Multiple Populations in Evolutionary Computation

Francisco Guerra$^{1[20172653882]}$, Ricardo Martins$^{1[2017246268]}$, and Tiago Fernandes$^{1[2017242428]}$

Departamento de Engenharia Informática Faculdade de Ciências e Tecnologia
Universidade de Coimbra Pólo II - Pinhal de Marrocos, Coimbra

Abstract. In this paper, we will study the influence of using multiple populations in two benchmark problems that have a binary representation, João Brandão's Numbers and Knapsack. We will evolve two populations individually and at a given time, we will use one of three different algorithms in order to change individuals between the populations, and see how it changes the evolution in the end.

Keywords: Multiple Populations · Evolutionary Algorithm · Random Immigrants · João Brandão's numbers · Binary Knapsack.

1 Introduction

The most common evolutionary algorithm follows the theory about the evolution of species proposed by Darwin. He defends that the individuals that fit better in the environment will survive longer, and their abilities will be passed to the new generation (crossover). The explanation for appearing new characteristics in the population is called a mutation. That change can be better and the individual can adapt better than the others or can be worse because we lose some ability that his parents have. For an evolutionary algorithm these are the biggest steps: find the fittest, a crossover between them to generate the newest population, and mutation in individuals to generate new characteristics to try to appear new aptitudes to the environment.

Like in the real world, multiple groups of people try to achieve the same objective and that's the main goal of this paper. We will study the influence of different populations with the same goal, on the same environment. That influence will be provoked in three different ways: swap a number of elements with the other population; insert elements from a non evolved population, outsiders of the populations that are evolving towards the same goal (random immigrants); and choosing the individual with the lowest fitness value and swapping them with the most different individual of the other population (based on the Hamming Distance).

In the next section, we will describe better our benchmark problems, and all the methods that we use that we refer to in the first paragraph of the introduction. In the third section we will talk about our goal and results of our experiment and we will finish on the fourth section with the conclusion of that work.

2 Implementation

2.1 Benchmark Problems

On our problem, we used two benchmark problems:

– **"João Brandão's numbers"**: in this problem, given a subset of the integers between 1 and n (maximum size) and no element in the subset is the average of other two in the subset.
– **"0 - 1 Knapsack"**: in this problem, given a knapsack and a set on n items, each with a certain weight and value, we need to maximize the value without surpassing the max capacity of the knapsack.

2.2 Fitness

Each benchmark a specific fitness function:

– **"João Brandão's numbers"**: the fitness function, that we use in this problem, checks for all the numbers in the set and sees if the previous and the next numbers of the number that is being checked exists in the set. If that condition is true, we aren't following the rules of the problem and that solution will be penalized.
– **"0 - 1 Knapsack"**: for this benchmark problem, our fitness function basically calculates the total value of the items in the knapsack, but if exceeds the max capacity, the fitness value is zero.

2.3 Variation Methods

Mutation - The genotype is in binary form so the mutation function that we are using is a XOR of the genotype with 1 if a random number generated is lower than a given probability.

Crossover - We have 3 types of crossover functions (Figure 1):

– **One point crossover**: if a random number generated between 0 and 1 is lower than a given probability, then we generate 2 offspring's such as one of them as the first half of the parent 1 and the second half of the parent 2;
– **Two point crossover**: if a random number generated between 0 and 1 is lower than a given probability, then we generate 2 offspring's such as one of them as the first and last third of the parent 1 and the second third of the parent 2 and the other offspring has the first and last third of the parent 2 and the second third of the parent 1;
– **Uniform crossover**: if a random number generated between 0 and 1 is lower than a given probability, then we generate n random numbers (size of the genotype) between 0 and 1 and if the number is lower than 0.5, we get the index of the parent 1, otherwise, we get the index of the parent 2, generating 2 offsprings in total.

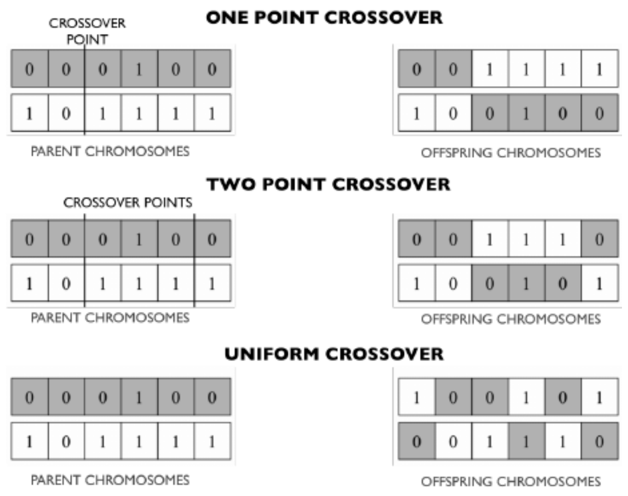


Fig. 1: 3 types of crossover functions

2.4 Migration between Populations

To start, we used the code given by the teachers on the third practical class. In that code, we started to make changes, such as adding a second population and it does exactly the same as the first population. After each generation ends, we see if we swap anything between populations (given a probability) and if we do, we have 4 methods to test:

- **Method 0 -** We don't swap any individuals between the populations.
- **Method 1 -** We start to select a number of individuals (n) to switch between populations and then we swap the last n individuals of each population by each other.
- **Method 2 -** We start to select a number of individuals (n) and we generate a new population with n individuals. After, we swap the last n individuals of the original population by that non-evolved populated.
- **Method 3 -** We start to select a number of individuals (n) and select the n individuals of one of the populations with the lowest value of fitness. Next, we'll check for each individual selected previously, the one on the other population with the highest value of Hamming Distance (i.e. the most different) and we copy that one to the first population, without making any changes to the second population.

3 Experimental Method and Results

To evaluate this problem we opted to use two different benchmark problems (João Brandão's Numbers and Knapsack). For each benchmark problem we look further into the behaviour of the implemented methods and their performance.

To perform the statistical analysis we use the code provided in the practical classes.

3.1 João Brandão's Numbers

Given the nature of the benchmark problem and its time complexity, we decided to set the number of generations to 150 and the size of the chromosome to 100. By setting the size of the chromosome to 100 we limit the fitness to a maximum of 27. This will have influence on the results since the search space is small.

Initial Parameters

To find the best parameters for the base algorithm we did a small scale experiment considering only six runs for each configuration. For this experiment we still evolve two populations, however we don't consider migration in the populations, the only goal is to decide the best options and values for the variation operators.

Both populations use the same mutation operator and the same fitness function. For the evolution of the first population in each immigration method we used the following base algorithm:

- Mutation probability: 0.01
- Crossover probability: 0.9
- Crossover operator: two point crossover

For the second population we change those parameters to:

- Mutation probability: 0.05
- Crossover probability: 0.5
- Crossover operator: uniform crossover

To evaluate the performance of each set of parameters we considered the average of the maximum fitness between the two populations over the five runs considered.

Comparing different immigrant algorithms

Here we will evaluate which method, between the three, produces the best individuals. To evaluate each method we consider the best individual found in either of the populations. This measure is used instead of the average between the two populations so that method 1 doesn't get an advantage.

The selected benchmark problem has a very limited search space, therefore we expect that all the variations tested for the problem will reach similar results. So for this test, we defined the null hypothesis (H0) as, *"All methods will get the same results"*, and the alternative hypothesis is *"There are differences in using* *the different methods"*. For all steps of the test, we will use a confidence level of 0.05.

Before doing the statistical tests we looked into how the data was distributed for each method (Figure 2). In Figure 2 we can see that, even though method 2 reaches higher fitness all seem to follow similar distributions.

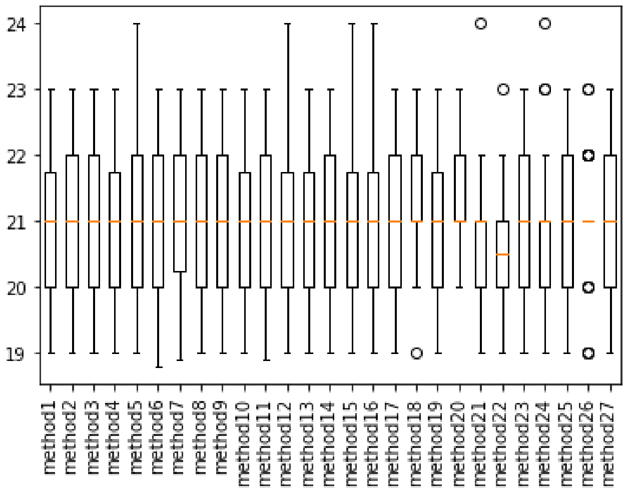


Fig. 2: Performance distribution for each method.

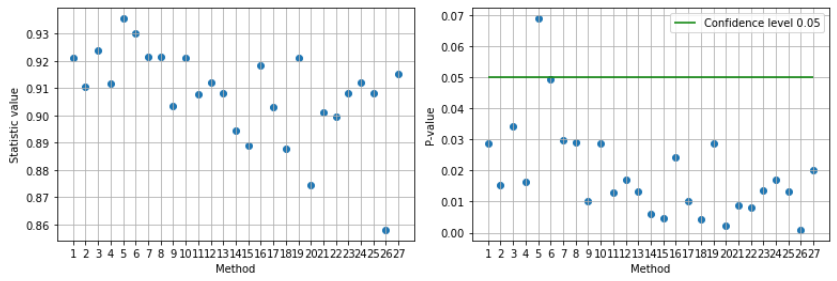
To test this we looked into all the configurations shown in table 1 (last page). We used 30 runs for each different configuration. After collecting the data we will compare all the different algorithms against each other.

First, we find if the data from the methods follow a normal distribution using Shapiro-Wilk test. The results are shown in figure 3.

Looking at figure 3b, we can see that most p-values are under 0.05. Therefore we reject the null hypothesis, and we can't assume a normal distribution. Since the data doesn't follow a normal distribution we need to use a non-parametric test. We are comparing more than two samples so we used Friedman's test. With the test, we obtained a p-value of 0.48222. Since the value for the p-value is higher than our confidence level we don't reject the null hypothesis. Therefore, the data of the difference seems to follow the same distribution, which means we don't reject the null hypothesis.

3.2 Uncorrelated Knapsack Problem

This problem has three popular formulations that depend on how correlated the data is. For our problem, we only looked at the performance of the different algorithms in the uncorrelated knapsack problem. This means that the values



(a) Statistics values.

(b) P-values.

Fig. 3: Shapiro-Wilk test for João Brandão's numbers.

and weights were generated using uniform distributions, as is shown in 1 and 2 where w_i is the array with the items weights and v_i is the array with the matching item value, v is the maximum for the random generator. In this experiment, we are using 100 items.

$$w_i = uniform([1..v]) \tag{1}$$

$$v_i = uniform([1..v]) \tag{2}$$

Initial Parameters

The hyperparameters for this problem were chosen based on the standard values used in most evolutionary algorithms. Both populations use the same mutation operator and the same fitness function. For the evolution of the first population in each immigration method we used the following base algorithm:

- Mutation probability: 0.01
- Crossover probability: 0.8
- Crossover operator: two point crossover

For the second population we change those parameters to:

- Mutation probability: 0.05
- Crossover probability: 0.6
- Crossover operator: uniform crossover

For each population evolution algorithm, we use a population size of 10 and 200 generations.

Comparing different immigrant algorithms

Like in the previous we will evaluate which method, between the three, produces the best individuals. As a measure of performance, we use the fitness of the best individual found in either of the populations.

Considering the problem, we expect that are many solutions closer to the best individual possible. Therefore finding the individuals closer to the "peak" is extremely important. With this in mind, we expect that some methods will find those "peaks" more often than others, meaning that some algorithms will perform better than others.

So for this test, we defined the null hypothesis (H0) as, *"All methods will get the same results"*, and the alternative hypothesis is *"There are differences in using the different methods"*. For all steps of the test, we will use a confidence level of 0.05.

Before doing the statistical tests we looked into how the data was distributed for each method. (Figure 4). Analyzing the plot in figure 4 it looks appears to show what we expected, some methods seem worse than the others. For example method 26 seems to clearly underperform when compared to other algorithms.

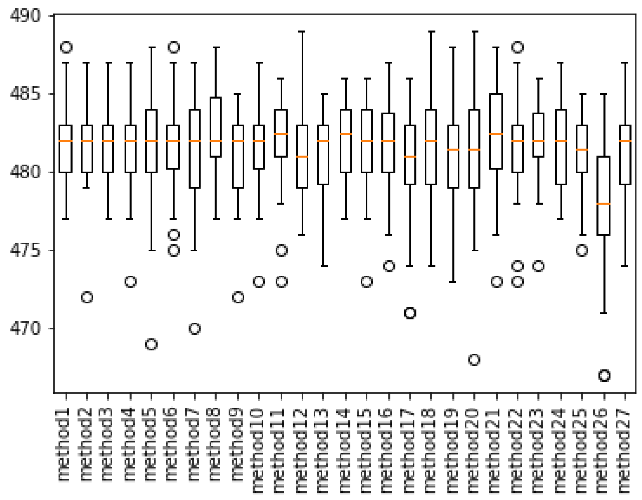
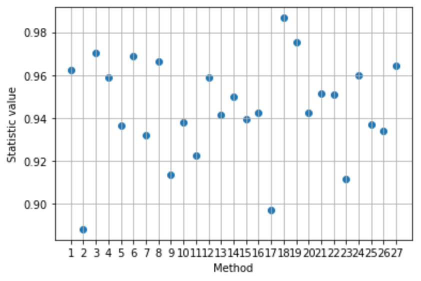
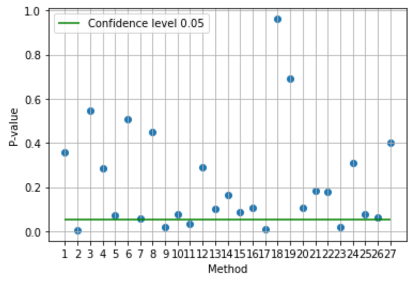


Fig. 4: Performance distribution for each algorithm knapsack problem.

Now that the hypothesis is defined, we can compare all algorithms tested (shown in table 1 - last page of the report). The first step is to look at the distributions of the data gathered from the algorithms, for this purpose we used the Shapiro-Wilk test with $alpha = 0.05$. If we look at the results shown in figure 5a we can see that some of the values are below the line, this means that for those algorithms $p\_value < alpha$ so we reject the null hypothesis of the Shapiro-Wilk test, which means that those algorithms don't follow a Gaussian distribution.



(a) Statistics values.



(b) P-values.

Fig. 5: Shapiro-Wilk test for uncorrelated numbers.

Since some of the algorithms don't follow a Gaussian distribution we need to use non-parametric tests to make decisions over the hypothesis that was defined earlier. Since we want to compare multiple algorithms at once and for our tests all populations have the same initial populations (dependent samples). Considering the mentioned characteristics of the data we decided to use Friedman's test. With this method we got $pvalue = 0.0741$. Since $pvalue > alpha$ we can't reject the null hypothesis which means that for the tested uncorrelated knapsack problem there are no statistical differences between the algorithms compared.

The fact that all the algorithms have no differences might be related to the difficulty of the problem. Since the problem setup is not extremely hard all algorithms reach solutions close to the best individual which in this case has a fitness of 491.

4 Conclusions

With the goal of comparing two different algorithms that rely on immigrants, we set up two binary problems to compare how each algorithm would perform. For each of the algorithms, different values were tested for the frequency of individuals exchanged and the number of individuals traded.

Using João Brandão's numbers with a size of 100 numbers and the Uncorrelated Knapsack we didn't find statistically significant differences in the performance for the implemented migration algorithms.

If we tried to use another benchmark problem (for example, a function like De Jong or Rosenbrock), the results we would get could be different.

Name	**Frequency**	**% of replacement**	**Replacement Method**
method1	0.1	0.1	Migration by fitness
method2	0.1	0.1	Random Immigrants
method3	0.1	0.1	Migration by variation
method4	0.1	0.25	Migration by fitness
method5	0.1	0.25	Random Immigrants
method6	0.1	0.25	Migration by variation
method7	0.1	0.5	Migration by fitness
method8	0.1	0.5	Random Immigrants
method9	0.1	0.5	Migration by variation
method10	0.5	0.1	Migration by fitness
method11	0.5	0.1	Random Immigrants
method12	0.5	0.1	Migration by variation
method13	0.5	0.25	Migration by fitness
method14	0.5	0.25	Random Immigrants
method15	0.5	0.25	Migration by variation
method16	0.5	0.5	Migration by fitness
method17	0.5	0.5	Random Immigrants
method18	0.5	0.5	Migration by variation
method19	0.9	0.1	Migration by fitness
method20	0.9	0.1	Random Immigrants
method21	0.9	0.1	Migration by variation
method22	0.9	0.25	Migration by fitness
method23	0.9	0.25	Random Immigrants
method24	0.9	0.25	Migration by variation
method25	0.9	0.5	Migration by fitness
method26	0.9	0.5	Random Immigrants
method27	0.9	0.5	Migration by variation

Table 1: Table with tested algorithms