P10

edgardjfc

May 2022

1 Introduction

The base for this code was written based on the work of our teacher E. Schaeffer [3] and classmate J. "FeroxDeitas" [2]. With the entirety of this code being found in the github repository of E. "edgardjfc" [1].

For this assignment the objective was to model a knapsack problem scenario. A knapsack problem consist of putting different particles into a knapsack in the most optimal way, the particles can have different parameters, but in this case they have a value and a weight. For this assignment, the goal was to generate multiple particles through a genetic algorithm and select them to put them into the knapsack, as well as using three different ways to generate the particles, one where both have random normal distribution of value and weight independent from each other, a second where the value and weight generate proportionally inverse from one another and a third where the weight is generated with a normal distribution while the value is squarely proportional to the weight.

2 Code

For each of the three previous cases there were three different variables that changed together.

For the first combination of parameters we had:

```
pm, rep, init = 0.05, 50, 100
```

And for the second combination of parameters we had:

```
pm, rep, init = 0.025, 100, 200
```

Where "pm" is the probability to mutate, "rep" the number of crossings and "init" is the initial population. Other initial conditions are n=100 meaning that 100 values and weights will be created, tmax=150 meaning that 150 steps will be created, and finally iterations=20.

The three different codes were similar in execution, however with the key change in the generators of value and weight. For the first case, where both weight and value are independent and with normal distribution the code is as follows:

```
def weightsGenerator(amount, low, high):
    return np.round(normalize(np.random.uniform
        (low=low, high=high, size = amount)) * (high - low) + low)

def valuesGenerator(weights, low, high):
    return np.round(normalize(np.random.uniform
        (low = low, high = high, size = weights)) * (high - low) + low)
```

For the second case where the weight is inversely proportional to the value, and the value is normally distributed:

```
def weightsGenerator(values, low, high):
    cant = 1 / values
    return np.round(((normalize(cant))) * (high - low) + low)

def valuesGenerator(weights, low, high):
    cant = np.arange(0, weights)
    return np.round(normalize(expon.pdf(cant))  * (high - low) + low)
```

For the third case the value is proportional to the square of the weight under normal distribution:

```
def weightsGenerator(cuantos, low, high):
    return np.round
        (normalize(np.random.normal(size = cuantos)) * (high - low) + low)

def valuesGenerator(pesos, low, high):
    return np.round((pesos**2) * (high - low) + low)
```

Beyond these three different generator functions, the rest of the code runs essentially the same way, calling different functions throughout a for loop in charge of running the simulation under the specific initial parameters.

3 Results

For the results of the experiment we can see here the different figures where the highest value for each of the steps was found.

Furthermore, violin plot graphs were produced to observe the average distribution for each instance and combination, yielding the following figures:

4 Analysis

For the first iteration we have fig. 1 and fig. 2 where we can see a gradual increment in the highest value, closing towards the optimal value, but eventually plateauing at a sub-optimal value. We can see the steps that it takes for the second combination to reach the plateau is slightly longer, but it has a closer

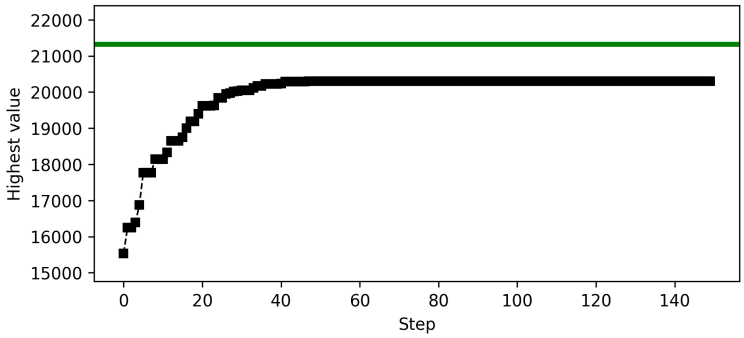

Figure 1: First iteration: first combination

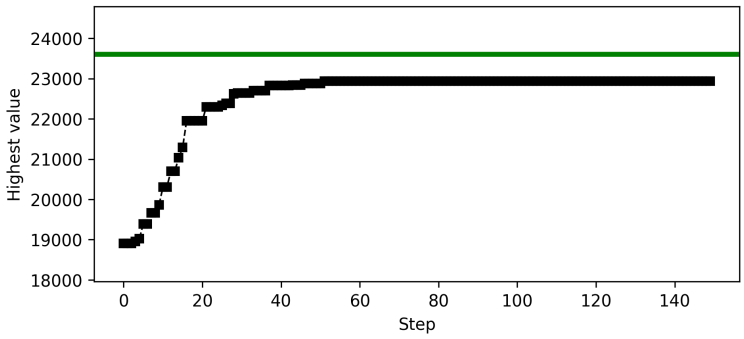

Figure 2: First iteration: second combination

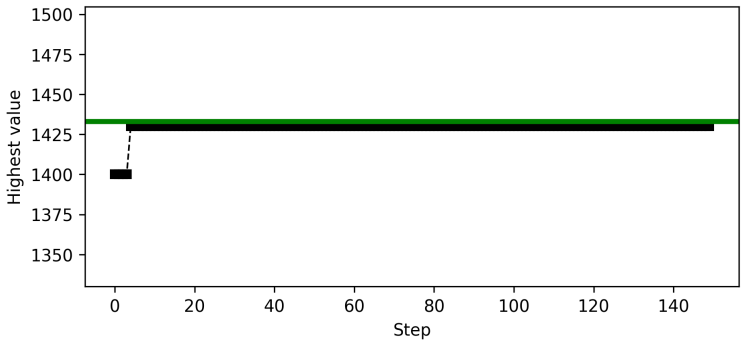

Figure 3: Second iteration: first combination

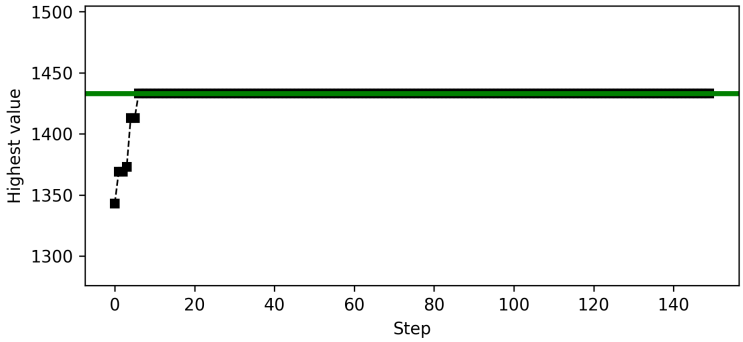

Figure 4: Second iteration: second combination

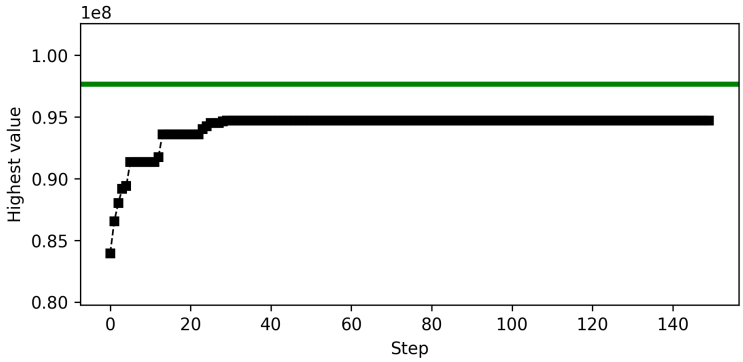

Figure 5: Third iteration: first combination

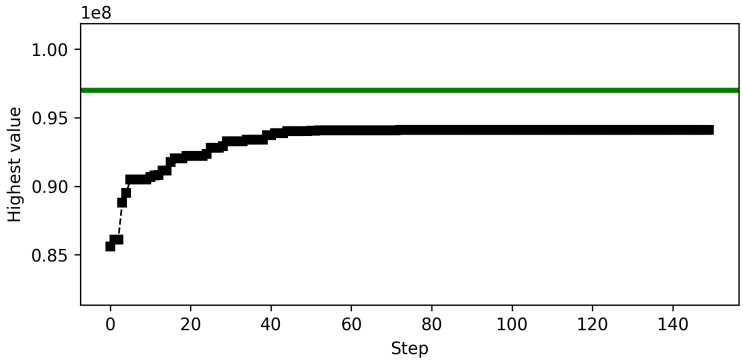

Figure 6: Third iteration: second combination

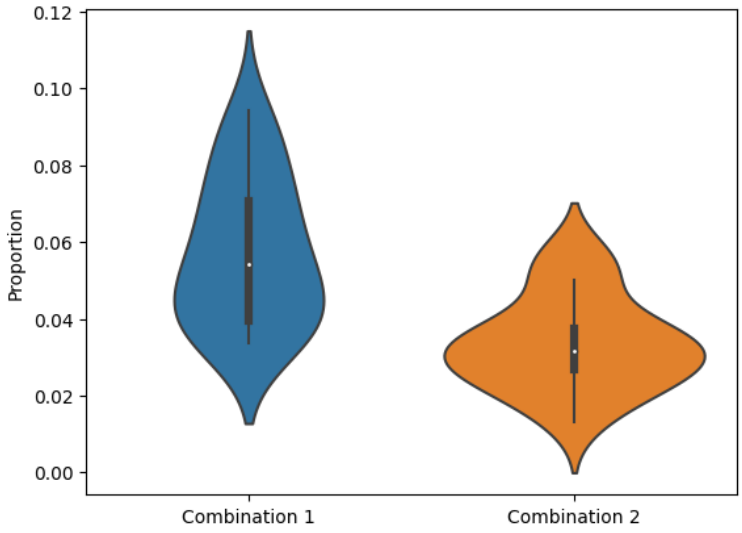

Figure 7: Distribution of iteration 1

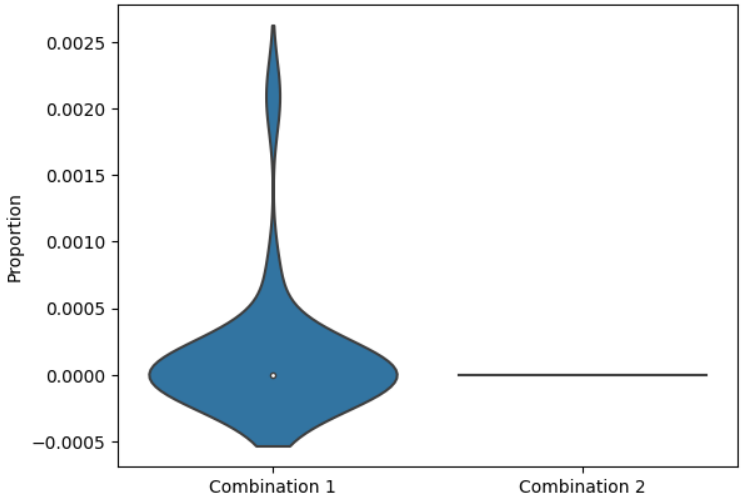

Figure 8: Distribution of iteration 2

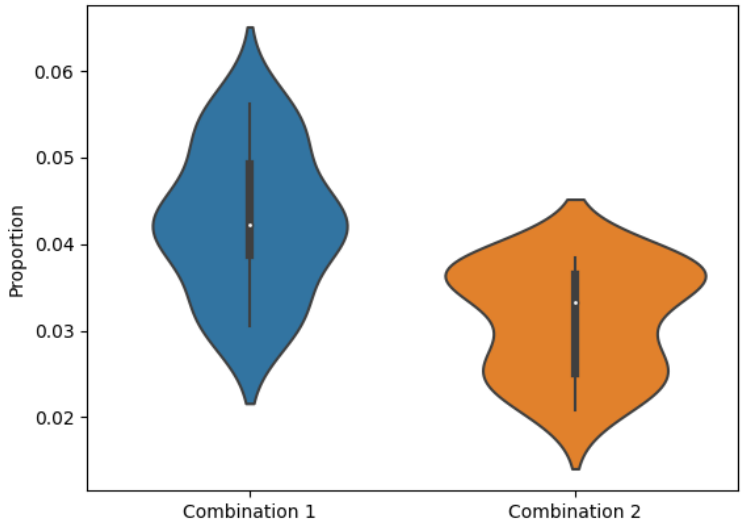

Figure 9: Distribution for iteration 3

approximation to the optimal value. For this iteration, the variance statistical analysis of typew ANOVA yielded a result of variance of ¡ 0.05, meaning that there was the variation was of statistical significance.

For the second iteration we have fig. 3 and fig. 4 where we can see both of them reach almost imediately an optimal highest value for the knapsack, after running an ANOVA type statistical analysis test, the value yielded was ¿ 0.05, meaning that there was no statistical significance to the variation in this particular iteration of the experiment.

For the third iteration we have fig. 5 and fig. 6 where we can see a very similar behaviour to the first iteration, and unsurprisingly we found through the ANOVA type statistical analysis that indeed there was statistical significance between the combination of values 1 and 2.

5 Conclusion

We observed how in iterations 1 and 3 we had a significant impact on the values we applied to the experiment, however the same was not the case for iteration 2, where there was an inversely proportional relationship between the weights and the values. This inverse proportion caused a dramatic and consistent behaviour on the experiments regardless of the combination of values "pm", "rep" and "init".

References

[1] E. Edgardjfc. *GitHub,P10*. URL: https://github.com/edgardjfc/Simulacion-Nano-2022/tree/main/P9.

[2] J. FeroxDeitas. *GitHub,P10*. URL: https://github.com/FeroxDeitas/Simulacion-Nano/tree/main/Tareas/P_10.

[3] E. Schaeffer. *GitHub,geneticAlgorithm*. URL: https://github.com/satuelisa/Simulation/tree/master/GeneticAlgorithm.