# Evolution of Complexity: Assignment 2

Jordan Milton
jksm1g10

January 7, 2014

## 1 Introduction

### 1.1 Description

One way in which the fitness of a genome in a Genetic Algorythm can be determined is by evaluating it against other genomes, which can provide several benefits over using an objective metric to measure performance. These benefits include:

**Providing a hittable target:** This is when evaluating the fitness of a genome by comparing it to another gives a better gradient for improvement over an objective metric. For instance, if a game playing machine was being evolved it would be more meaningful to compare it to other players of around the same ability than having them compete against a perfect player. This is because the latter makes it difficult to distinguish which genome is superior, where as in the former scenario this should become clear.

**Providing a relevant target:** In some cases, in can be difficult to create a static test of fitness that makes the genome optimise all dimensions involved. To return to the game player example, it is possible to have the fitness be decided by the number of victorys against a predetermined set of other players. However it may be possible to beat these players without having optimised all dimensions. By having the opponent selected be another evolving genome the adaptive competition should focus the adaptation on aspects not yet optimised.

**No defined end goal:** Another advantage of coevolution is that as the target moves with the fitness of the genome, there is nearly always room for improvement. To stick with the game player idea, with a fixed set of opponents once the genome outperforms all opponents it no longer has a target. However, when competing against other improving genomes there is always the possibility of becoming a better player than the best player found so far. Once found, this player is the new target.

However, it is also true that coevolution can introduce problems that would otherwise not exist. The problems introduced have a close relation to the problems it purportedly solves. The first issue relates to scenarios where in the populations seperate enough so that they no longer provide a good gradient for learning. The second issue relates to how focussing can create genomes that focus on the wrong things. Thus a generalist is never developed. The last issue is due to the loss of a objective fitness, and so it is possible that players do not improve in any objective sense. The paper that has been reimplemented provides a description of these, and then describes a minimal substrate for demonstrating those issues[3]. This minimal substrate consists of multiple number games played by competing genomes. This has two significant advantages. The first is that by having the fitness be determined by simple number games the algorythm is reasonably simple to implement. The second main reason is that it allows the objective fitness of genomes to be easily observed, even though it isn't used in the algorythm itself. This is often not the case in a real world application, which can make these problems difficult to spot.

### 1.2 Experimental Setup

In order to demonstrate the ideas discussed above, three different fitness functions need to be defined, with two different genome definitions. For the first game the genome will be defined as a binary string 100 bits long. The value of the genome will be determined as the number of 1s in the string. The fitness function is defined as:

$$fitness1(a, S) = \sum_{i=1}^{|S|} score1(a, S_i) \tag{1}$$

$$score1(a, b) = \begin{cases} 1 & \text{if } a > b \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

For the second game the genomes will have 10 dimensions opposed to the previous 1. However, the size of each dimension will be 10 bits. This keeps means that the sum of all dimensions still ranges from $0 - 100$. The new fitness function will be defined as above, but only one dimension will be chosed for comparison. This will be decided by choosing the dimension in which the two genomes are most different. Below is a mathematical definition for the game for if it was in two dimensions instead of ten:

$$fitness2(a, S) = \sum_{i=1}^{|S|} score2(a, S_i) \tag{3}$$

$$score2(a, b) = \begin{cases} score1(a_x, b_x) & \text{if } |a_x - b_x| > |a_y - b_y| \\ score1(a_y, b_y) & \text{otherwise} \end{cases} \tag{4}$$

For the last game, the only change is in the fitness function. Instead of picking the dimension the two genomes differ most, instead the dimension where they differ least is used for comparison. Below is the mathematical definition of the game in two dimensions instead of ten:

$$fitness3(a, S) = \sum_{i=1}^{|S|} score3(a, S_i) \tag{5}$$

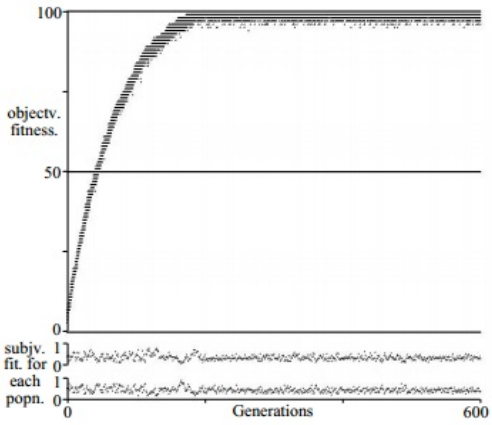
$$score3(a, b) = \begin{cases} score1(a_x, b_x) & \text{if } |a_x - b_x| < |a_y - b_y| \\ score1(a_y, b_y) & \text{otherwise} \end{cases} \tag{6}$$

The genetic algorythm will use a generational approach, and fitness proportionate selection will be used for choosing parents. For simplicity, the only variational operator will be mutation. For any test run with two populations, the sample to test against will be taken entirely from the other population. A table of all parameters is given below, assume these values are used unless otherwise stated.

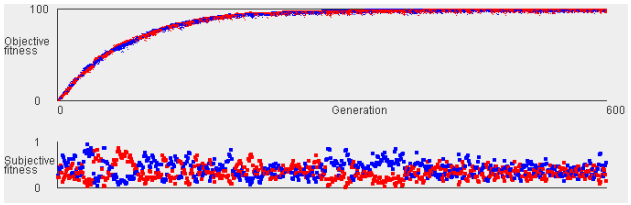
| Variable | Value |
|---|---|
| Mutation Rate | 0.005 |
| Population Size | 25 |
| Sample Size | 15 |
| Number of Generations | 600 |
| Number of populations | 2 |

## 2 Reimplemented results

This section goes through the experiments from the original paper alongside the results from the reimplementation.

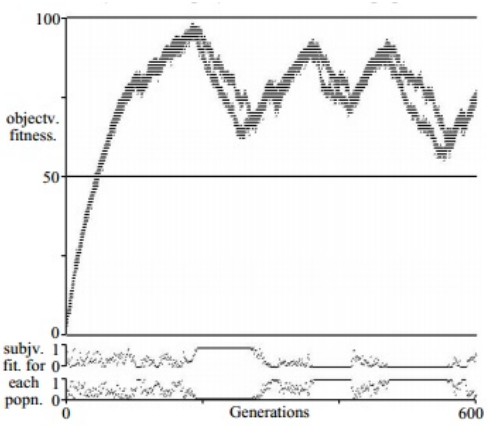



(a) Original

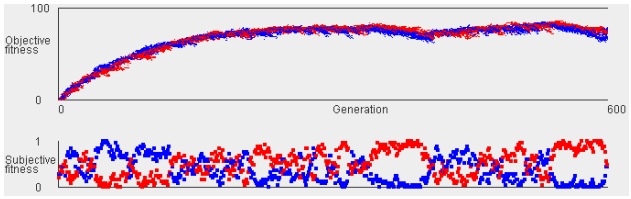



(b) Reimplementation

Figure 1: Coevolution using the first game




(a) Original



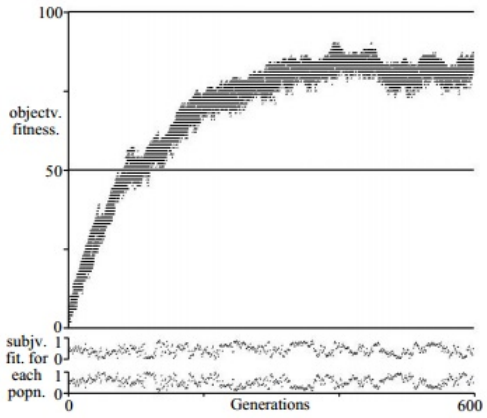

(b) Reimplementation

Figure 2: Coevolution using the first game and sample size 1

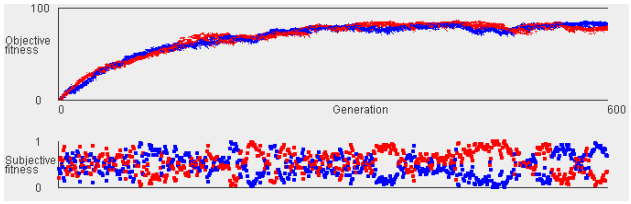
Figure 1 shows the results from running the GA using the first fitness function. Red and blue in the reimplementation represent the two different populations. While this does reach the optimum state (objective fitness equals 100), the subjective fitness graph looks no different by generation 600 than it does at generation 300. What this shows is that the seperation of objective fitness from subjective fitness can make it difficult to know how well a coevolved population is progressing.

Next, the algorythm is run again. However this time the sample size is set to 1, so that each player plays against one genome from the other population. The result is shown in figure 2. In both the original and the reimplemented result you can see that the algorythm now behaves completely differently. Looking at the subjective fitness, it can be seen that there are large peroids where the average fitness for either population is either 1 or 0. This means that genomes can't be differentiated and any selective pressure is lost. This leads to the ocassional phenomenon where the populations actually decrease in objective fitness, as without a selective pressure they drift until they reconnect with the other population. This effect is more obvious in the original but is still visible in the reimplementation.

The next test involves the second game and fitness function. The results of running this game can be seen in figure 3. For the purpose of showing output, the objective fitness is the sum of all dimensions. This looks

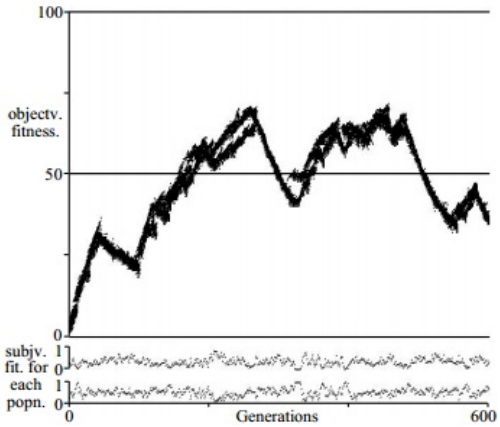



(a) Original

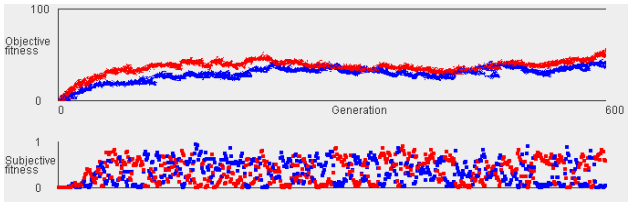



(b) Reimplementation

Figure 3: Coevolution using the second game




(a) Original




(b) Reimplementation

Figure 4: Coevolution using the third game

very similar to figure 1, however unlike that experiment it fails to reach the optimal point at 100. While the selection pressure does seem to be changing what dimension is important, all other dimensions drift towards the mutation bias as described in the previous paper[3]. This means that it never produces a generalist.

The last experiment uses the third game. This test demonstrates the concept of relatavism where the selection pressure provided by coevolution fails to drive the population towards an objectively better genome. The results can be seen in figure 4. Two big differences from figure 1 are immediately visible. The first is that neither population ever approaches the objective optimal point, and indeed sometimes actively moves away from it. The second is that the populations occassionally clearly split objectively, however not subjectively. The reason for this can be seen by inspecting the fitness function. Due to the nature of the function, it can occassionally be beneficial to reduce the value of a dimension. This can be either to increase the gap between a low scoring dimension and the genomes opponents, or to reduce a high scoring dimension to make it more relevant. This can occassionally lead to the objective fitness being driven down. The split in populations occurs because it is possible to win frequently with many very low scoring dimensions, indeed it is sometimes easier to do so.

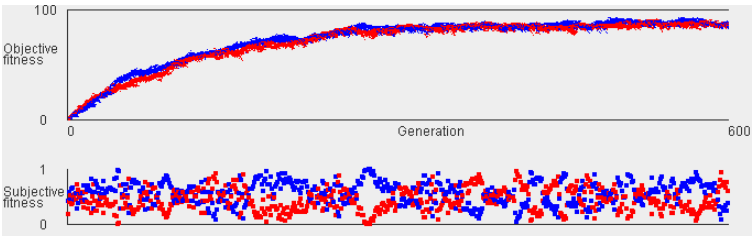



Figure 5: Game two with crossover

# 3 Extension

## 3.1 Hypothesis

While evaluating the results of the second game, where the dimension of a genome being optimised changes causing other dimensions to drift back to the bias, it seemed like it might have the makings of being a building block problem[2, 1]. The hypothesis then is that performing crossover in a population playing the second game will overcome the focussing problem.

The basis for this hypothesis is that while each high scoring genome has only necessitated optimising one dimension, it seems likely that different high scoring genomes will have scored well based on different dimensions. In that case, performing crossover should create a more generalist candidate. Over time this will hopefully lead to candidates that excell in all dimensions.

While this might initially seem like an attempt at a specific solution to this one game, the game is designed to be a simple example of a problem that exists in multiple real world applications. If making this change can create more generalists in this example, it is possible that a similar course of action could improve other systems that also suffer from focussing.

In order to carry out this test it is necessary to make a change to how children are created in the second game. Before mutation occurs, it will be necessary to take two genomes and perform crossover. The output of the crossover will be mutated and used as the child to be put in the next generation. The next question is then in what way will crossover be implemented. As the idea is to hopefully combine good dimensions from different individuals, fixed point crossover will be used. For simplicity, which parent a dimension is taken from for the child will be alternated. So parent one will pass on dimensions one, three, five, seven and nine.

## 3.2 Results

The result of the hypothesis test is that crossover made no difference at all. As can be seen in figure 5, the output of the populations when performing crossover is almost identical to that of the populations without crossover (figure 3b). No change is seen either in the objective or subjective fitness.

This initially caused some confusion. While it wasn't necessarily expected to remove the problem completely, some movement was expected. It was suspected after a while that the problem was to do with a lack of diversity in the populations, at least among the high scoring genomes. To test this, one population was selected. For every genome in the population, if it won more than a quarter of the games played the highest scoring dimension was recorded. Those who scored less are ignored as they are deemed unlikely to reproduce. From that, it was determined which dimension was most favored by high scorers that generation, and what proportion of high scorers had optimised that dimension. This was recorded for every generation, along with a count of all individuals that scored above the cap. This was graphed accross all generations, and can be seen in figure 6. The blue line represents the total of players above the cap, and the red line the number that optimised the same dimension. The y axis ranges from 0 at the bottom to the total population count, and the x axis ranges from 0 to 600.

As can be seen in figure 6, there is very little diversity amongst individuals likely to reproduce. Most of the time, the number of individuals above the cap and the number that have optimised the same dimension

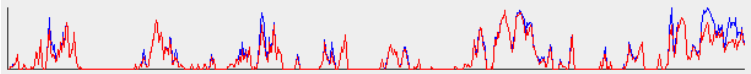


Figure 6: Test of Diversity

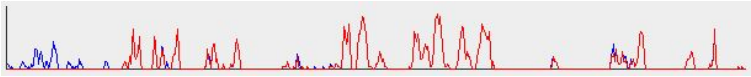


Figure 7: Number of players with a maximised dimension

are the same. This means that crossover will have no effect as they have all built the same blocks, and nothing is gained during crossover. However, when observing the value for whether the best dimension has also been maximised it turns out that this is usually the case. As can be seen in figure 7. The axis of the graph are the same as the previous one, and the blue line again represents players above the cap. However, the red line in this case represents those players who score 10 in their best dimension. In conclusion, the hypothesis was incorrect. Adding crossover to the reproduction of new individuals does not alleviate the problem of focussing in coevolution. Based on the results, however, it would be interesting to redo the experiment with some diversity maintenance. As has been shown, the players do build the whole of their block. The issue is with diversity, and that there is nothing to be gained in crossover in this set up. If diversity maintenance were introduced, that could make crossover worthwhile and alleviate the focussing problem.

# References


[1] Stephanie Forrest and Melanie Mitchell. Relative building-block fitness and the building-block hypothesis. *Ann Arbor*, 1001:48109, 1993.

[2] Richard A Watson and Thomas Jansen. A building-block royal road where crossover is provably essential. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1452–1459. ACM, 2007.

[3] Richard A Watson and Jordan B Pollack. Coevolutionary dynamics in a minimal substrate. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 702–709. Morgan Kaufmann, 2001.


# A GameRunner

```
1  package controller;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import javax.swing.JFrame;
7
8  import model.DefaultMember;
9  import model.MemberInterface;
10 import model.Population;
11 import view.MultiDimView;
12 import view.OneDimGraphPanel;
13
14 public class GameRunner {
```

```
15
16      private final static int SAMPLE_SIZE = 15, POP_SIZE = 25;
17
18      public static void main(String[] args) {
19
20          oneDimGame();
21          greatestGapGame();
22          smallestGapGame();
23          greatestGapCrossover();
24      }
25
26      private static void greatestGapCrossover(){
27          ArrayList<List<MemberInterface>> popOneHistory, popTwoHistory;
28          popOneHistory = new ArrayList<List<MemberInterface>>();
29          popTwoHistory = new ArrayList<List<MemberInterface>>();
30
31
32          Population popControllerOne, popControllerTwo;
33
34          List<MemberInterface> popOne, popTwo;
35          popOne = createTenDim();
36          popTwo = createTenDim();
37
38          popControllerOne = new Population(popOne, 0, true);
39          popControllerTwo = new Population(popTwo, 1, false);
40
41          GreatestGapScorer scorer = new GreatestGapScorer(SAMPLE_SIZE, new
                Population[]{popControllerOne, popControllerTwo});
42          popControllerOne.setScorer(scorer);
43          popControllerTwo.setScorer(scorer);
44
45          for(int i = 0; i < 600; i++){
46
47              List<MemberInterface> newOne, newTwo;
48
49              newOne = popControllerOne.evolvePopulation();
50              newTwo = popControllerTwo.evolvePopulation();
51
52              popOneHistory.add(newOne);
53              popTwoHistory.add(newTwo);
54
55              popControllerOne.setPopulation(newOne);
56              popControllerTwo.setPopulation(newTwo);
57
58          }
59
60          popControllerOne.evolvePopulation();
61          popControllerTwo.evolvePopulation();
62
63          JFrame frame = new JFrame("Greatest Gap Crossover");
64
```

```
65          MultiDimView panel = new MultiDimView(popOneHistory, popTwoHistory
                , SAMPLE_SIZE);
66
67          frame.setBounds(50,50, 1000, 600);
68
69          frame.setContentPane(panel);
70
71          frame.setVisible(true);
72      }
73
74      private static void smallestGapGame(){
75          ArrayList<List<MemberInterface>> popOneHistory, popTwoHistory;
76          popOneHistory = new ArrayList<List<MemberInterface>>();
77          popTwoHistory = new ArrayList<List<MemberInterface>>();
78
79
80          Population popControllerOne, popControllerTwo;
81
82          List<MemberInterface> popOne, popTwo;
83          popOne = createTenDim();
84          popTwo = createTenDim();
85
86          popControllerOne = new Population(popOne, 0, false);
87          popControllerTwo = new Population(popTwo, 1, false);
88
89          SmallestGapScorer scorer = new SmallestGapScorer(SAMPLE_SIZE, new
                Population[]{popControllerOne, popControllerTwo});
90          popControllerOne.setScorer(scorer);
91          popControllerTwo.setScorer(scorer);
92
93          for(int i = 0; i < 600; i++){
94
95              List<MemberInterface> newOne, newTwo;
96
97              newOne = popControllerOne.evolvePopulation();
98              newTwo = popControllerTwo.evolvePopulation();
99
100             popOneHistory.add(newOne);
101             popTwoHistory.add(newTwo);
102
103             popControllerOne.setPopulation(newOne);
104             popControllerTwo.setPopulation(newTwo);
105
106         }
107
108         popControllerOne.evolvePopulation();
109         popControllerTwo.evolvePopulation();
110
111         JFrame frame = new JFrame("Smallest Gap");
112
113         MultiDimView panel = new MultiDimView(popOneHistory, popTwoHistory
```

```
            , SAMPLE_SIZE);

        frame.setBounds(50,50, 1000, 600);

        frame.setContentPane(panel);

        frame.setVisible(true);
    }

    private static void greatestGapGame(){
        ArrayList<List<MemberInterface>> popOneHistory, popTwoHistory;
        popOneHistory = new ArrayList<List<MemberInterface>>();
        popTwoHistory = new ArrayList<List<MemberInterface>>();


        Population popControllerOne, popControllerTwo;

        List<MemberInterface> popOne, popTwo;
        popOne = createTenDim();
        popTwo = createTenDim();

        popControllerOne = new Population(popOne, 0, false);
        popControllerTwo = new Population(popTwo, 1, false);

        GreatestGapScorer scorer = new GreatestGapScorer(SAMPLE_SIZE, new
            Population[]{popControllerOne, popControllerTwo});
        popControllerOne.setScorer(scorer);
        popControllerTwo.setScorer(scorer);

        for(int i = 0; i < 600; i++){

            List<MemberInterface> newOne, newTwo;

            newOne = popControllerOne.evolvePopulation();
            newTwo = popControllerTwo.evolvePopulation();

            popOneHistory.add(newOne);
            popTwoHistory.add(newTwo);

            popControllerOne.setPopulation(newOne);
            popControllerTwo.setPopulation(newTwo);

        }

        popControllerOne.evolvePopulation();
        popControllerTwo.evolvePopulation();

        JFrame frame = new JFrame("Greatest Gap");

        MultiDimView panel = new MultiDimView(popOneHistory, popTwoHistory
            , SAMPLE_SIZE);
```

```
162
163         frame.setBounds(50,50, 1000, 600);
164
165         frame.setContentPane(panel);
166
167         frame.setVisible(true);
168     }
169
170     private static void oneDimGame(){
171         ArrayList<List<MemberInterface>> popOneHistory, popTwoHistory;
172         popOneHistory = new ArrayList<List<MemberInterface>>();
173         popTwoHistory = new ArrayList<List<MemberInterface>>();
174
175
176         Population popControllerOne, popControllerTwo;
177
178         List<MemberInterface> popOne, popTwo;
179         popOne = createOneDim();
180         popTwo = createOneDim();
181
182         popControllerOne = new Population(popOne, 0, false);
183         popControllerTwo = new Population(popTwo, 1, false);
184
185
186         OneDimScorer scorer = new OneDimScorer(SAMPLE_SIZE, new Population
                []{popControllerOne, popControllerTwo});
187
188         popControllerOne.setScorer(scorer);
189         popControllerTwo.setScorer(scorer);
190
191         for(int i = 0; i < 600; i++){
192
193             List<MemberInterface> newOne, newTwo;
194
195             newOne = popControllerOne.evolvePopulation();
196             newTwo = popControllerTwo.evolvePopulation();
197
198             popOneHistory.add(newOne);
199             popTwoHistory.add(newTwo);
200
201             popControllerOne.setPopulation(newOne);
202             popControllerTwo.setPopulation(newTwo);
203
204         }
205
206         popControllerOne.evolvePopulation();
207         popControllerTwo.evolvePopulation();
208
209         JFrame frame = new JFrame("One dimensional");
210
211         OneDimGraphPanel panel = new OneDimGraphPanel(popOneHistory,
```

```
            popTwoHistory, SAMPLE_SIZE);

        frame.setBounds(50,50, 1000, 600);

        frame.setContentPane(panel);

        frame.setVisible(true);
    }

    private static List<MemberInterface> createOneDim(){

        ArrayList<MemberInterface> popOne;
        popOne = new ArrayList<MemberInterface>();



        for(int i = 0; i < POP_SIZE; i++){

            int[][] genomeOne = new int[1][];

            genomeOne[0] = new int[100];

            for(int j = 0; j < 100; j++){
                genomeOne[0][j] = 0;
            }

            popOne.add(new DefaultMember(genomeOne));

        }

        return popOne;
    }

    private static List<MemberInterface> createTenDim(){

        ArrayList<MemberInterface> popOne;
        popOne = new ArrayList<MemberInterface>();


        for(int i = 0; i < POP_SIZE; i++){

            int[][] genomeOne = new int[10][];

            for(int k = 0; k < 10; k++){
                genomeOne[k] = new int[10];

                for(int j = 0; j < 10; j++){
                    genomeOne[k][j] = 0;
                }
            }
            popOne.add(new DefaultMember(genomeOne));
```

```
262
263         }
264
265         return popOne;
266     }
267
268 }
```

# B GreatestGapScorer

```
1  package controller;
2
3  import java.util.Set;
4
5  import model.MemberInterface;
6  import model.Population;
7  import model.ScoringInterface;
8
9  public class GreatestGapScorer implements ScoringInterface {
10
11     private int sampleSize;
12     private Population[] populations;
13
14     public GreatestGapScorer(int sampleSize, Population[] populations) {
15         super();
16         this.sampleSize = sampleSize;
17         this.populations = populations;
18     }
19
20     @Override
21     public int score(MemberInterface member, int parentPopulation) {
22         int opposingPopulation;
23         opposingPopulation = Math.abs(parentPopulation - 1);
24
25         Set<MemberInterface> sample = populations[opposingPopulation].
               getSample(sampleSize);
26
27         int score = 1;
28
29         for(MemberInterface m : sample){
30
31             int largestGap = -1, dimensionPicked = 0;
32
33             for(int i = 0; i < 10; i++){
34                 int gap = Math.abs(m.getMemberValue()[i] - member.
                       getMemberValue()[i]);
35
36                 if(gap > largestGap){
37                     largestGap = gap;
38                     dimensionPicked = i;
39                 }
```

```
40
41              }
42
43              if(member.getMemberValue()[dimensionPicked] > m.getMemberValue
                    ()[dimensionPicked]){
44                  score++;
45              }
46
47          }
48
49          return score;
50      }
51
52  }
```

# C OneDimScorer

```
1   package controller;
2
3   import java.util.Set;
4
5   import model.MemberInterface;
6   import model.Population;
7   import model.ScoringInterface;
8
9   public class OneDimScorer implements ScoringInterface {
10
11      private int sampleSize;
12      private Population[] populations;
13
14      public OneDimScorer(int sampleSize, Population[] populations) {
15          super();
16          this.sampleSize = sampleSize;
17          this.populations = populations;
18      }
19
20
21      @Override
22      public int score(MemberInterface member, int parentPopulation) {
23
24          int opposingPopulation;
25
26          opposingPopulation = Math.abs(parentPopulation - 1);
27          Set<MemberInterface> sample = populations[opposingPopulation].
                getSample(sampleSize);
28
29          int score = 1;
30
31          for(MemberInterface m : sample){
32              if(m.getMemberValue()[0] < member.getMemberValue()[0]){
33                  score++;
```

```
            }
        }

        return score;
    }

}
```

# D SmallestGapScorer

```
package controller;

import java.util.Set;

import model.MemberInterface;
import model.Population;
import model.ScoringInterface;

public class SmallestGapScorer implements ScoringInterface {

    private int sampleSize;
    private Population[] populations;

    public SmallestGapScorer(int sampleSize, Population[] populations) {
        super();
        this.sampleSize = sampleSize;
        this.populations = populations;
    }

    @Override
    public int score(MemberInterface member, int parentPopulation) {
        int opposingPopulation;
        opposingPopulation = Math.abs(parentPopulation - 1);

        Set<MemberInterface> sample = populations[opposingPopulation].
           getSample(sampleSize);

        int score = 1;

        for(MemberInterface m : sample){

            int smallestGap = Integer.MAX_VALUE, dimensionPicked = 0;

            for(int i = 0; i < 10; i++){
                int gap = Math.abs(m.getMemberValue()[i] - member.
                   getMemberValue()[i]);

                if(gap < smallestGap){
                    smallestGap = gap;
                    dimensionPicked = i;
                }
```

```
40
41              }
42
43              if(member.getMemberValue()[dimensionPicked] > m.getMemberValue
                    ()[dimensionPicked]){
44                  score++;
45              }
46
47          }
48
49          return score;
50      }
51
52  }
```

# E DefaultMember

```
1   package model;
2
3   import java.lang.reflect.Array;
4
5   public class DefaultMember implements MemberInterface {
6
7       public static double MUTATIONRATE = 0.005;
8
9       private int subjectiveFitness;
10      private int[][] genome;
11
12      @Override
13      public int compareTo(MemberInterface arg0) {
14
15          return subjectiveFitness - arg0.getSubjectiveFitness();
16
17      }
18
19      @Override
20      public int[][] getMemberGenome(){
21          return genome;
22      }
23
24      @Override
25      public int[] getMemberValue() {
26
27          int dimensions = Array.getLength(genome);
28          int[] usefulValue = new int[dimensions];
29          int dimSize = Array.getLength(genome[0]);
30
31          for(int i = 0; i < dimensions; i++){
32
33              int count = 0;
34
```

```
35              for(int j = 0; j < dimSize; j++){
36                  count+= genome[i][j];
37              }
38
39              usefulValue[i] = count;
40
41          }
42
43          return usefulValue;
44      }
45
46      @Override
47      public int getSubjectiveFitness() {
48
49          return subjectiveFitness;
50      }
51
52      @Override
53      public void setSubjectiveFitness(int score) {
54          subjectiveFitness = score;
55
56      }
57
58      @Override
59      public MemberInterface evolveMember() {
60
61          int dimensions = Array.getLength(genome);
62          int dimSize = Array.getLength(genome[0]);
63
64          int[][] newMemberGenome = new int[dimensions][];
65
66          for(int i =0; i< dimensions; i++){
67              newMemberGenome[i] = new int[dimSize];
68
69              for(int j = 0; j < dimSize; j++){
70
71                  newMemberGenome[i][j] = genome[i][j];
72
73                  if(Math.random()<MUTATIONRATE) {
74
75                      newMemberGenome[i][j] = Math.abs(newMemberGenome[i][j]
                            - 1);
76
77                  }
78
79              }
80
81          }
82
83          return new DefaultMember(newMemberGenome);
84      }
```

```
85
86
87      public DefaultMember(int[][] newGenome){
88          genome = newGenome;
89      }
90
91  }
```

## F MemberInterface

```
1   package model;
2
3   public interface MemberInterface extends Comparable<MemberInterface>{
4
5       public int[] getMemberValue();
6       public int getSubjectiveFitness();
7       public void setSubjectiveFitness(int score);
8       public MemberInterface evolveMember();
9       public int[][] getMemberGenome();
10  }
```

## G Population

```
1   package model;
2
3   import java.lang.reflect.Array;
4   import java.util.ArrayList;
5   import java.util.Collections;
6   import java.util.HashSet;
7   import java.util.List;
8   import java.util.Random;
9   import java.util.Set;
10
11  public class Population {
12
13      private ScoringInterface scorer;
14      private List<MemberInterface> currentPopulation;
15      private int popNum;
16      private boolean crossover;
17
18      public Population(List<MemberInterface> initialPopulation, int
            popNumber, boolean doesCrossover){
19          currentPopulation = initialPopulation;
20          popNum = popNumber;
21          crossover = doesCrossover;
22      }
23
24      public void setScorer(ScoringInterface scorer){
25          this.scorer = scorer;
26      }
27
```

```
28      public Set<MemberInterface> getSample(int sampleSize){
29          HashSet<MemberInterface> sample = new HashSet<MemberInterface>();
30          Random selector = new Random();
31
32          for(int i = 0; i < sampleSize; i++){
33
34              boolean newElement = false;
35              while(!newElement){
36                  int memberNum = selector.nextInt(currentPopulation.size())
                        ;
37                  MemberInterface member = currentPopulation.get(memberNum);
38
39                  newElement = sample.add(member);
40
41              }
42
43          }
44
45          return sample;
46      }
47
48      private int evaluatePopulation(){
49
50          int totalScore = 0;
51
52          for(MemberInterface member : currentPopulation){
53              int subjectiveScore = scorer.score(member, popNum);
54              member.setSubjectiveFitness(subjectiveScore);
55              totalScore += subjectiveScore;
56          }
57
58          return totalScore;
59
60      }
61
62      public List<MemberInterface> evolvePopulation(){
63          ArrayList<MemberInterface> newPopulation = new ArrayList<
                MemberInterface>();
64
65          int totalScore = evaluatePopulation();
66          Collections.sort(currentPopulation);
67
68          for(int i =0; i < currentPopulation.size(); i++){
69              int member = fitnessProportionateSelect(totalScore);
70
71              MemberInterface memberToEvolve = currentPopulation.get(member)
                    ;
72
73              if(crossover){
74                  int partner;
75                  do{
```

```
76                      partner = fitnessProportionateSelect(totalScore);
77                  } while(partner != member);
78
79                  memberToEvolve = crossoverMembers(memberToEvolve,
                        currentPopulation.get(partner));
80
81              }
82
83              newPopulation.add(currentPopulation.get(member).evolveMember()
                    );
84          }
85
86          return newPopulation;
87      }
88
89      public void setPopulation(List<MemberInterface> newPop){
90          currentPopulation = newPop;
91      }
92
93      private MemberInterface crossoverMembers(MemberInterface parentOne,
            MemberInterface parentTwo){
94
95          int[][] genome;
96          int[][] parentOneGenome = parentOne.getMemberGenome();
97          int[][] parentTwoGenome = parentTwo.getMemberGenome();
98
99          int numDimensions = Array.getLength(parentOneGenome); //doesn't
                matter which parent is used
100
101         genome = new int[numDimensions][];
102
103         for(int i = 0; i < numDimensions; i++){
104
105             if( 0 != i % 2){
106
107                 genome[i] = parentOneGenome[i].clone();
108
109             } else {
110
111                 genome[i] = parentTwoGenome[i].clone();
112
113             }
114
115         }
116
117         return new DefaultMember(genome);
118
119     }
120
121     private int fitnessProportionateSelect(int totalScore){
122
```

```
123         Random picker = new Random();
124         int pick = picker.nextInt(totalScore) + 1;
125 
126         int cummulativeScore = 0;
127         int currentMember = -1;
128 
129         while(cummulativeScore < pick){
130             currentMember ++;
131             cummulativeScore += currentPopulation.get(currentMember).
                    getSubjectiveFitness();
132         }
133 
134         return currentMember;
135     }
136 
137 }
```

## H ScoringInterface

```
1  package model;
2 
3  public interface ScoringInterface {
4 
5      public int score(MemberInterface member, int parentPopulation);
6 
7  }
```

## I MultiDimView

```
1  package view;
2 
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.util.HashMap;
7  import java.util.List;
8 
9  import javax.swing.JPanel;
10 
11 import model.MemberInterface;
12 
13 @SuppressWarnings("serial")
14 public class MultiDimView extends JPanel {
15 
16     private List<List<MemberInterface>> popOne, popTwo;
17     private double sampleSize;
18     public MultiDimView(List<List<MemberInterface>> popOne, List<List<
           MemberInterface>> popTwo, int sampleSize) {
19         super();
20         this.popOne = popOne;
21         this.popTwo = popTwo;
```

```
22          this.sampleSize = (double) sampleSize;
23      }
24
25      @Override
26      public void paint(Graphics g){
27
28          super.paint(g);
29
30          Graphics2D graphTwoDim = (Graphics2D) g;
31
32          graphTwoDim.drawString("100", 30, 10);
33          graphTwoDim.drawString("Objective", 2, 45);
34          graphTwoDim.drawString("fitness", 2, 55);
35          graphTwoDim.drawString("0", 35, 110);
36
37          graphTwoDim.drawLine(60, 5, 660, 5);
38          graphTwoDim.drawLine(60, 5, 60, 105);
39          graphTwoDim.drawLine(60, 105, 660, 105);
40
41          graphTwoDim.drawLine(60, 150, 60, 200);
42          graphTwoDim.drawLine(60, 200, 660, 200);
43
44          graphTwoDim.drawString("1", 35, 155);
45          graphTwoDim.drawString("0", 35, 205);
46
47          graphTwoDim.drawString("Subjective", 2, 175);
48          graphTwoDim.drawString("fitness", 2, 185);
49
50          graphTwoDim.drawString("0", 60, 120);
51          graphTwoDim.drawString("600", 660, 120);
52          graphTwoDim.drawString("Generation", 360, 120);
53
54          graphTwoDim.drawLine(60, 250, 60, 300);
55          graphTwoDim.drawLine(60, 300, 660, 300);
56
57          double popSize = (double) popOne.get(0).size();
58          int[] highScoringSameDimCount = new int[600], numberHighScoring =
                new int[600];
59
60          for(int i = 0; i < 600; i++){
61              HashMap<Integer, Integer> count = setUpMap();
62              int totalSubjectiveFitnessOne=0, totalSubjectiveFitnessTwo=0;
63              numberHighScoring[i] = 0;
64              for(int j = 0; j < popSize; j++){
65
66                  int valOne = tallyDimensions(popOne.get(i).get(j));
67                  int valTwo = tallyDimensions(popTwo.get(i).get(j));
68
69                  graphTwoDim.setColor(Color.BLUE);
70                  graphTwoDim.drawLine(61 + i, 105 - valOne, 61 + i, 105 -
                        valOne);
```

```
71
72                  graphTwoDim.setColor(Color.RED);
73                  graphTwoDim.drawLine(61 + i, 105 - valTwo, 61 + i, 105 -
                        valTwo);
74
75                  totalSubjectiveFitnessOne += popOne.get(i).get(j).
                        getSubjectiveFitness() - 1; //correct for scoring 'fix'
76                  totalSubjectiveFitnessTwo += popTwo.get(i).get(j).
                        getSubjectiveFitness() - 1;
77
78                  if(popOne.get(i).get(j).getSubjectiveFitness() - 1 >= Math
                        .round(sampleSize + 1 / 4)){
79                      numberHighScoring[i]++;
80                      count.put(getBestDimension(popOne.get(i).get(j)),
                            count.get(getBestDimension(popOne.get(i).get(j))) +
                             1);
81                  }
82
83              }
84
85              double averageFitnessOne, averageFitnessTwo;
86
87
88              averageFitnessOne = ((double) totalSubjectiveFitnessOne) /
                    popSize;
89              averageFitnessTwo = ((double) totalSubjectiveFitnessTwo) /
                    popSize;
90
91              averageFitnessOne = averageFitnessOne / sampleSize; //put
                    between 0 and 1;
92              averageFitnessTwo = averageFitnessTwo / sampleSize;
93
94              int offsetOne, offsetTwo;
95
96              offsetOne = (int) Math.round(averageFitnessOne * 50);
97              offsetTwo = (int) Math.round(averageFitnessTwo * 50);
98
99              graphTwoDim.setColor(Color.BLUE);
100             graphTwoDim.fillRect(59 + i, 198 - offsetOne, 4, 4);
101
102             graphTwoDim.setColor(Color.RED);
103             graphTwoDim.fillRect(59 + i, 198 - offsetTwo, 4, 4);
104
105             highScoringSameDimCount[i] = getBestCount(count);
106
107         }
108         double modifier = 50 / popSize;
109         for(int i = 0; i < 599; i++){
110
111             graphTwoDim.setColor(Color.BLUE);
112             graphTwoDim.drawLine(61 + i, (int) (300 - Math.round(
```

```
                numberHighScoring[i] * modifier)), 62 + i, (int) (300 -
                Math.round(numberHighScoring[i + 1] * modifier)));
113             graphTwoDim.setColor(Color.RED);
114             graphTwoDim.drawLine(61 + i, (int) (300 - Math.round(
                    highScoringSameDimCount[i] * modifier)), 62 + i, (int) (300
                     - Math.round(highScoringSameDimCount[i + 1] * modifier)));
115         }
116
117     }
118
119     private HashMap<Integer, Integer> setUpMap(){
120         HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
121
122         for(int i = 0; i<= 10; i++){
123             map.put(i, 0);
124         }
125
126         return map;
127     }
128
129     private int getBestCount(HashMap<Integer, Integer> counts){
130
131         int currentBest= -1;
132
133         for(int i = 0; i < 10; i++){
134             if(counts.get(i) > currentBest){
135                 currentBest = counts.get(i);
136             }
137         }
138         //System.out.println("best count: " + currentBest);
139         //return currentBest;
140         return counts.get(10);
141     }
142
143     private int getBestDimension(MemberInterface member){
144         int currentBestDim = -1, currentBest= -1;
145
146         int[] values = member.getMemberValue();
147
148         for(int i = 0; i < 10; i++){
149             if(values[i] > currentBest){
150                 currentBest = values[i];
151                 currentBestDim = i;
152             }
153         }
154
155         System.out.println(currentBest);
156
157         return currentBest;
158     }
159
```

```
160     private int tallyDimensions(MemberInterface member){
161 
162         int objectiveFitness = 0;
163 
164         int[]values = member.getMemberValue();
165 
166         for(int v : values){
167 
168             objectiveFitness += v;
169 
170         }
171 
172         return objectiveFitness;
173     }
174 
175 }
```

# J OneDimGraphPanel

```
1  package view;
2 
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.util.List;
7 
8  import javax.swing.JPanel;
9 
10 import model.MemberInterface;
11 
12 @SuppressWarnings("serial")
13 public class OneDimGraphPanel extends JPanel {
14 
15     private List<List<MemberInterface>> popOne, popTwo;
16     private double sampleSize;
17     public OneDimGraphPanel(List<List<MemberInterface>> popOne, List<List<
           MemberInterface>> popTwo, int sampleSize) {
18         super();
19         this.popOne = popOne;
20         this.popTwo = popTwo;
21         this.sampleSize = (double) sampleSize;
22     }
23 
24     @Override
25     public void paint(Graphics g){
26 
27         super.paint(g);
28 
29         Graphics2D graphTwoDim = (Graphics2D) g;
30 
31         graphTwoDim.drawString("100", 30, 10);
```

```
32          graphTwoDim.drawString("Objective", 2, 45);
33          graphTwoDim.drawString("fitness", 2, 55);
34          graphTwoDim.drawString("0", 35, 110);
35
36          graphTwoDim.drawLine(60, 5, 660, 5);
37          graphTwoDim.drawLine(60, 5, 60, 105);
38          graphTwoDim.drawLine(60, 105, 660, 105);
39
40          graphTwoDim.drawLine(60, 150, 60, 200);
41          graphTwoDim.drawLine(60, 200, 660, 200);
42
43          graphTwoDim.drawString("1", 35, 155);
44          graphTwoDim.drawString("0", 35, 205);
45
46          graphTwoDim.drawString("Subjective", 2, 175);
47          graphTwoDim.drawString("fitness", 2, 185);
48
49          graphTwoDim.drawString("0", 60, 120);
50          graphTwoDim.drawString("600", 660, 120);
51          graphTwoDim.drawString("Generation", 360, 120);
52          double popSize = (double) popOne.get(0).size();
53          for(int i = 0; i < 600; i++){
54
55              int totalSubjectiveFitnessOne=0, totalSubjectiveFitnessTwo=0;
56
57              for(int j = 0; j < (int) popSize; j++){
58
59                  int valOne = popOne.get(i).get(j).getMemberValue()[0];
60                  int valTwo = popTwo.get(i).get(j).getMemberValue()[0];
61
62                  graphTwoDim.setColor(Color.BLUE);
63                  graphTwoDim.drawLine(61 + i, 105 - valOne, 61 + i, 105 -
                        valOne);
64
65                  graphTwoDim.setColor(Color.RED);
66                  graphTwoDim.drawLine(61 + i, 105 - valTwo, 61 + i, 105 -
                        valTwo);
67
68                  totalSubjectiveFitnessOne += popOne.get(i).get(j).
                        getSubjectiveFitness() - 1; //correct for scoring 'fix'
69                  totalSubjectiveFitnessTwo += popTwo.get(i).get(j).
                        getSubjectiveFitness() - 1;
70
71              }
72
73              double averageFitnessOne, averageFitnessTwo;
74
75
76
77              averageFitnessOne = ((double) totalSubjectiveFitnessOne) /
                    popSize;
```

```
78              averageFitnessTwo = ((double) totalSubjectiveFitnessTwo) /
                    popSize;
79
80              averageFitnessOne = averageFitnessOne / sampleSize; //put
                    between 0 and 1;
81              averageFitnessTwo = averageFitnessTwo / sampleSize;
82
83              int offsetOne, offsetTwo;
84
85              offsetOne = (int) Math.round(averageFitnessOne * 50);
86              offsetTwo = (int) Math.round(averageFitnessTwo * 50);
87
88              graphTwoDim.setColor(Color.BLUE);
89              graphTwoDim.fillRect(59 + i, 198 - offsetOne, 4, 4);
90
91              graphTwoDim.setColor(Color.RED);
92              graphTwoDim.fillRect(59 + i, 198 - offsetTwo, 4, 4);
93
94          }
95
96      }
97
98
99
100 }
```