HOMEWORK 5 – MAX-SAT SOLUTION USING A GENETIC ALGORITHM

BY ANDRÉS RUBIO RAMOS

WHY GENETIC ALGORITHMS?

As I said in the previous homework I have chosen to do homework 4 and 5 on genetic algorithms. The reason is that I have already worked with them, I feel much more comfortable and I like them much more than the others. I have worked with them on a course at my home university in Seville (Spain), University of Seville.

DATASET

I am going to work with generated sets of weighted 3-SAT benchmark instances that you have recommended to us. The instances wuf-M, wuf-N, wuf-Q, wuf-R and wuf-A. Of which I will choose the ones that have the optimal solution calculated in order to calculate the relative error of the solutions provided by my algorithm.

CODE

I have relied mainly on the "org.apache.commons.math3.genetics" library , which has a lot of variety of chromosomes, mutations, crossovers, stop conditions... This library was used by my professor Miguel Toro to create a more direct implementation of the algorithms, which is in the java project called "Geneticos" that I added in the code zip.

I have divided the code used into 4 Java classes and some modifications to the Apache code:

DataMaxSat class

In this class is where I have the information about the instance of the MAX-SAT problem to be solved.

The attributes of the class are an integer “n” with the number of variables in the problem, a list of the weights of the corresponding variables and a list with the information of the different clauses.

```java
public class DataMaxSat {
    public static record Clause(Integer first, Integer second, Integer third) {
        public Integer absFirst() {
            //-1 because clauses variables start at 1 and no in 0
            return Math.abs(first)-1;
        }
        public Integer absSecond() {
            return Math.abs(second)-1;
        }
        public Integer absThird() {
            return Math.abs(third)-1;
        }
        public Integer valueFirst() {
            Integer value=0;
            if (first>0) value=1;
            return value;
        }
        public Integer valueSecond() {
            Integer value=0;
            if (second>0) value=1;
            return value;
        }
        public Integer valueThird() {
            Integer value=0;
            if (third>0) value=1;
            return value;
        }
    }
    public static Integer n; //Number variables
    public static List<Integer> weights;
    public static List<Clause> clauses;
```

This is the function that initializes the data load by parsing it from the supplied input document.

```java
    public static void iniData(String file) {
        List<String> lines=Files2.linesFromFile(file);
        weights=new ArrayList<>();
        clauses=new ArrayList<>();
        for (String line:lines) {
            if (!line.isBlank()) {
                char charFirst=line.charAt(0);
                if ('w'==charFirst) {
                    Scanner scanner= new Scanner(line);
                    scanner.next();
                    while (scanner.hasNextInt()) {
                        Integer weight=scanner.nextInt();
                        if (weight!=0)
                        weights.add(weight);
                    }
                    scanner.close();
                } else if ('c'!=charFirst && 'p'!=charFirst ) {
                    String[] parts = line.trim().split("\\s+");
                    Integer[] numbers = new Integer[parts.length - 1];
                    for (int i = 0; i < parts.length-1; i++) {
                        numbers[i] = Integer.parseInt(parts[i]);
                    }
                    Clause clause = new Clause(numbers[0], numbers[1], numbers[2]);
                    clauses.add(clause);
                }
            }
        }
        n=weights.size();
    }
```

Some methods used especially in calculating fitness:

```java
public static Integer getN() {
    return n;
}
public static List<Integer> getWeights() {
    return weights;
}
//For the fitness calculation
public static Double weightAverage() {
    return weights.stream()
            .mapToInt(Integer::intValue)
            .average()
            .orElse(0.0);
}
public static List<Clause> getClauses() {
    return clauses;
}
```

```java
public static Integer getWeight(Integer i) {
    return weights.get(i);
}
public static Clause getClause(Integer i) {
    return clauses.get(i);
}
public static Integer getFirstClause(Integer i) {
    return clauses.get(i).first();
}
public static Integer getSecondClause(Integer i) {
    return clauses.get(i).second();
}
public static Integer getThirdClause(Integer i) {
    return clauses.get(i).third();
}
```

GeneticMaxSat class

In this class I have implemented what will be the type of **chromosome** to work with also its characteristics. In this case I am going to work with a chromosome that is a **binary list** (order matters). If a 1 appears in position i of the chromosome, then variable i is considered to have value 1 (true). On the other hand, if a 0 appears, then variable i is not considered to have value 0 (false).

Chromosome **size** is the total number of variables.

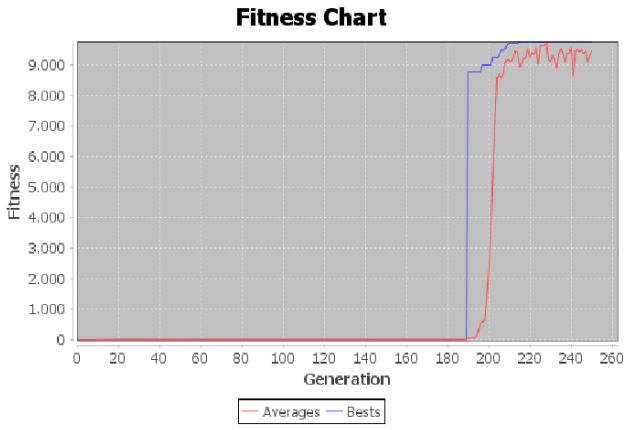
The **fitness** value is divided into two cases:

The first is if all the clauses are satisfied, then the fitness value is the sum of all the weights of the variables with a value of 1.

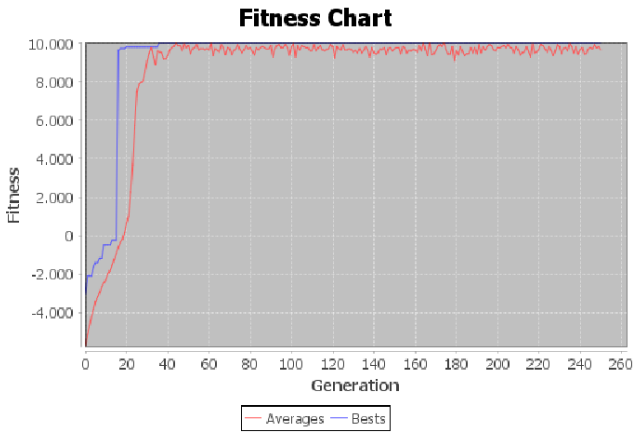
On the other hand, if there is at least one clause not satisfied, then the value of the fitness is the negative value of the product of a scalar times the number of unsatisfied clauses. I have added the scalar product since when doing the tests at the beginning the values of the graph when the fitness of the population are still negative and a valid combination is being searched are very small in absolute value compared to the fitness values when We are already working with valid combinations that we are trying to optimize their sum of weights. That is why I have decided to multiply the number of unsatisfied clauses by the average value of the weights. (see the graphs of instance "wuf50-0123.mwcnf" with 202 clauses)

The strategy is based on maximizing said value.

As we can see in the phase of searching for valid solutions, it seems that the fitness is always 0, although in this case it takes values between -202 and 0. Which are insignificant for the sum of weights values that appear in the second phase.



Here we change the value of the scalar from 1 to the average of the variable weights. Then we can better appreciate the phase of searching for valid solutions, and even obtain better results with this change.



One thing to take into account is the chromosomes that represent an **invalid configuration** for our solution. These invalid configurations would be those that generate one or more unsatisfied clauses.

To solve it, as was already the case with the fitness function used, I have decided to use a **relaxation** strategy to penalize fitness when not all clauses are satisfied. When at least one clause is not satisfied the fitness value will be negative and will decrease as more and more clauses are unsatisfied. In this way we will guide the algorithm to try to reduce the number of unsatisfied clauses until it reaches valid options and begins the optimization phase of the maximum weights of the variables.

```
public class GeneticMaxSat implements BinaryData<SolutionMaxSat>{
    public static GeneticMaxSat create(String file) {
        return new GeneticMaxSat(file);
    }
    private GeneticMaxSat(String file) {
        DataMaxSat.iniData(file);
    }
    public Integer size() {
        return DataMaxSat.getN();
    }
    public Double fitnessFunction(List<Integer> value) {
        Double fitness=0.;
        Integer nInvalidClauses=0;
        Double scalar=DataMaxSat.weightAverage();
        for (Clause clause:DataMaxSat.getClauses()) {
            //Check if the clause is not satisfied
            if (value.get(clause.absFirst())!=clause.valueFirst() &&
                    value.get(clause.absSecond())!=clause.valueSecond() &&
                    value.get(clause.absThird())!=clause.valueThird()) {
                nInvalidClauses+=1;
            }
        }
        if (nInvalidClauses==0) {
            for (Integer i=0;i<DataMaxSat.getN();i++) {
                if (value.get(i)==1) {
                    fitness+=DataMaxSat.getWeight(i);
                }
            }
        } else fitness=-scalar*nInvalidClauses;
        return fitness;
    }
    public SolutionMaxSat solucion(List<Integer> value) {
        return SolutionMaxSat.create(value);
    }
}
```

SolutionMaxSat class

In this class I have implemented the transformation of the enconding of an individual (genotype) to the abstract representation (phenotype).

In this case I have decided that the phenotype remains the binary list because the solution is more visual than “The 1st variable value is true, 2nd variable is, etc, ...". And I have also added that the fitness (the total weight or unsatisfied clauses) is shown.

```
public class SolutionMaxSat {
    private List<Integer> variables;
    private Double sumWeights;
    private SolutionMaxSat(List<Integer> value) {
        sumWeights=0.;
        variables=value;
        Double nInvalidClauses=0.;
        Double scalar=DataMaxSat.weightAverage();
        for (Clause clause:DataMaxSat.getClauses()) {
            //Check if the clause is satisfied
            if (value.get(clause.absFirst())!=clause.valueFirst() &&
                    value.get(clause.absSecond())!=clause.valueSecond() &&
                    value.get(clause.absThird())!=clause.valueThird()) {
                nInvalidClauses+=1;
            }
        }
        if (nInvalidClauses==0) {
            for (Integer i=0;i<DataMaxSat.getN();i++) {
                if (value.get(i)==1) {
                    sumWeights+=DataMaxSat.getWeight(i);
                }
            }
        } else sumWeights=-scalar*nInvalidClauses;
    }
    public static SolutionMaxSat create(List<Integer> value) {
        return new SolutionMaxSat(value);
    }
    public String toString() {
        return "Result:\nThe total weight is: "
                +sumWeights+"\nThe list of the values of the variables is: "
                +variables.toString();
    }
}
```

TestMaxSat class

In this class is where I have created the functions to do the test.

"read":

Reads the file and measures the execution time. I have added a loop to calculate the average result of different executions to have a much more objective result. Due to the random nature of the algorithm

"test_gen":

Sets the genetic parameters and runs the algorithm.

```java
public class TestMaxSat {
    public static void main(String[] args) {
        List<Double> errors=new ArrayList<>();
        for (int i=0;i<50;i++) {
            errors.add(read("files/wuf-M/wuf50-0123.mwcnf"));
        }
        System.out.println(errors.stream().mapToDouble(x->x).average().getAsDouble());
    }
    private static Double read(String path) {
        long startTime = System.nanoTime();
        Double error=test_Gen(path);
        long endTime = System.nanoTime();
        long elapsedTimeInNanoseconds = endTime - startTime;
        double elapsedTimeInSeconds = (double) elapsedTimeInNanoseconds / 1e9;
        System.out.println("Time of CPU consumed (in seconds): "+elapsedTimeInSeconds);
        return error;
    }
    private static Double test_Gen(String path) {
        AlgoritmoAG.POPULATION_SIZE=400; //Default: 30
        ChromosomeFactory.TOURNAMENT_ARITY=2; //Default: 2
        ChromosomeFactory.crossoverType=CrossoverType.OnePoint; //Default: OnePoint
        AlgoritmoAG.CROSSOVER_RATE=0.95; //Default: 0.8
        AlgoritmoAG.MUTATION_RATE=0.05; //Default: 0.6
        AlgoritmoAG.ELITISM_RATE=0.034; //Default: 0.2
        StoppingConditionFactory.stoppingConditionType=
                StoppingConditionType.GenerationCount; //Default: GenerationCount
        StoppingConditionFactory.NUM_GENERATIONS = 250; //Default: Integer.MAX_VALUE

        GeneticMaxSat problem= GeneticMaxSat.create(path);
        var alg=AlgoritmoAG.of(problem); //AlgorithmAG
        alg.ejecuta(); //execute
        graphic(alg.getPopulations()); //graphic representation of the fitness
        System.out.println(alg.bestSolution().toString());
        Double error=(1.0-(alg.getBestChromosome().fitness()/116))*100.0;
        System.out.println("Relative error: "+error+"%");
        return error;
    }
```

“graphic”:

Creates the linear graph of the average and best fitness across different generations.

```java
    private static void graphic(List<Population> populations) {
        List<Double> averages=new ArrayList<>();
        List<Double> bests=new ArrayList<>();
        for (int i=0;i<populations.size();i++) {
            Double sum=0.;
            for (Chromosome c:populations.get(i)) {
                sum+=c.getFitness();
            }
            averages.add(sum/populations.get(i).getPopulationSize());
            bests.add(populations.get(i).getFittestChromosome().getFitness());
        }
        //Graphic definition
        XYSeries averageSeries = new XYSeries("Averages");
        XYSeries bestSeries = new XYSeries("Bests");
        for (int i = 0; i < averages.size(); i++) {
            averageSeries.add(i, averages.get(i));
            bestSeries.add(i, bests.get(i));
        }
        XYSeriesCollection dataset = new XYSeriesCollection();
        dataset.addSeries(averageSeries);
        dataset.addSeries(bestSeries);
        JFreeChart chart = ChartFactory.createXYLineChart(
                "Fitness Chart",
                "Generation",
                "Fitness",
                dataset,
                PlotOrientation.VERTICAL,
                true,
                true,
                false
        );
        ChartPanel chartPanel = new ChartPanel(chart);
        chartPanel.setPreferredSize(new Dimension(560, 370));
        JFrame frame = new JFrame("Fitness Chart");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(chartPanel);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

Apache code modification (the same as for homework4)

I have had to make these changes to be able to have data from the intermediate generations and not just the initial and final ones. In order to represent the graphs with the evolution of the maximum and average fitness.

org.apache.commons.math3.genetics.GeneticAlgorithm class

I have added the "listPopulations" variable to store all the populations that are generated with the execution of the algorithm.

```
public class GeneticAlgorithm {

    private static RandomGenerator randomGenerator = new JDKRandomGenerator();

    /** the crossover policy used by the algorithm. */
    private final CrossoverPolicy crossoverPolicy;

    /** the rate of crossover for the algorithm. */
    private final double crossoverRate;

    /** the mutation policy used by the algorithm. */
    private final MutationPolicy mutationPolicy;

    /** the rate of mutation for the algorithm. */
    private final double mutationRate;

    /** the selection policy used by the algorithm. */
    private final SelectionPolicy selectionPolicy;

    /** the number of generations evolved to reach {@link StoppingCondition} in
    private int generationsEvolved = 0;

    private static List<Population> listPopulations=new ArrayList<>();
    /**
     * Create a new genetic algorithm.
     * @param crossoverPolicy The {@link CrossoverPolicy}
     * @param crossoverRate The crossover rate as a percentage (0-1 inclusive)
     * @param mutationPolicy The {@link MutationPolicy}
```

This is the function that evolves from one population to the next, and I have introduced two lines in which I add the current population to the list.

```
/**
 * Evolve the given population. Evolution stops when the stopping condition
 * is satisfied. Updates the {@link #getGenerationsEvolved() generationsEvolved}
 * property with the number of generations evolved before the StoppingCondition
 * is satisfied.
 *
 * @param initial the initial, seed population.
 * @param condition the stopping condition used to stop evolution.
 * @return the population that satisfies the stopping condition.
 */
public Population evolve(final Population initial, final StoppingCondition condition) {
    Population current = initial;
    listPopulations.add(current);
    generationsEvolved = 0;
    while (!condition.isSatisfied(current)) {
        current = nextGeneration(current);
        listPopulations.add(current);
        generationsEvolved++;
    }
    return current;
}
```

And at the end I have added the corresponding getter method.

```
     * @return number of generations evolved
     * @since 2.1
     */
    public int getGenerationsEvolved() {
        return generationsEvolved;
    }

    public List<Population> getPopulations() {
        return listPopulations;
    }
```

Experimental results

For the study of results, I will work with the provided instances with their respective optimal solutions.

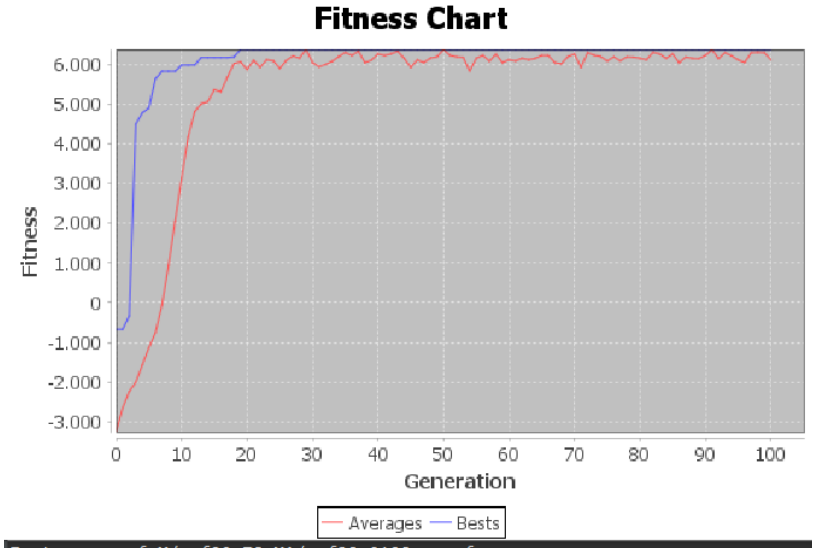
The parameters of the first version will be the following:

```
Integer popSize=200;
Double e=1.; //Number of individuals for elitism
AlgoritmoAG.POPULATION_SIZE=popSize; //Default: 30
ChromosomeFactory.TOURNAMENT_ARITY=2; //Default: 2
ChromosomeFactory.crossoverType=CrossoverType.OnePoint; //Default: OnePoint
AlgoritmoAG.CROSSOVER_RATE=0.95; //Default: 0.8
AlgoritmoAG.MUTATION_RATE=0.05; //Default: 0.6
AlgoritmoAG.ELITISM_RATE=e/popSize; //Default: 0.2
StoppingConditionFactory.stoppingConditionType=
        StoppingConditionType.GenerationCount; //Default: GenerationCount
StoppingConditionFactory.NUM_GENERATIONS = 100; //Default: Integer.MAX_VALUE
```

This time I am going to take as base parameters those recommended in the class lectures with a population of 200 and a number of generations of 100 at first. I have set the crossover rate to 0.95 instead of 0.8 since it had a better result for the previous homework.

The parameters that it says are default are the ones that appear in the Apache code. Everything seems fine except for the Mutation and elitism rate, which seem very high.

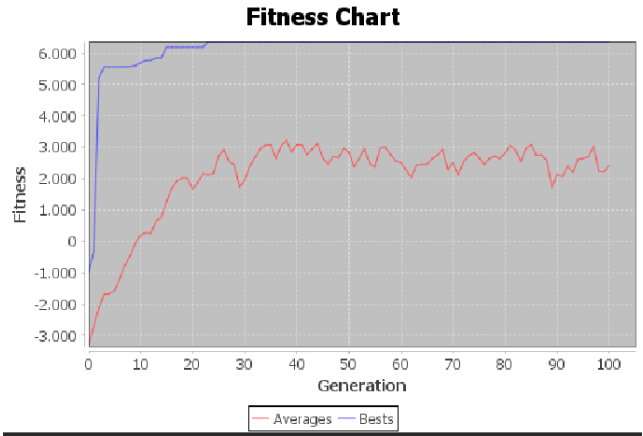
Wuf-M and Wuf-N instances




```
Instance: wuf-M/wuf20-78-M1/wuf20-0100.mwcnf
Result:
The total weight is: 6375.0
The list of the values of the variables is: [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1
Global optimum: 6375
Relative error: 0.0%
Time of CPU consumed (in seconds): 0.3666686
```

With the parameters mentioned above, this is an execution for the simplest instances. Most of the executions have returned a **0% error**, except in some very rare cases 2%. By selecting as the **scalar** the **average of the weights** of the variables we can see much more **curved** functions in the first phase. It converges appropriately for the best and for the average. Apart from the fact that the **execution time** is **very low**. Then the parameters chosen are more than decent for the easiest instances.

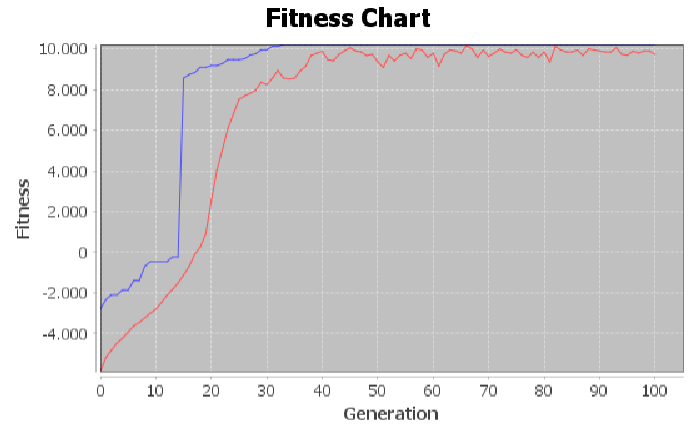
I have chosen the **mutation rate at 0.05** like in the lectures, since I did some runs like these with the crossover rate at 0.6. What happens is that there is no convergence of the fitness of the other individuals, only the elitist one. Thanks to its presence and the random search, the algorithm works as it is such a basic instance.




```
Instance: wuf-M/wuf20-78-M1/wuf20-0100.mwcnf
Result:
The total weight is: 6375.0
The list of the values of the variables is: [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1
Global optimum: 6375
Relative error: 0.0%
Time of CPU consumed (in seconds): 0.3919947
```

Now I will continue testing this configuration with larger instances. Not only 20 variables and 78 clauses.

This is an example of 50 variables and 201 clauses. Here the error of 3.04% does appear in this case. I have calculated the average of 100 executions and I get an error of 21.39% (it is higher because in some cases it does not pass the first phase so the error in that case is more than 100%)




```
Instance: wuf-M/wuf50-201-M1/wuf50-0123.mwcnf
Result:
The total weight is: 10219.0
The list of the values of the variables is: [1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0
Global optimum: 10540
Relative error: 3.045540796963941 4%
Time of CPU consumed (in seconds): 0.4119696
```

Now I have tried **increasing** the **population size**, as it worked for the knapsack problem, up to **600** and I have reached an error of around 0-6%.

Once with that error I tried to reduce it by varying some parameters but the only thing it generated was increasing the relative error more and more. Until I started to **reduce** the **crossover rate** slightly from 95% to try to maintain diversity and avoid rapid convergence to a particular region of the search space.

These are the results of decreasing it a little and try 100 executions:

-80%: 5.39% average relative error

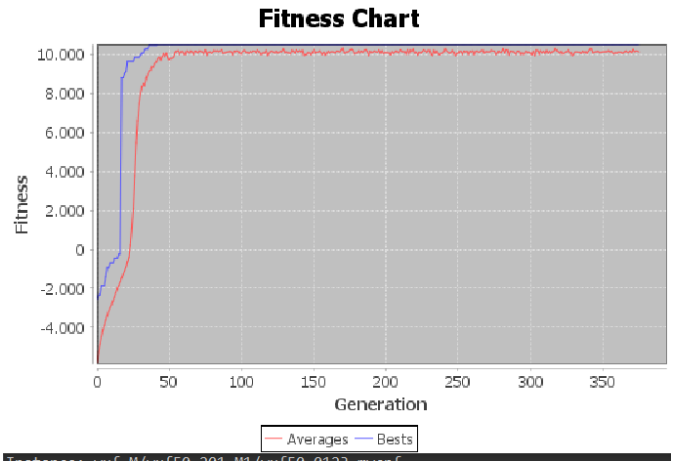
-85%: 2.43% average relative error ←

-90%: 2.65% average relative error

-95%: 3.26% average relative error

Then I will continue increasing the population and the number of generations with a crossover rate of 85%

This is a run with **0% relative error** with a population of 1500 and a number of generations of 375. (I have tried to maintain the **4/1 ratio** which was working pretty well). I have run the same configuration 50 times and it gave me an average relative error of **0.41%**

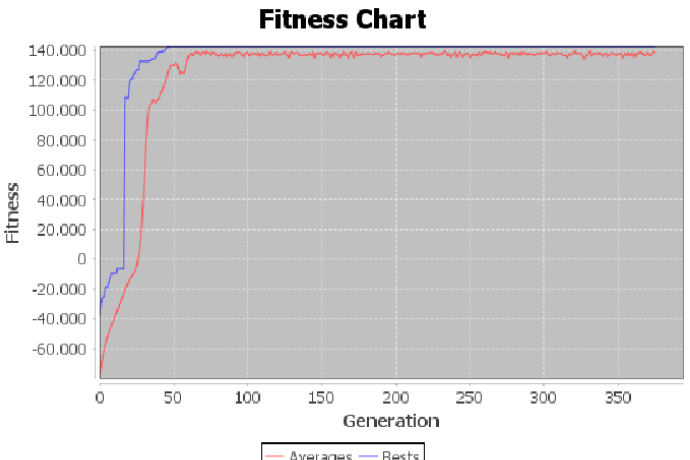



```
Instance: wuf-M/wuf50-201-M1/wuf50-0123.mwcnf
Result:
The total weight is: 10540.0
The list of the values of the variables is: [1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1
Global optimum: 10540
Relative error: 0.0%
Time of CPU consumed (in seconds): 2.5876842
```

We could say that it has improved considerably from 5% average relative error to 0.41%. This could be less if we continue to increase the population and generations, but I think it would be enough.

Now I'm going to run an instance of the wuf-N set to see how our configuration performs with this other somewhat similar instance.

Again we have another execution with **0%** relative error of a wuf-N instance with 50 variables and 201 clauses. Which has a very good behavior, **converging without problems.**

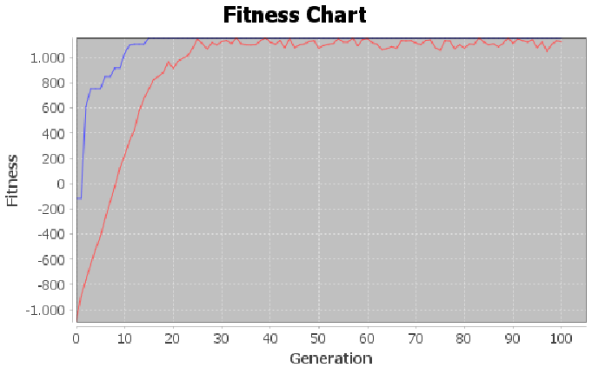



```
Instance: wuf-N/wuf50-201-N1/wuf50-0123.mwcnf
Result:
The total weight is: 143016.0
The list of the values of the variables is: [1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1
Global optimum: 143016
Relative error: 0.0%
Time of CPU consumed (in seconds): 2.5438814
```

I have run the same instance again 50 times and have obtained an **average relative error of 0.468%.** Which is very similar to the error obtained in the instance of the wuf-M set due to its similarity. Then our **algorithm and configuration** is **approximating** the solutions of different instances **quite well**.

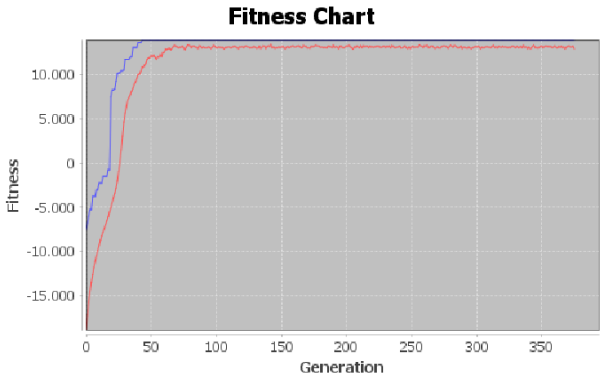
Wuf-Q instances

I have tried running the R instance with 20 variables and 78 clauses with **the initial configuration** and again it has been **effective** for these **small instances**. The convergence can be seen perfectly. Having a 0% error in the vast majority of executions.




```
Instance: wuf-Q/wuf20-78-Q1/wuf20-01000.mwcnf
Result:
The total weight is: 1158.0
The list of the values of the variables is: [0, 1, 1, 1, 1, 1, 1, 1, 1, 0
Global optimum: 1158
Relative error: 0.0%
Time of CPU consumed (in seconds): 0.3935153
```

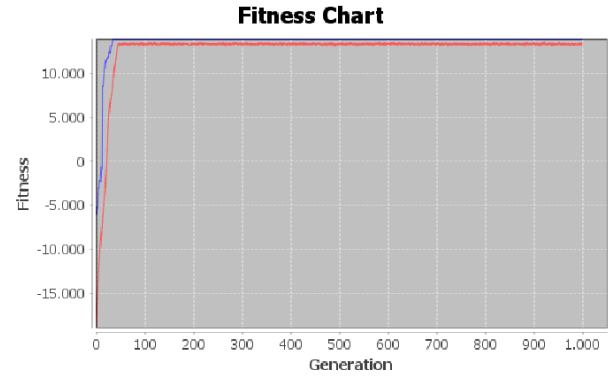
This is an execution of an instance with 50 variables and 201 clauses. I have used the **same configuration** as for large wuf-M or wuf-N instances. But in this case there has **not been good unemployment.**




```
Instance: wuf-Q/wuf50-201-Q1/wuf50-0123.mwcnf
Result:
The total weight is: 13965.0
The list of the values of the variables is: [1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0
Global optimum: 23103
Relative error: 39.55330476561486%
Time of CPU consumed (in seconds): 2.6151097
```

To try to find the optimal one, I first chose to continue increasing the population and the number of generations. And although the error decreased it continued to stop at local maximums.

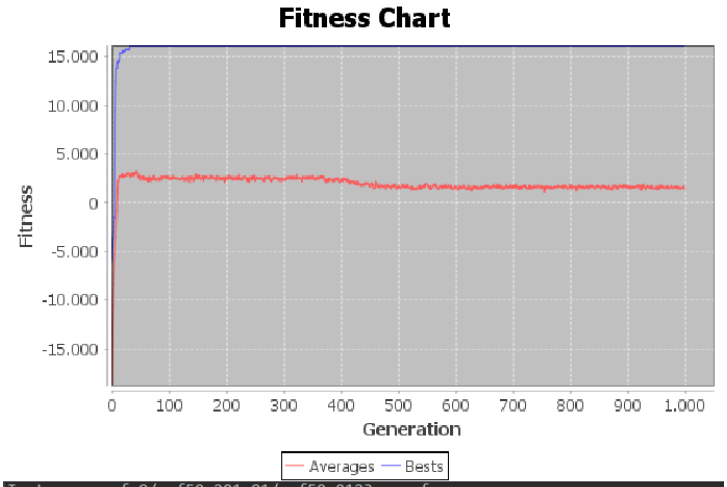
```
Integer popSize=4000; //200 1500
Double e=1.; //Number of individuals for elitism
AlgoritmoAG.POPULATION_SIZE=popSize; //Default: 30
ChromosomeFactory.TOURNAMENT_ARITY=2; //Default: 2
ChromosomeFactory.crossoverType=CrossoverType.OnePoint;
AlgoritmoAG.CROSSOVER_RATE=0.85; //Default: 0.8
AlgoritmoAG.MUTATION_RATE=0.05; //Default: 0.6
AlgoritmoAG.ELITISM_RATE=e/popSize; //Default: 0.2
StoppingConditionFactory.stoppingConditionType=
        StoppingConditionType.GenerationCount; //Defaul
StoppingConditionFactory.NUM_GENERATIONS = 1000; //Defa
```




```
Instance: wuf-Q/wuf50-201-Q1/wuf50-0123.mwcnf
Result:
The total weight is: 13965.0
The list of the values of the variables is: [1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0,
Global optimum: 23103
Relative error: 39.55330476561486%
Time of CPU consumed (in seconds): 27.1444201
```

Then I modified all the possible parameters to try to evade those local maxima:

```
Integer popSize=4000; //200 1500
Double e=1.; //Number of individuals for elitism
AlgoritmoAG.POPULATION_SIZE=popSize; //Default: 30
ChromosomeFactory.TOURNAMENT_ARITY=10; //Default: 2
ChromosomeFactory.crossoverType=CrossoverType.OnePoint;
AlgoritmoAG.CROSSOVER_RATE=0.1; //Default: 0.8
AlgoritmoAG.MUTATION_RATE=0.9; //Default: 0.6
AlgoritmoAG.ELITISM_RATE=e/popSize; //Default: 0.2
StoppingConditionFactory.stoppingConditionType=
        StoppingConditionType.GenerationCount; //Defaul
StoppingConditionFactory.NUM_GENERATIONS = 1000; //Defa
```

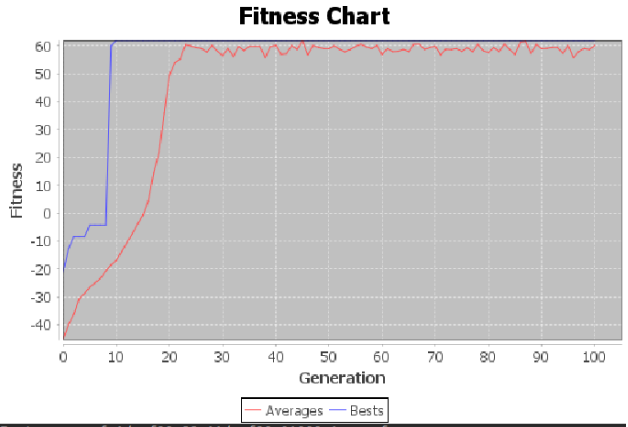



```
Instance: wuf-Q/wuf50-201-Q1/wuf50-0123.mwcnf
Result:
The total weight is: 16157.0
The list of the values of the variables is: [1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1,
Global optimum: 23103
Relative error: 30.065359477124186%
Time of CPU consumed (in seconds): 36.6397552
```

Even with a purely random search with a heuristic to avoid losing the best solutions, it was not possible to find the global maximum.

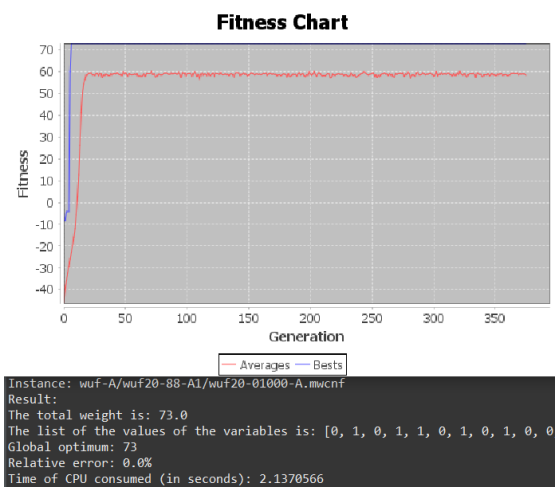
Wuf-A instances

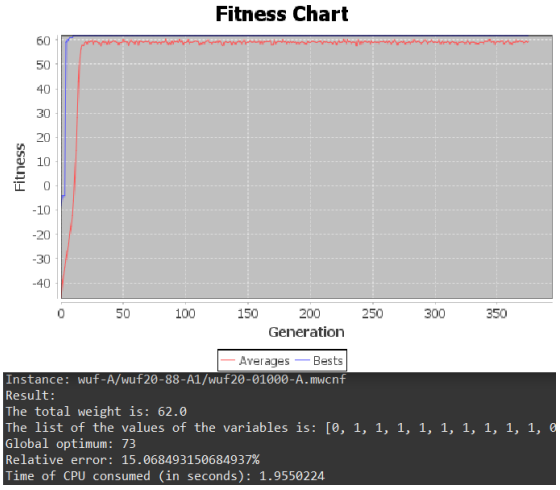
I have tried running the A instance with 20 variables and 88 clauses with **the initial configuration** and again it has **not** been **effective** for these instances.




```
Instance: wuf-A/wuf20-88-A1/wuf20-01000-A.mwcnf
Result:
The total weight is: 62.0
The list of the values of the variables is: [0, 1, 1, 1, 1, 1, 1, 1, 1, 0
Global optimum: 73
Relative error: 15.068493150684937%
Time of CPU consumed (in seconds): 0.3964016
```

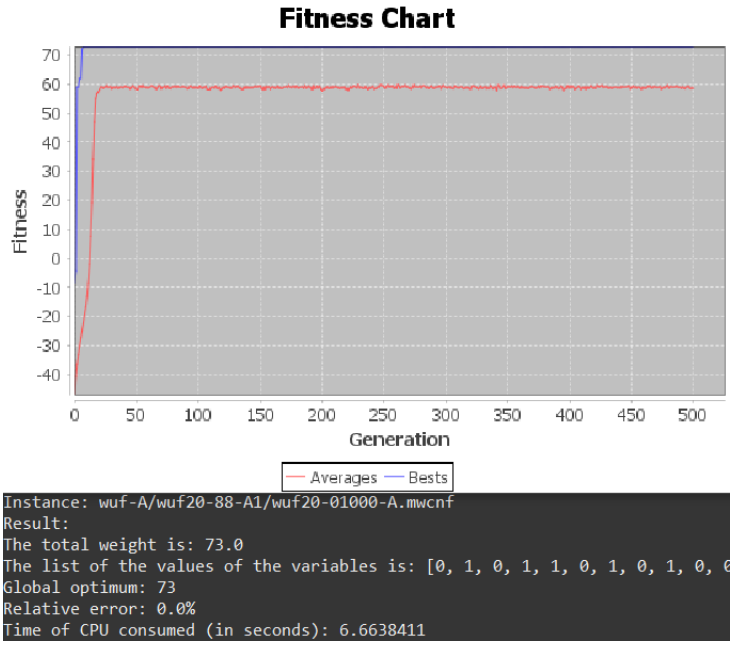
Later I tried to **run** the same instance but with the **optimal configuration** that I obtained in the wuf-M and wuf-N instances to see if they would work for this case.





And the results have always been these two with a higher percentage of appearance the second. This is because in these instances the greatest difficulty is **the limited number of solutions** they have. Then it is normal that, for example, for this configuration the same solutions always appear.

To always try to obtain the optimal solution, I have been increasing the **population and the number of generations up to 2000 and 500**. And the vast majority of times it returns the optimal solution.



CONCLUSIONS

In this study, I have analyzed a genetic algorithm applied to a **MAX-SAT problem** instance. This are the conclusions obtained:

The heuristic has been able to obtain the **majority of the optimal values** of the instances that have been used thanks to adjusting the configuration of the heuristic based on the results that we have been obtaining.

The only instances that have **failed** to obtain the optimum have been the **wuf-Q** instances with **50 variables and 201 clauses**. I have tried to modify the configuration in every possible way to try to reach the optimal one but it has not worked. Even with a purely random search with an extremely large population and number of generations.

Once we **established the optimal parameters** of crossover rate, mutation rate, elitism rate, tournament arity for the wuf-A instance, **these have served us** for the **other instances** without the need to modify them:

-Tournament arity: 2 -Crossover rate: 85%

-Mutation rate: 5% -Elitism rate: 1/populationSize

Simply the **best solution** when the optimal solution was far away was to **increase** the size of the **population** and the **number of generations**. I also observed that better results were obtained (taking into account not excessively increasing performance) if a proportion of 3/1, 4/1 or 5/1 was maintained in the relationship between population and the number of generations.