HOMEWORK 4 – SOLVING THE KNAPSACK PROBLEM BY GENETIC ALGORITHM

BY ANDRÉS RUBIO RAMOS

WHY GENETIC ALGORITHMS?

I have chosen to do homework 4 and 5 on genetic algorithms. The reason is that I have already worked with them and 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.

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:

DataKnapsack class

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

```java
public class DataKnapsack {
    public static record Item(Integer index,
            Integer weight, Integer cost) {
    }
    public static Integer id;
    public static Integer n;
    public static Integer M;
    public static List<Item> items;

    public static void iniData(String line) {
        String[] v=line.split(" ");
        id=Integer.valueOf(v[0].trim());
        n=Integer.valueOf(v[1].trim());
        M=Integer.valueOf(v[2].trim());
        items=List2.empty();
        Integer max=3+(n*2);
        for (int i=3;i<max;i=i+2) {
            Item item=new Item((i-3)/2, Integer.valueOf(v[i]),
                    Integer.valueOf(v[i+1]));
            items.add(item);
        }
    }
    public static Integer getId() {
        return id;
    }
    public static Integer getN() {
        return n;
    }
    public static Integer getM() {
        return M;
    }
    public static Integer index(Integer i) {
        return items.get(i).index();
    }
    public static Integer weight(Integer i) {
        return items.get(i).weight();
    }
    public static Integer cost(Integer i) {
        return items.get(i).cost();
    }
}
```

GeneticKnapsack 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 item i is considered added. On the other hand, if a 0 appears, then item i is not considered added.

Chromosome size is the total number of items and fitness is the sum of the costs of the items that have been selected to be placed in the knapsack. And the strategy is based on maximizing said value.

One thing to take into account is the chromosomes that represent an invalid configuration for our solution, as would be the case of those where the sum of weights exceeds the capacity of the knapsack. To solve this problem I have considered not adding the last items for which if they are added the capacity of the knapsack is exceeded. This assumption only affects the calculation of fitness and its real representation (phenotype) which I will talk about in the next class. For example, for a knapsack with 4 items, if a chromosome is 1011 and the 1st and 3rd items fit but not the 4th, then said chromosome will translate in fitness and representation to chromosome 1010.

I could also have opted for a relaxation strategy, but this way the evolution of average fitness will be better seen in the graphs and subsequent studies.

```
public class GeneticKnapsack implements BinaryData<SolutionKnapsack> {
    public static GeneticKnapsack create(String line) {
        return new GeneticKnapsack(line);
    }
    private GeneticKnapsack(String line) {
        DataKnapsack.iniData(line);
    }
    public Integer size() {
        return DataKnapsack.getN();
    }
    public Double fitnessFunction(List<Integer> value) {
        double fitness=0.;
        Integer capacity=DataKnapsack.getM();
        for (Integer i=0;i<value.size();i++) {
            if (value.get(i)==1 && DataKnapsack.weight(i)<=capacity) {
                fitness+=DataKnapsack.cost(i);
                capacity-=DataKnapsack.weight(i);
            }
        }
        return fitness;
    }
    public SolutionKnapsack solucion(List<Integer> value) {
        return SolutionKnapsack.create(value);
    }

}
```

SolutionKnapsack 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 item is not in the knapsack, 2nd item is, etc, ...". And I have also added that the fitness (the total cost) is shown.

One thing to keep in mind as I already said when calculating the fitness of a chromosome is to treat only valid configurations. For that I have chosen, as in fitness, to consider as not added the last items that no longer fit in the knapsack (it is in the "if" of the "for"). This way we only work with valid configurations, optimizing the algorithm.

```
public class SolutionKnapsack {
    private List<Integer> items;
    private Integer totalCost;
    private SolutionKnapsack(List<Integer> value) {
        totalCost=0;
        items=List2.empty();
        Integer capacity=DataKnapsack.getM();
        for (Integer i=0;i<value.size();i++) {
            if (value.get(i)==1 && DataKnapsack.weight(i)<=capacity) {
                items.add(1);
                totalCost+=DataKnapsack.cost(i);
                capacity-=DataKnapsack.weight(i);
            } else items.add(0);
        }
    }
    public static SolutionKnapsack create(List<Integer> value) {
        return new SolutionKnapsack(value);
    }
    public String toString() {
        return "Result:\nThe total cost is: "
                +totalCost+"\nThe list of items selected is: "
                +items.toString();
    }
}
```

TestKnapsack class

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

"read":

Reads the file and measures the execution time.

"test_gen":

Sets the genetic parameters and runs the algorithm.

```java
public class TestKnapsack {
    public static void main(String[] args) {
        read("files/custom/custom100.dat", 1);
    }
    private static void read(String path, Integer numInstances) {
        try {
            FileReader archivoReader = new FileReader(path);
            BufferedReader bufferedReader = new BufferedReader(archivoReader);
            Integer i=0;
            String line;
            long startTime = System.nanoTime();
            while (((line = bufferedReader.readLine()) != null)&&i<numInstances) {
                test_Gen(line);
                i++;
            }
            bufferedReader.close();
            long endTime = System.nanoTime();
            long elapsedTimeInNanoseconds = endTime - startTime;
            double elapsedTimeInSeconds = (double) elapsedTimeInNanoseconds / 1e9;
            System.out.println("Time of CPU consumed (in seconds): "+elapsedTimeInSeconds);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    private static void test_Gen(String data) {
        AlgoritmoAG.POPULATION_SIZE=30; //Default: 30
        ChromosomeFactory.TOURNAMENT_ARITY=2; //Default: 2
        ChromosomeFactory.crossoverType=CrossoverType.OnePoint; //Default: OnePoint
        AlgoritmoAG.CROSSOVER_RATE=0.8; //Default: 0.8
        AlgoritmoAG.MUTATION_RATE=0.6; //Default: 0.6
        AlgoritmoAG.ELITISM_RATE=0.2; //Default: 0.2
        StoppingConditionFactory.stoppingConditionType=
                StoppingConditionType.GenerationCount; //Default: GenerationCount
        StoppingConditionFactory.NUM_GENERATIONS = 80; //Default: Integer.MAX_VALUE

        GeneticKnapsack problem= GeneticKnapsack.create(data);
        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()/118706))*100.0;
        System.out.println("Relative error: "+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

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 I am going to work on an instance that I have generated with the instance generator with the following properties:

-Num Items: 100 -Maximum weight: 250

-Maximum cost: 2500 -Ratio capacity/totalWeight: 0.8

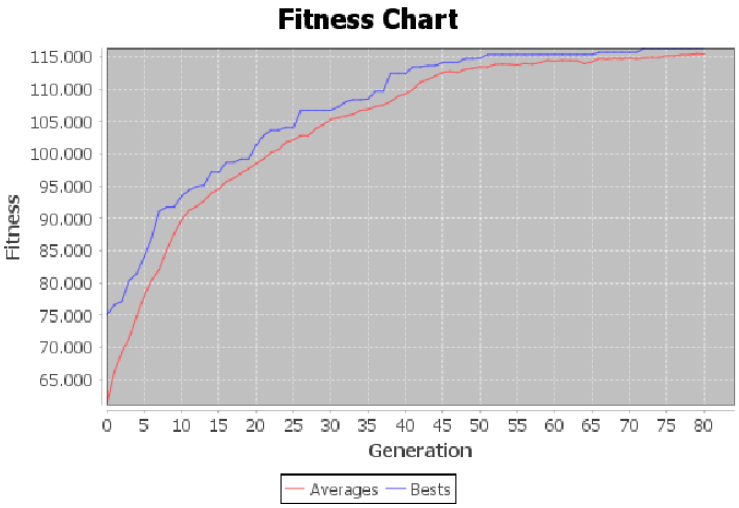
I have used the dynamic programming algorithm to obtain the global optimal which is 118706. With this value in mind we can see how close our approximation given by the algorithm will be.

The parameters of the base example will be the following:

```
AlgoritmoAG.POPULATION_SIZE=30; //Default: 30
ChromosomeFactory.TOURNAMENT_ARITY=2; //Default: 2
ChromosomeFactory.crossoverType=CrossoverType.OnePoint; //Default: OnePoint
AlgoritmoAG.CROSSOVER_RATE=0.8; //Default: 0.8
AlgoritmoAG.MUTATION_RATE=0.6; //Default: 0.6
AlgoritmoAG.ELITISM_RATE=0.2; //Default: 0.2
StoppingConditionFactory.stoppingConditionType=
        StoppingConditionType.GenerationCount; //Default: GenerationCount
StoppingConditionFactory.NUM_GENERATIONS = 80; //Default: Integer.MAX_VALUE
```

I have chosen the base parameters that appeared in the Apache code. Everything seems fine except for the Mutation and elitism rate, which seem very high.

This is the result of the execution. It took 0.72 seconds. It has obtained 116341 fitness, which generates 1.99% relative error.



Now we are going to start with the modifications of the parameters to see if the performance and results obtained improve or worsen.

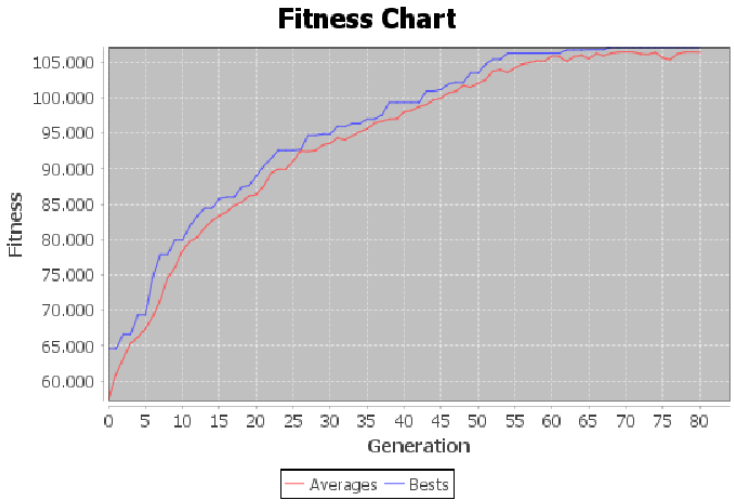
Population size (default 30)

-For 10:

CPU time: 0.709 ⬇

Fitness: 107125 ⬇

Rel. error: 9.75% ⬆

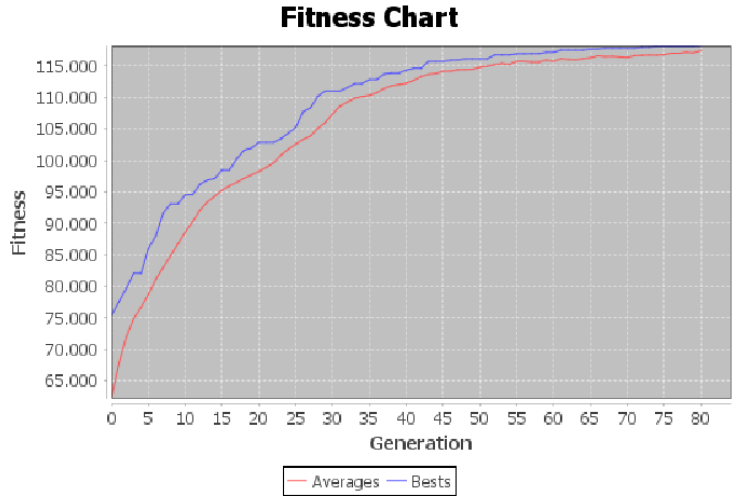


-For 50:

CPU time: 0.73 ⬆

Fitness: 118199 ⬆

Rel. error: 0.42% ⬇

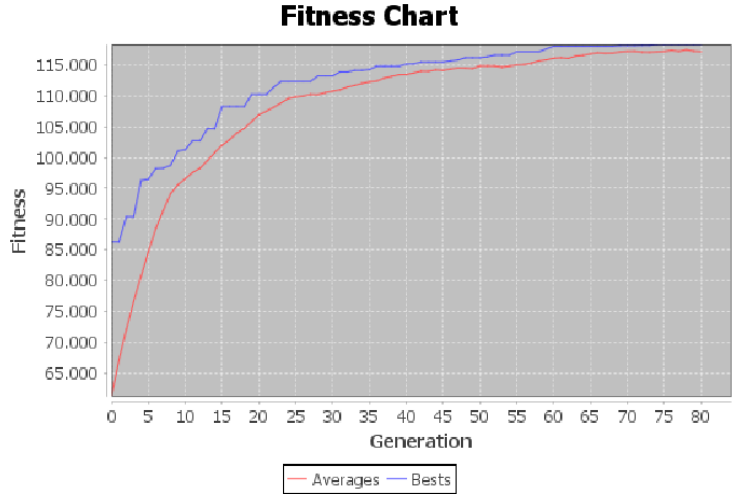


-For 100:

CPU time: 0.76 ⬆

Fitness: 118405 ⬆

Rel. error: 0.25% ⬇



We can see that increasing the population improves the results considerably without affecting the execution time to a large extent. While decreasing negatively affects the result as expected.

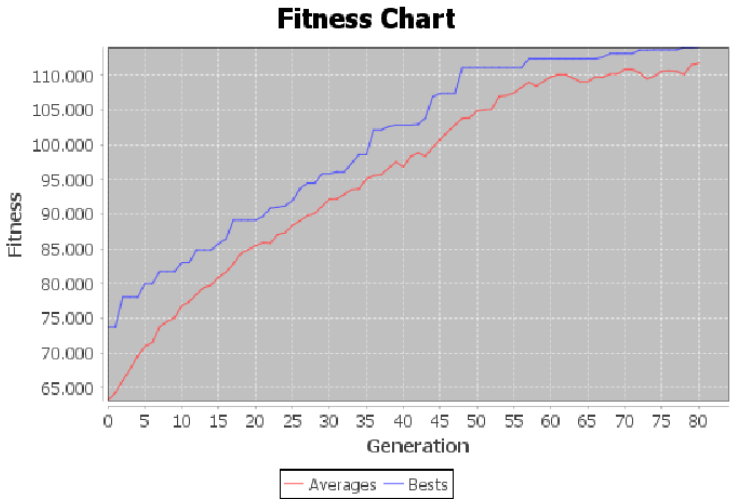
Tournament arity (default 2)

-For 1:

CPU time: 0.69 ⬇

Fitness: 114076 ⬇

Rel. error: 3.90% ⬆

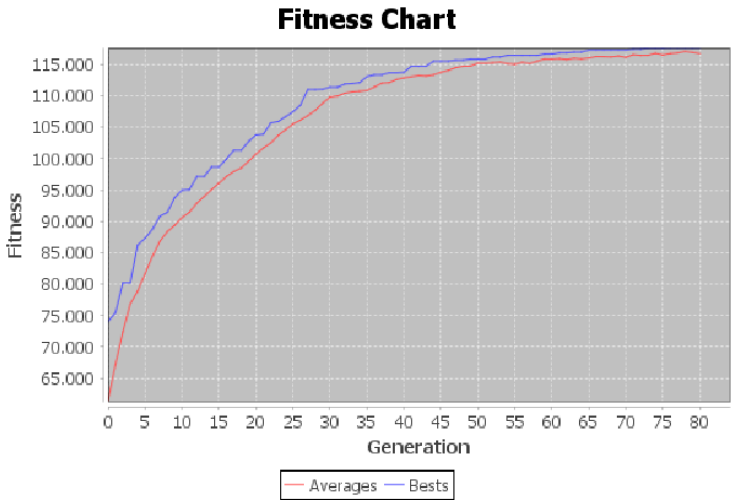


-For 3:

CPU time: 0.72 ⬆

Fitness: 117643 ⬆

Rel. error: 0.89% ⬇

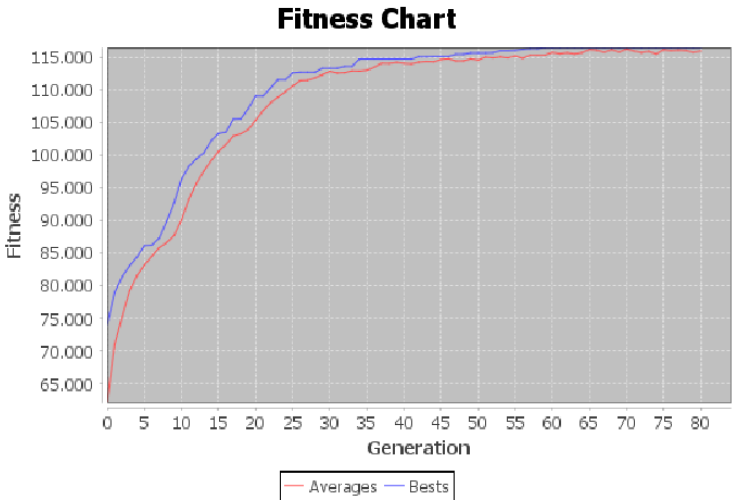


-For 5:

CPU time: 0.72 ⬆

Fitness: 116980 ⬆

Rel. error: 1.49% ⬇



Arity 1 is the worst by far since it is choosing random. And too much arity favors the chromosomes with the highest fitness, which are already represented with the high elitism 6/30. So the best values are 2 or 3.

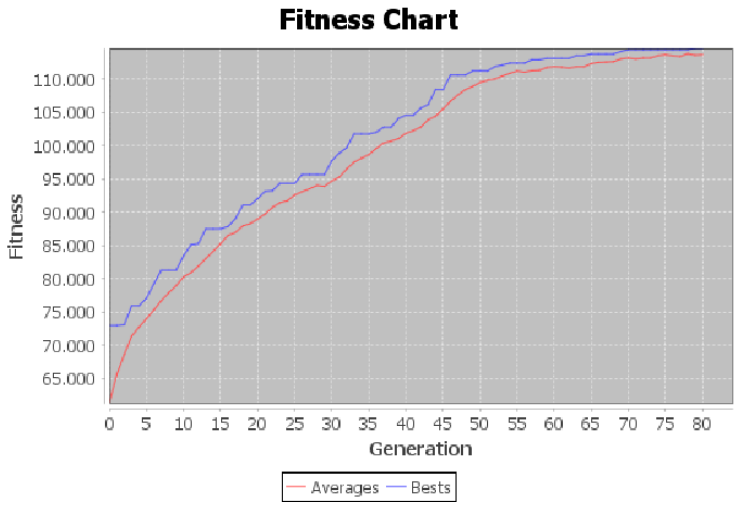
Crossover rate (default 0.8)

-For 0.2:

CPU time: 0.61 ⬇

Fitness: 114662 ⬇

Rel. error: 3.40% ⬆

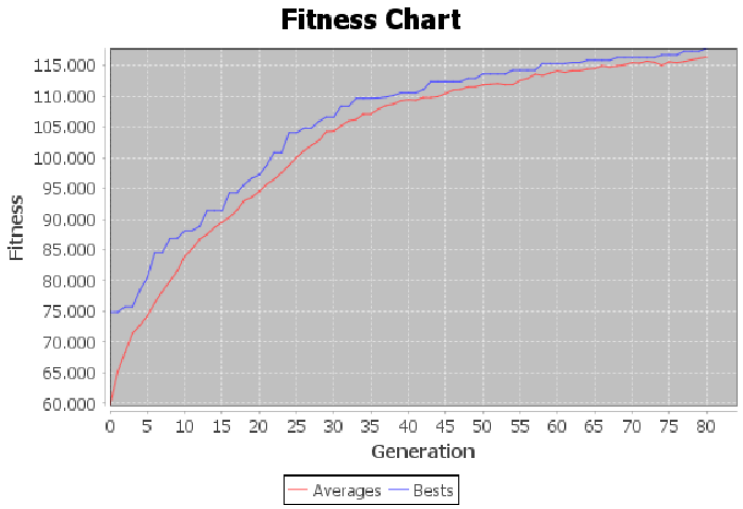


-For 0.5:

CPU time: 0.64 ⬇

Fitness: 117791 ⬆

Rel. error: 0.77% ⬇

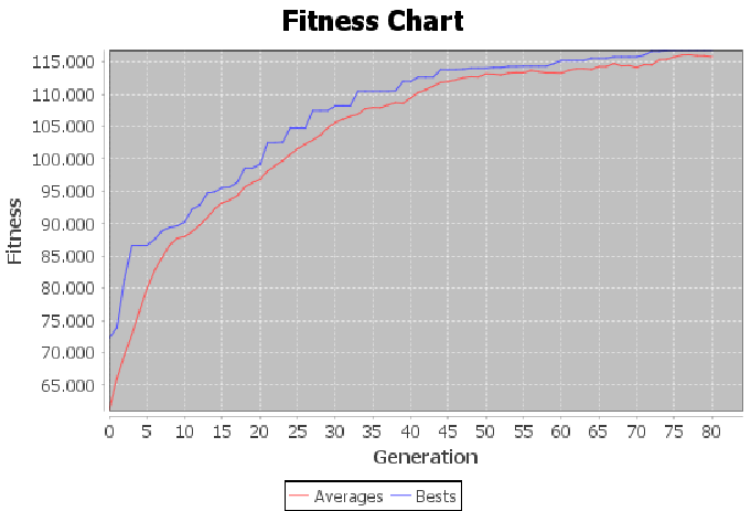


-For 0.9:

CPU time: 0.65 ⬇

Fitness: 116861 ⬆

Rel. error: 1.55% ⬇



A crossover rate of 0.5 strikes a balance between exploration and exploitation, fostering genetic diversity and preventing premature convergence. Higher rates, such as 0.8 or 0.9, may lead to quicker convergence and reduced diversity, potentially hindering the algorithm's ability to find optimal solutions.

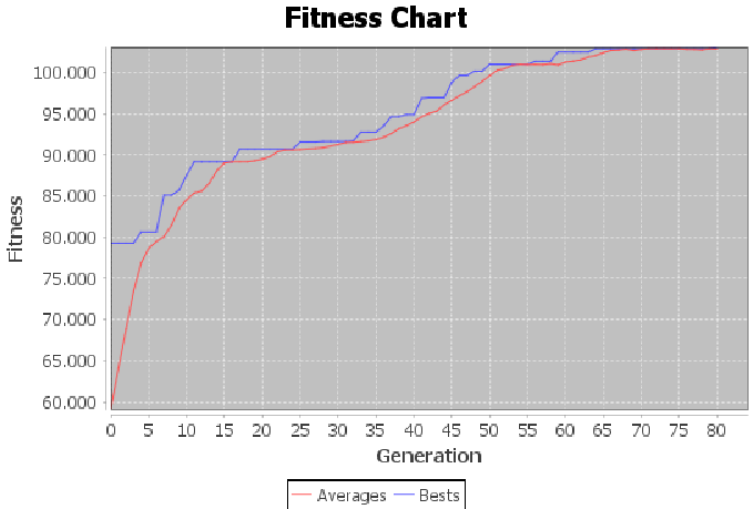
Mutation rate (default 0.6)

-For 0.05:

CPU time: 0.73 ⬆

Fitness: 103083 ⬇

Rel. error: 13.16% ⬆

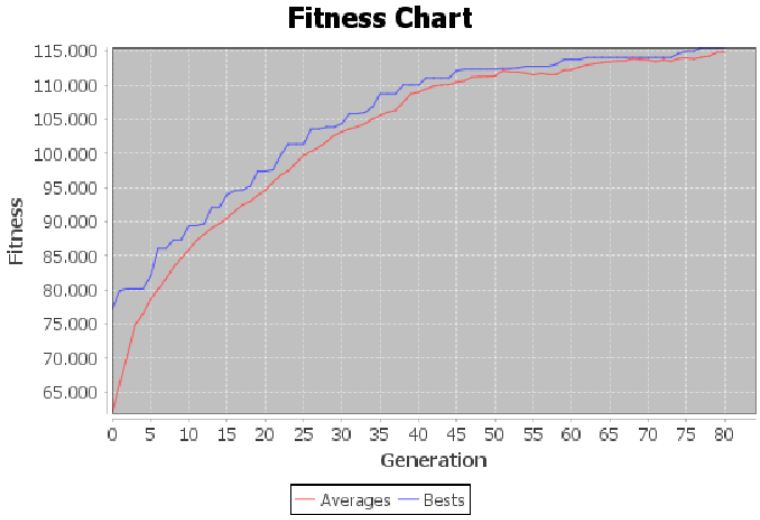


-For 0.3:

CPU time: 0.75 ⬆

Fitness: 110594 ⬇

Rel. error: 6.83% ⬆

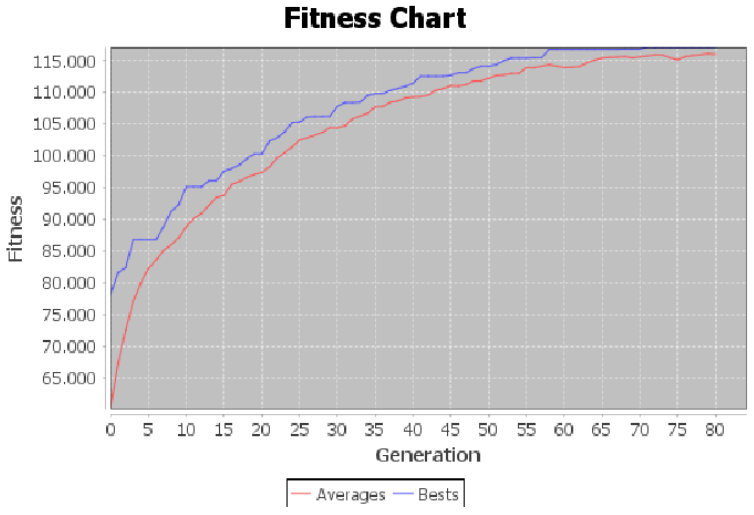


-For 0.8:

CPU time: 0.76 ⬆

Fitness: 117121 ⬆

Rel. error: 1.33% ⬇



A higher mutation rate can lead to better results as it introduces greater diversity in the genetic population, increasing the likelihood of exploring a wider search space. This can help the algorithm escape local optima and discover more diverse solutions, potentially improving convergence towards a globally optimal solution. However, an excessively high mutation rate may lead to too much randomness, affecting the algorithm's stability and convergence.

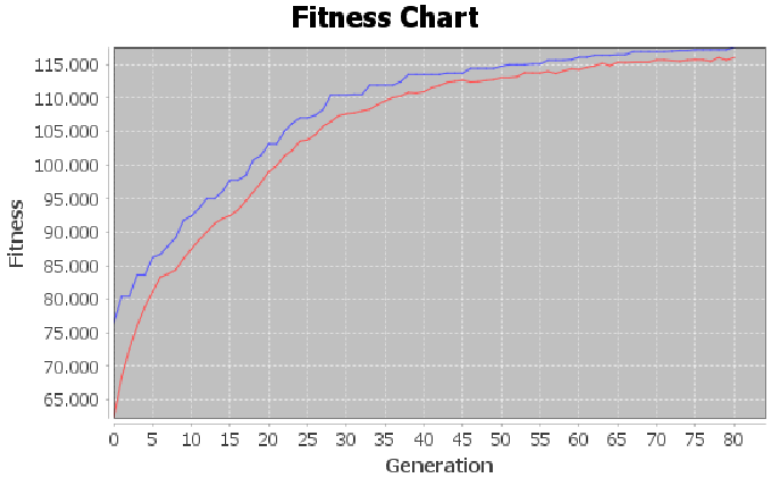
Elitism rate (default 0.2)

-For 0.1 (3/30):

CPU time: 0.72 ⬇

Fitness: 117610 ⬆

Rel. error: 0.92% ⬇

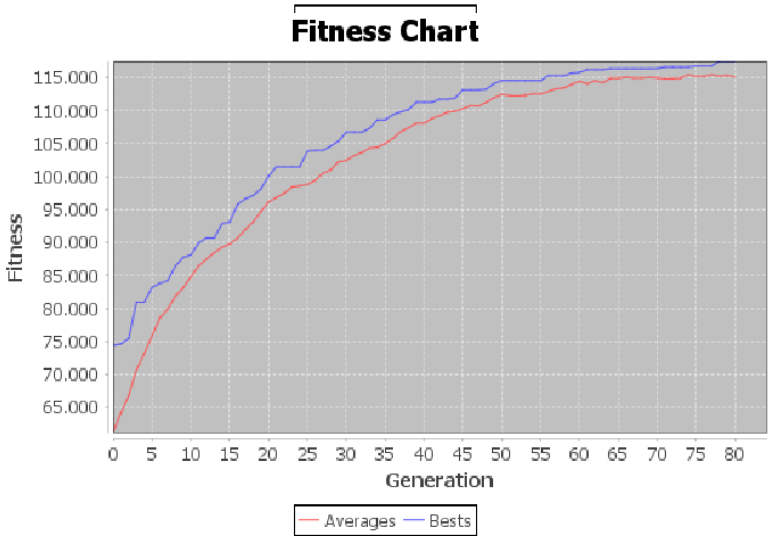


-For 0.34 (1/30):

CPU time: 0.72 ⬇

Fitness: 117497 ⬆

Rel. error: 1.01% ⬇

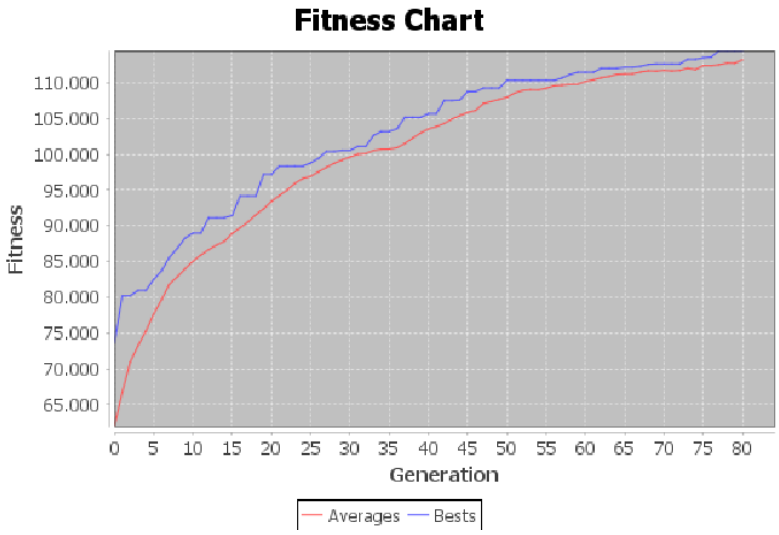


-For 0.5 (15/30):

CPU time: 0.67 ⬇

Fitness: 114515 ⬇

Rel. error: 3.53% ⬆



A higher elitism rate in genetic algorithms may lead to worse results because it increases the likelihood of preserving suboptimal individuals, hindering the exploration of the solution space. A higher elitism rate prioritizes the retention of the best individuals from one generation to the next, potentially preventing the algorithm from escaping local optima and finding more globally optimal solutions.

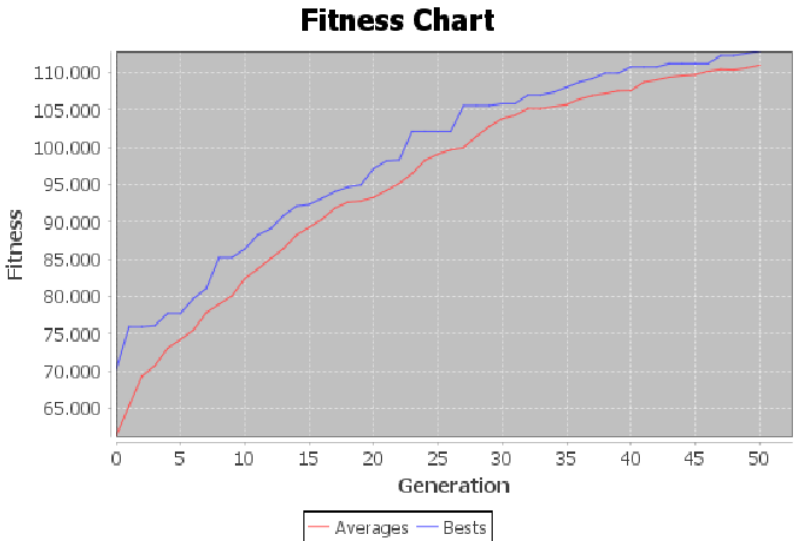
Num generations (default 80)

-For 50:

CPU time: 0.58 ⬇

Fitness: 112854 ⬇

Rel. error: 4.92% ⬆

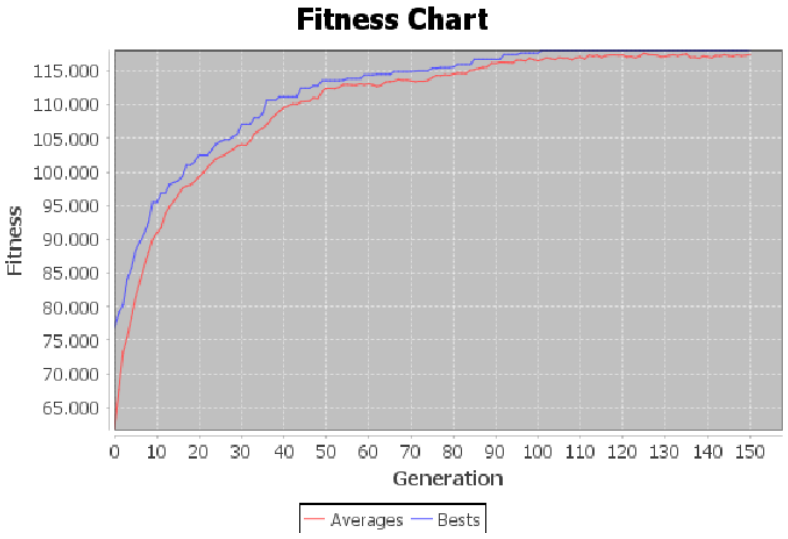


-For 150:

CPU time: 0.89 ⬆

Fitness: 118111 ⬆

Rel. error: 0.50% ⬇

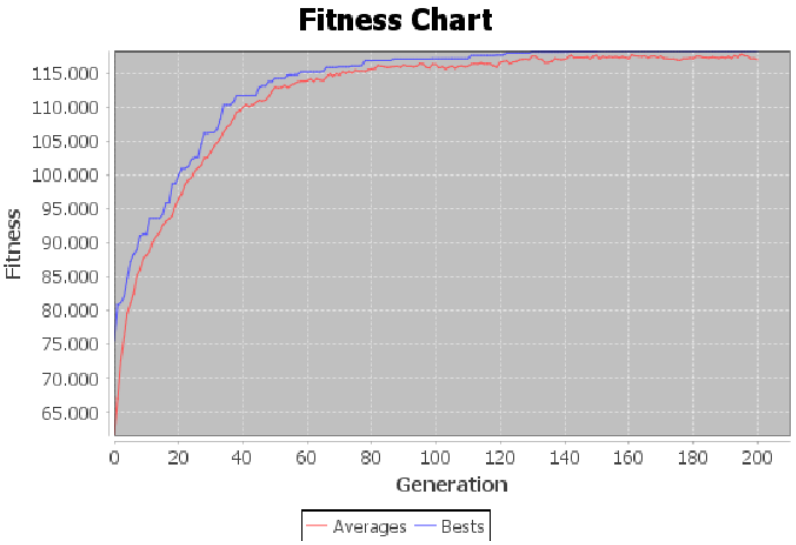


-For 200:

CPU time: 1.01 ⬆

Fitness: 118314 ⬆

Rel. error: 0.33% ⬇



A higher number of generations in a genetic algorithm allows for more iterations of the evolutionary process, increasing the likelihood of exploring a broader solution space. Although with a value that is too high, fitness can stagnate due to the low diversity of the population.

Looking for the best configuration

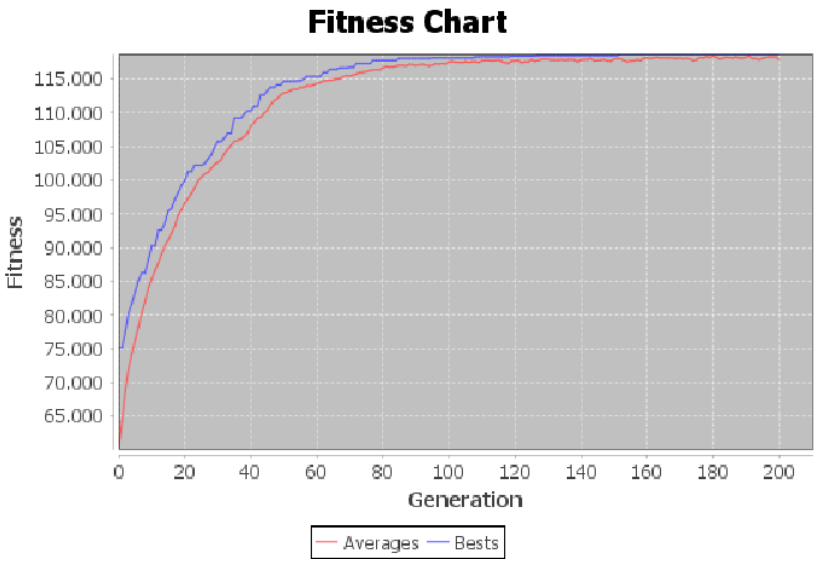
With everything learned in the previous tests, now I am going to try to modify all the values at the same time to find the best configuration that tries to find the optimal solution.

```
AlgoritmoAG.POPULATION_SIZE=50; //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=0.6; //Default: 0.6
AlgoritmoAG.ELITISM_RATE=0.34; //Default: 0.2
StoppingConditionFactory.stoppingConditionType=
        StoppingConditionType.GenerationCount; //Default: GenerationCount
StoppingConditionFactory.NUM_GENERATIONS = 200; //Default: Integer.MAX_VALUE
```

CPU time: 1.04

Fitness: 118705

Rel. error: 0.0008%



Let us remember that the fitness of the optimal solution is 118706, that is, 118705 is almost the global optimum. So we can conclude that this configuration solves the problem quite well. I have also run it several times to check if it was a coincidence or not and most of the time it reaches this solution.

Scalability with the instance size

The previous instance was 100 items, now I am going to test the previous optimal configuration to see how it goes with instances of 125 and 150 items. In both instances I have only changed the number of items and I have searched for the global optimal solution with the dynamic programming algorithm.

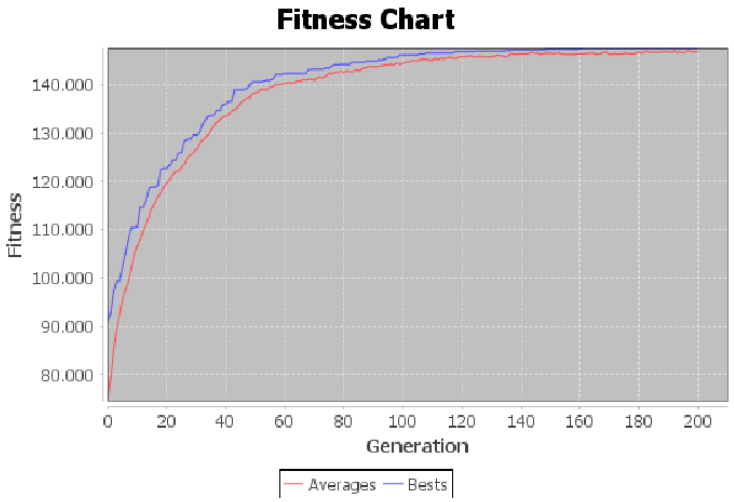
125 items instance:

-Same configuration as with 100 items:

Fitness: 147700

Optimum: 148806

Rel. error: 0.74%



It has increased as we expected and the error is no longer approaching 0 as it previously did.

Now I'm going to change the algorithm configuration to try to improve the solution:

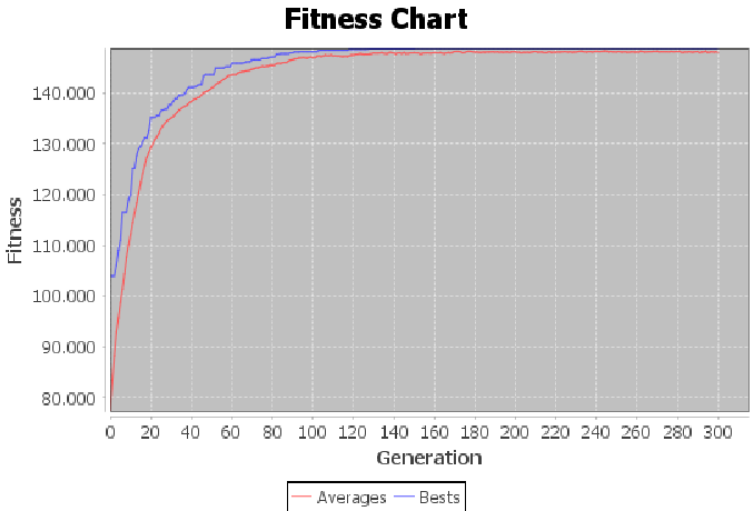
-Population size: 50 -> 200

-Num generations: 200 -> 300

CPU time: 2.2

Fitness: 148806

Rel. error: 0.0%



I have run it several times and the relative errors were always between 0.0-0.05%.

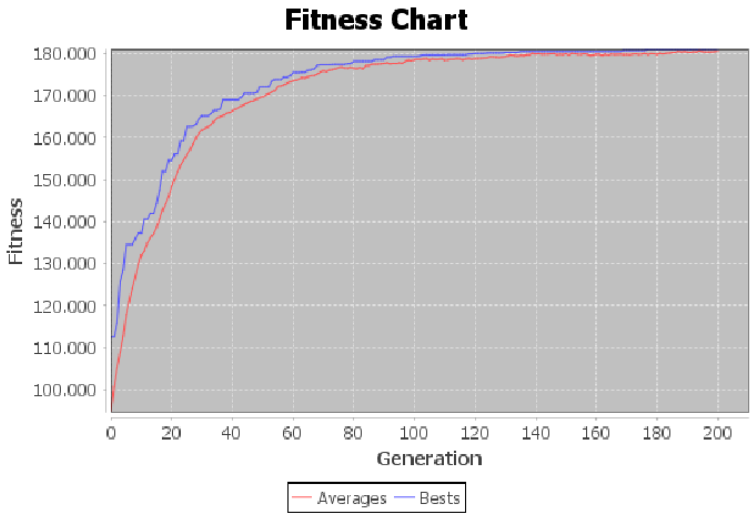
150 items instance:

-Same configuration as with 100 items:

Fitness: 181058

Optimum: 182856

Rel. error: 0.98%



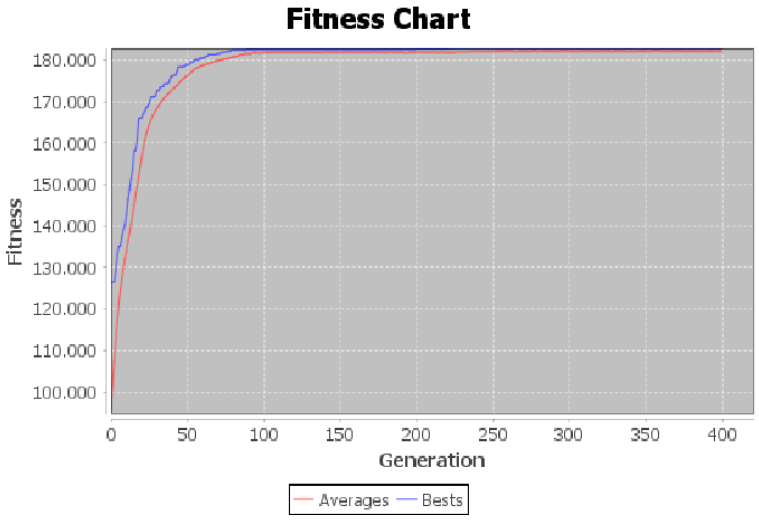
It has increased even more as we expected and the error is no longer approaching 0 as it previously did.

The new changes: - Population size: 50 -> 400 -Num generations: 200 -> 400

CPU time: 3.4

Fitness: 182856

Rel. error: 0.0%



I have run it several times and the relative errors were always between 0.0-0.07%.

With these results we can see that by simply modifying the population size and the number of generations we can adapt the algorithm to larger instances and continue to be effective while increasing the CPU execution time.

CONCLUSIONS

In this study, I have analyzed a genetic algorithm applied to an knapsack problem instance. This are the conclusions obtained:

Increasing the population size significantly enhances results without a substantial impact on execution time, while decreasing size adversely affects outcomes.

Arity 1 performs poorly due to random selection, and higher arity (especially with elitism 6/30) is deemed optimal.

A crossover rate of 0.5 strikes a balance between exploration and exploitation, but higher rates risk premature convergence.

A higher mutation rate enhances diversity, aiding global search, but excessive rates can introduce instability.

Elevated elitism rates hinder solution space exploration by preserving suboptimal individuals. While more generations enable broader exploration, excessively high values may lead to fitness stagnation.

Adjusting population size and generations effectively tailors the algorithm to larger instances, influencing CPU execution time.