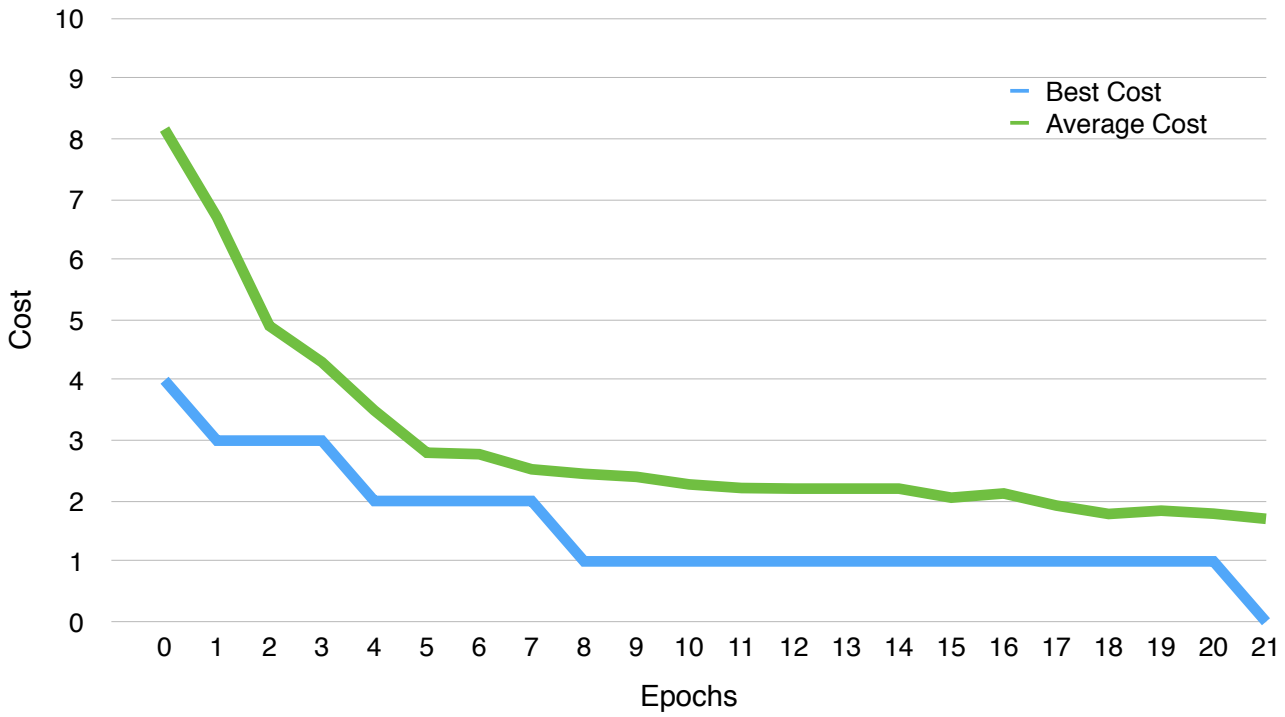
(C) Table showing the resulting epochs from parent selection and crossover point combinations. (Population size 500, Parent Size 240, Mutation value of 0.001, epoch limit of 5000 used)

Using single point-fixed position crossover point gave fairly even results for all types of parent selection, with pairs being slightly less effective than the rest.
Single point-random crossover gave the most effective combination when teamed with the randomly chosen parent, and gave most of the other parent selections an increase in their effectiveness.
Double fixed-point crossover was the least effective of all, giving all of the epochs in some cases, double the epoch count. On the other hand, the double point random gave the second best epoch counts for all parent selections.

Based on the results in the table above, choosing a random single-point for crossover and random parents from the parent population, is the most effective way to generate a solution to the n-queens problem.

Genetic Algorithm : Results

The graph below shows the best cost and average cost for each generation.



Genetic Algorithm Parameters:

Population size:	200
Number of parents:	100
Mutation Rate:	0.01
Number of generations:	10000
Parent selection type:	Random parent
Crossover type:	Single random-point

The two lines show a strong correlation to each other, where the average cost and best cost decrease and converge at a comparable rate. This shows us that our reproduction between parents is working effectively where the mutation rate keeps us out of local minimum and the offspring produced are always progressing towards a solution.

The parameters for the algorithm have been adapted from "*optimal*" to show a gradual change in the costs whilst converging to a global minimum solution. A larger population could have been used for a more optimal solution.

```
import java.util.ArrayList;
import java.util.Random;


public class NQueensGA {


    private static int POP_SIZE;    // Size of the population to be used
    private static int BOARD_SIZE;  // Size of the board i.e. number of queens
    private static double MUTATION_RATE;
    private static int NO_PARENT;   // Number of offspring in each generation
    private static int NO_GEN; // Maximum number of gens to be run

    private static int RUN_COUNT;   // Number of times to run algorithm
    private static int PARENT_SEL;  // Which parent selection method to use
    private static int CROSS_OVER;  // Which parent crossover method to use

    public static boolean complete = false; // True when solution is found
    public static int parIndex;             // Index used for parent selection
    private static int EPOCHS;              // Number of generations elapsed

    private static int avEpoch = 0;
    private static int avCost = 0;
    private static int max = 0;

    private static ArrayList<Board> population = null; // Population of boards


    /*  Main method, handles command line inputs for setting class variables
     *  and invoking the genetic algorithm.
     */
    public static void main(String[] args) {
        // Usage BOARD_SIZE, POP_SIZE, NO_PARENTS, NO_GENERATIONS
        // MUTATION_RATE, RUN_COUNT, PARENT_SELECT, CROSSOVER
            try {
                BOARD_SIZE      = Integer.parseInt(args[0]);
                POP_SIZE        = Integer.parseInt(args[1]);
                NO_PARENT       = Integer.parseInt(args[2]);
                NO_GEN          = Integer.parseInt(args[3]);
                MUTATION_RATE   = Double.parseDouble(args[4]);
                RUN_COUNT       = Integer.parseInt(args[5]);
                PARENT_SEL      = Integer.parseInt(args[6]);
                CROSS_OVER      = Integer.parseInt(args[7]);


            } catch (NumberFormatException e) {
                System.err.println("Argument " +
                    args[0]+ " must be an integer or double for mutation rate");
                System.exit(1);
            } catch (ArrayIndexOutOfBoundsException e) {
                System.err.println("Usage: \nBOARD_SIZE, POP_SIZE, NO_PARENTS,
                    NO_GENERATIONS, \nMUTATION_RATE, RUN_COUNT, PARENT_SELECT,
                                    CROSSOVER");
                System.exit(1);
            }
```

```
        /*System.out.print("\n\nBoard Size : " + BOARD_SIZE + " : ");
        System.out.print("Population Size : " + POP_SIZE + " : ");
        System.out.print("Number Parent : " + NO_PARENT + " : ");
        System.out.print("\nMax Num Generation : " + NO_GEN + " : ");
        System.out.print("Mutation Rate : " + MUTATION_RATE + " : ");
        System.out.print("Run Count : " + RUN_COUNT + " \n");
        System.out.print("Parent Type : " + PARENT_SEL + " : ");
        System.out.print("Cross Type : " + CROSS_OVER + " \n\n");
        */

    // Run algorithm for RUN_COUNT iterations
    for (int i = 0; i < RUN_COUNT; i++) {
        EPOCHS = 0;
        complete = false;
        population = new ArrayList<Board>();
        geneticAlgorithm();
        //System.out.println("Pop:\t" + (i+1) + "\t\t Count:\t" + EPOCHS);
        if (EPOCHS != NO_GEN) avEpoch += EPOCHS;
        else max++;

    }
    System.out.println("Average Epochs: " + avEpoch / RUN_COUNT);
    System.out.println("Max: " + max);
}

/*
 *  Genetic Algorithm begins here, initialisation, sorting and looping
 *  take place here along with checking the stopping condition.
 */
public static void geneticAlgorithm() {

    initPop();  // Initalise the population then evaluate all members.
    sort();     // Sort the population based on each members cost.
    System.out.println(population.get(0).cost());
    System.out.println(avCost/POP_SIZE);
    // Until a solution (member with cost = 0) is found, do:
    while (!complete) {
        avCost = 0;
        reproduce();    // Take two parents and create two offspring
        sort();         // Sort the population based on each members cost
        //System.out.println(population.get(0).cost());
        //System.out.println(avCost/POP_SIZE);
        if (EPOCHS >= NO_GEN) break; // If at generation limit.
        else EPOCHS++;
    }
}


/*  Initalise the Population - For each member of the population, create a
 *  new Board with random queen pieces. Then evaluate the board to find
 *  the fitness of the board. Then add the board to the population.
 */
static void initPop() {
    Board newBoard = null;
    for (int g = 0; g < POP_SIZE; g++) {
        newBoard = new Board();
```

```
            newBoard.evalBoard();
            population.add(newBoard);
        }
    }

    /*  Reproduce population - Take the current population and select parents
     *  to mate and create new offspring. This is done by crossing over two
     *  selected parents (parents have the better (smaller) cost) and the
     *  result is two child offspring which are then put into the population.
     *
     *  Case 0: Single fixed-point crossover
     *  Case 1: Single random-point crossover
     *  Case 2: Double fixed-points crossover
     *  Case 3: Double random-points crossover
     */
    static void reproduce() {
        int p1 = 0,  p2 = 0;
        Random rn = new Random();
        parIndex = 1;

        /* For all of the non-parent members of the population, replace with
         * offspring of parent breeding
         */
        for (int i = NO_PARENT; i <= POP_SIZE - 1; i += 2) {
            Board ch1 = new Board();
            Board ch2 = new Board();

            Board par1, par2 = null;

            // Support for parent selections - (2)Alpha-male and (3)pairs
            if (PARENT_SEL == 2) {
                par1 = population.get(0);
                par2 = genParent();
            } if (PARENT_SEL == 3) {
                par1 = genParent();
                par2 = population.get(population.indexOf(par1) + 1);
            } else {
                par1 = genParent();
                par2 = genParent();
            }

            // Ensure two parents are not same (at same index)
            while (population.indexOf(par1) == population.indexOf(par2)) {
                par1 = population.get(rn.nextInt(NO_PARENT + 1));
            }

            // Switch for selecting crossover
            switch(CROSS_OVER) {
                case 0:
                        // Single point, fixed position
                        ch1.vector(par2.crossBoard(par1, 0, BOARD_SIZE/2));
                        ch2.vector(par1.crossBoard(par2, 0, BOARD_SIZE/2));
                        break;
                case 1:
                        // Single point, random position
                        p1 = 1 + rn.nextInt(BOARD_SIZE - 1);
```

```
                    ch1.vector(par2.crossBoard(par1, 0, p1));
                    ch2.vector(par1.crossBoard(par2, 0, p1));
                    break;
            case 2:
                    // Multiple point, fixed position
                    ch1.vector(par2.crossBoard(par1, 2, 5));
                    ch2.vector(par1.crossBoard(par2, BOARD_SIZE/3,
                                               BOARD_SIZE/3 * 2));
                    break;
            case 3:
                    // Multiple point, random position
                    p1 = 1 + rn.nextInt(BOARD_SIZE - 1);
                    p2 = 1 + rn.nextInt(BOARD_SIZE - 1);
                    int min = Math.min(p1, p2), max = Math.max(p1, p2);

                    ch1.vector(par1.crossBoard(par2, min, max));
                    ch2.vector(par2.crossBoard(par1, min, max));
                    break;
        }

        // Mutate the two children
        ch1.mutateBoard();
        ch2.mutateBoard();

        // Evaluate the cost of the two children
        complete = ch1.evalBoard();
        complete = ch2.evalBoard();

        // Insert the two children ito the population
        population.set(i, ch1);
        population.set(i + 1, ch2);
    }
}

/*  Generate a parent to be used for creating new offspring.
 *  Depending on which case is selected, a parent is chosen from the
 *  parent population and is returned.
 *
 *  @return Board - The selected parent
 *
 *  Case 0: Superior parent selcetion - binary tournment (Best of two
 *  parents)
 *  Case 1: Random member from the parent population is returned
 *  Case 2: Alpha-male, the best parent is mated with all other parents
 *  Case 3: Mate pairs of population members - 0 with 1, 2 with 3, etc
 */
static Board genParent() {
    Random rn = new Random();

    switch (PARENT_SEL) {
        case 0:
                // Generate two random numbers in the parent population
                int r1 = rn.nextInt(NO_PARENT + 1);
                int r2 = rn.nextInt(NO_PARENT + 1);

                // Compare the fitness values of the two random boards
```

```
                int f1 = population.get(r1).cost();
                int f2 = population.get(r2).cost();

                // Return the better of the two parents
                return f1 < f2 ? population.get(r1) : population.get(r2);
        case 1:
                // Return a random parent from the parent population
                return population.get(rn.nextInt(NO_PARENT + 1));
        case 2:
                // Return the next parent to be mated with the alpha-male
                if (parIndex == NO_PARENT + 1) parIndex = 1;
                return population.get(parIndex++);
        case 3:
                // Return the first
                if (parIndex == NO_PARENT + 1) parIndex = 1;
                parIndex += 2;
                return population.get(parIndex - 3);

        default: return population.get(rn.nextInt(NO_PARENT + 1));

    }
}


/*
 *  Sort the population based on the cost of each member.
 *  Sorted from lowest cost to highest cost, as the lower the cost
 *  the better the member (solution) is.
 */
static void sort() {
    // Selection sort is used to sort the members in the population
    for (int i = 0; i < POP_SIZE; i++) {
        int first = i;
        for (int j = i + 1; j < POP_SIZE; j++) {
            if (population.get(j).cost() < population.get(first).cost()) {
                first = j;
            }
        }
        Board temp = population.get(first);
        population.set(first, population.get(i));
        population.set(i, temp);
    }
return;
}


/*
 *  Print out the population of members.
 *  Iteratively goes through all members vectors and iterates over all
 *  Queen pieces to print out all of their values.
 */
static void print() {

    for (int i = 0; i < POP_SIZE; i++) {
        Board temp = population.get(i);
        System.out.print("Board : " + i + "\t [ ");
```

```
        for (int j = 0; j < BOARD_SIZE; j++) {
            System.out.print(temp.vector()[j] + ", ");             }
            System.out.println("\t Cost : " + population.get(i).cost()
                                                    + "  ");         }
            System.out.println("\n\n");     }



/*
 *  Board class is used to hold data and methods that relate to the board
 *  (set of queens in vector).
 */
private static class Board {

    private int vector[] = new int[BOARD_SIZE];//Vector for queen positions
    private int cost = 0;                       // Cost of the vector


    /*
     *  Initaise the board (vector) with random queen positions with
     *  values between 1 and 8.
     */
    public Board() {
        for (int i = 0; i < BOARD_SIZE; i++) {
            int min = 1, max = 8;
            this.vector[i] = min + (int)(Math.random() * ((max-min) + 1));
        }
    }

    /*
     *  Evaluates the board (vector) to calculate the fitness of the
     *  board vector.
     *  Cost is calculated by checking for colisions on the horizontal
     *  axis, and both the upper and lower diagonal axes for each posioin
     *  in the vector.
     *  Checking out of bounds in the y axis is not implemented, as their
     *  is no risk of indexing outof bounds as we are only comparing two
     *  integer values. This simplifies code.
     */
    public boolean evalBoard() {
        cost = 0;

        // For each position in the vector, check for collisions
        for (int index = 0; index < BOARD_SIZE; index++) {
            int cur = vector[index], dia = 1;

            for (int i = index + 1; i < BOARD_SIZE; i++) {
                if (cur == vector[i]) cost++;       // Horizontal
                if (cur + dia == vector[i]) cost++; // Up diagonal
                if (cur - dia == vector[i]) cost++; // Down daigonal
                dia++;
            }
        }
            avCost += cost;
            return cost == 0 ? true : complete;
```

```
}

/*
 *  Crossover two parents at given points to generate a new offspring.
 *  NOTE: Call method twice (with parents inverted) to get two
 *  offspring.
 *  @param Board - Parent Board to be crossed with current Board.
 *  @param int - Cross point one - First point to crossover
 *  @param int - Cross point two - Second point to crossover.
 */
public int[] crossBoard(Board board, int crossP1, int crossP2) {
    Board child = new Board();

    // Take parent1 (current board) up to point 1
    for (int i = 0; i < crossP1; i++) {
        child.vector[i] = this.vector[i];
    }
    // Take parent 2 from point 1 to point 2
    for (int j = crossP1; j < crossP2; j++) {
        child.vector[j] = board.vector[j];
    }
    // Take parent 1 (current board) from point 2 to end
    for (int j = crossP2; j < BOARD_SIZE; j++) {
        child.vector[j] = this.vector[j];
    }
    return child.vector();
}

/*
 *  Randomly takes a bit from the vector and chooses a new random value
 *  to put in its place.
 *  Random probability is denoted by the mutation rate
 */
public void mutateBoard() {

    for (int j = 0; j < BOARD_SIZE; j++) {
        if (Math.random() < MUTATION_RATE) {
            this.vector[j] = 1 + (int)(Math.random() * (8));
        }
    }
}

/*
 *  Method to set the vector of a board
 */
public void vector(int[] vector) {
    this.vector = vector;
}

/*
 *  Method to retrieve the vector of a board
 */
public int[] vector() {
    return this.vector;
}
```

```
        /*
         *  Method to retrieve the cost of a board
         */
        public int cost() {
            return cost;
        }
    }
}
```