Computational Intelligence

Assessment 2

Nicholas Randles – B00058026

Task 1 – Changes to source code

ValueEncodedChromosome.java

Constructor

In the constructor, the genes array is initialised with the length value passed through one of the parameters. Also, the genes are assigned random values between 1 and the upper bound.

```
genes = new int[length];
// Initialises the genes to random values between 1 - upper_bound, 30 in this case
for(int i = 0; i < genes.length; i++){
    int random = (int) (Math.random() * ((upper_bound - 1) + 1));
    genes[i] = random;
}
```

getGeneAt method

Returns a gene at a certain position in the array.

```
public int getGeneAt(int pos){
    //TODO
    return genes[pos];
}
```

setGeneAt method

Sets the value for a gene array variable at a selected position with a selected value.

```
public void setGeneAt(int pos, int val){
    //TODO
    genes[pos] = val;
}
```

mutateGeneAt method

Generates a random value and assigns it to a gene array variable at a selected position.

```
public void mutateGeneAt(int pos){
    //TODO
    int random = (int) (Math.random() * ((upper_bound - 1) + 1));
    genes[pos] = random;
}
```

setLength method

Sets the length of a chromosome with the inputted value.

```
public void setLength(int length){
    //TODO
    this.length = length;
}
```

getLength method

Returns the length of a chromosome.

```
public int getLength(){
    //TODO
    return length;
}
```

setFitness method

Sets the fitness of a chromosome to the input value.

```
public void setFitness(int fitness){
    //TODO
    this.fitness = fitness;
}
```

equals method

Takes in a chromosome object and compares its genes to its own. If they match it will return true, otherwise it will return false.

```
public boolean equals(ValueEncodedChromosome c){
    //TODO
    for(int i = 0; i < c.length; i++){
        if(genes[i] == c.genes[i]){
            return true;
        }
    }
    return false;
}
```

toString method

Joins all of the genes into one string and returns it.

```
public String toString(){
    //TODO
    String values = "";
    for(int i = 0; i < genes.length; i++){
        values += genes[i] + " ";
    }
    return values;
}
```

Population.java

Fitness method

Takes in a set of values and sums them together. We then subtract it by 30 to see how big the difference is. It uses the Math.abs method to avoid negative numbers. It then returns the difference.

```
public int fitness(int a, int b, int c, int d){
    // TODO: Calculate the difference between chromosome solution and required equivalance of 30
    int solution = Math.abs((a + (2 * b) + (3 * c) + (4 * d)) - 30);
    return solution;
}
```

Converged method

Checks if the top chromosome's fitness is zero and if it is, it will return true, otherwise it will return false.

```
public boolean converged(){
    //TODO: return boolean value based on whether we have found a solution (True = converged, False = not converged)
    if(population[0].getFitness() == 0){
        return true;
    }
    return false;
}
```

Evaluate method

Loops through all of the chromosomes and assigns its genes to variables. Then it passes the variables to the fitness method and uses it to set the fitness of the chromosomes.

```
public void evaluate(){
    //TODO: Loop through the population and set each chromosomes fitness based on its values a,b,c,d
    for(int i = 0; i < population_size; i++){
        int a = population[i].getGeneAt(0);
        int b = population[i].getGeneAt(1);
        int c = population[i].getGeneAt(2);
        int d = population[i].getGeneAt(3);
        population[i].setFitness(fitness(a,b,c,d));
    }
}
```

Evolve method

Gives us a method to complete a single evolution as it contains all of the methods necessary for evolution.

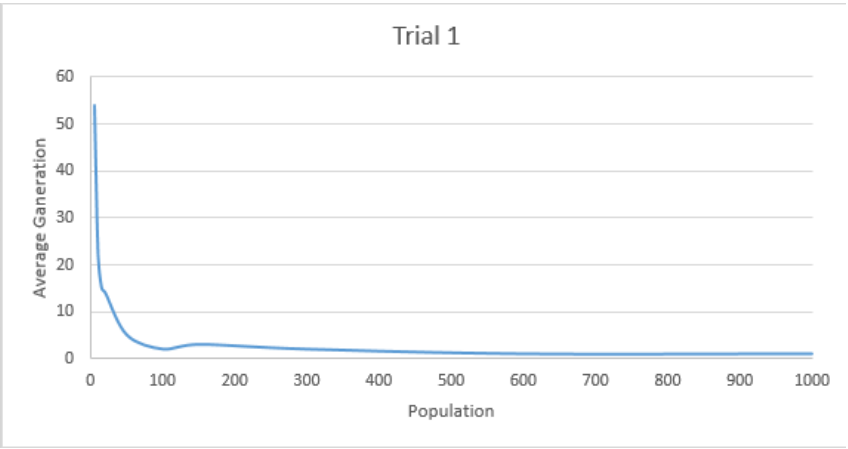
```
public void evolve(boolean display)
{
    //TODO: Evaluate, sort, display, crossover, mutate, remove duplicates
    // Only call display if display == true
    evaluate();
    sort();
    crossover();
    mutate();
    removeDuplicates();
    if(display == true){
        display();
    }
}
```

Task 2 – Experiments

Trial 1

For the first trial the crossover fraction is set to 0.5 and the mutation fraction is set to 0.2. The population starts of at 5 and finishes at 1000. As the population gets larger the average generations start to decrease. I believe this is because there is only a four genes used in this problem and the population is more likely to be closer to the goal as it contains more chromosomes.

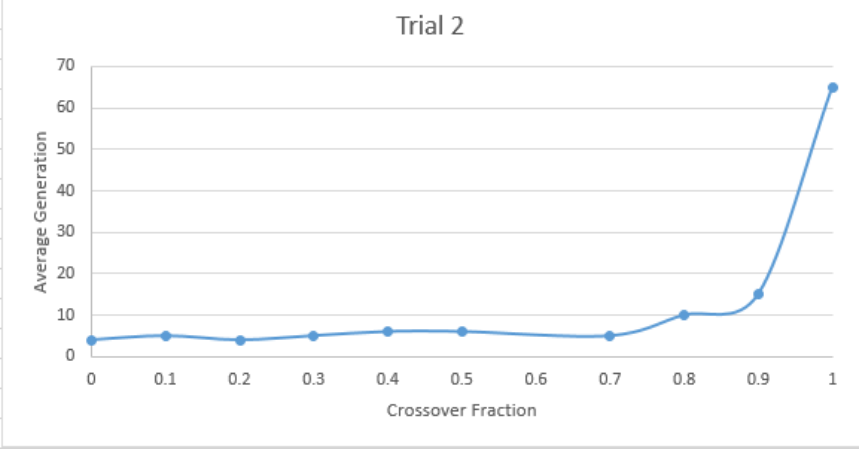
Trial 1	
Population: 5-1000	
Crossover: 0.5	
Mutation: 0.2	
Population	Average Generation
5	54
10	22
15	15
20	14
50	5
100	2
150	3
300	2
600	1
1000	1



Trial 2

In the second trial the population is set to 50 and the mutation fraction is set to 0.2. The crossover fraction ranges from 0 to 1. The average generations start off low because we are using crossover. As the crossover fraction increases the more average generations it takes to solve the problem. I believe this is because we are using less crossover. When we don't use crossover the population is only evolving through mutating one gene at a time, which will take longer.

Trial 2	
Population: 50	
Crossover: 0-1	
Mutation: 0.2	
Crossover Fraction	Average Generation
0	4
0.1	5
0.2	4
0.3	5
0.4	6
0.5	6
0.7	5
0.8	10
0.9	15
1	65



Trial 3

In the third trial the population is 50 and the crossover fraction is 0.5. The mutation fraction ranges from 0 to 3. As the mutation fraction increases so does the average generations. The job of mutation is to change one gene in a chromosome randomly. This is normally a good thing as it increases diversity and helps prevent local extremes. Although too much mutation can be bad, as it can cause too much randomization. I believe that the average generations are increasing as there is too much randomization occurring to the population.

Trial 3		
Population: 50		
Crossover: 0.5		
Mutation: 0-3		
Mutation Fraction	Average Generation	
0	4	
0.1	7	
0.2	7	
0.3	4	
0.4	6	
0.5	7	
0.7	6	
1	6	
2	8	
3	12