COMP6026 - Group selection (and evolving the parameters of it)

Pier Paolo Ippolito
University of Southampton
ppi1u16@ecs.soton.ac.uk

December 27, 2019

1 Introduction

Group selection is an evolutionary mechanism in which natural selection takes place in groups instead than at individual level. One of the main difficulties when trying to model these type of systems, is the fact that selfish individuals can try to take advantage of the group they are in by not giving any contribution to the group and making use at the same of the resources available (making them grow at a faster rate compared to cooperative individuals). This same concept of how having a cooperative or selfish behaviour can favour (or not) an individual, is studied in Behavioral Game Theory [1]. Additionally, it is at times argued that natural selection through evolution based on fitness ("survival of the fittest") incentivize individuals to be selfish in order to outpace the competition [2].

In "Individual Selection for Cooperative Group Formation" by Simon T. Powers et al [3], is examined how individuals cooperative/selfish behaviour in small and large groups can make the different groups evolve over time. A list of all the parameters used by Dr. Powers in order to create his results is available in Table 1. Additionally, in Appendix B are available some results obtained when trying to vary some of these parameters (eg. growth rate and death rate).

Parameter	Value
Death Rate, K	0.1
Large Group Size, L	40
Small Group Size, S	4
Small Group Resources, R_s	4
Large Group Resources, R_l	50
Growth rate (cooperative), G_c	0.018
Growth rate (selfish), G_s	0.02
Consumption rate (cooperative), C_c	0.1
Consumption rate (selfish), C_s	0.2
Population size, N	4000
Number of generations, T	1000
Reproduction cycles, t	4

Table 1: Experiment Parameters

As explained by Dr. Powers, each individual in the population ('pool') had a genotype that carries two parameters (being Cooperative/Selfish and being part of a Small/Large group). During the experiment, resources have been given to the different groups (with large groups having a greater per capita resource allocation compared to small groups) and each genotype had the opportunity to increase its presence in the community (through reproduction) depending on the amount of resources he had been allocated. In Equation 1 and 2 are shown respectively the formulas used in order to determine the amount of resources allocated (r_i) and the population size after reproduction ($n_i(t+1)$).

$$r_i = \frac{n_i G_i C_i}{\sum_j (n_j G_j C_j)} R \quad (1)$$

$$n_i(t+1) = n_i(t) + \frac{r_i}{C_i} - K n_i(t) \quad (2)$$

Overall, the selfish population had both higher consumption rate and growth compared to the cooperative part of the population. The selection of the large and small groups sizes have been chosen so to favourite cooperative expression in the population with small groups and favourite selfish expression in the population with large groups. Additionally, the number of time steps (Disposal Time) to perform reproduction within groups has been chosen so to favourite cooperation. In fact, increasing the amount of time spent in groups before mixing would then lead to a decrease in the expression of cooperators if also selfish individuals are present in the same groups. As a result of this demonstration, has been shown that the overall population (after a fixed number of iterations) converged to be composed by just one genotype: cooperative individuals in small groups.

Following Dr. Powers explanation in [3], the implemented algorithm can be broken down in the following list of steps:

1. **Initialization** = creation of a migrant pool (population) composed by N individuals.
2. **Group Formation** = division of the individuals in the pool into groups.
3. **Reproduction** = performing of reproduction within groups for a determined number of time-steps (t).
4. **Migration pool formation** = return each group result of reproduction to the migrant pool.
5. **Maintaining the global carrying capacity** = rescaling the newly generated migrant pool to its original size N (keeping each genotype individuals proportion).
6. **Iteration** = repeat steps from 2 till 5 for a predetermined number of iterations (T).

In this report, will be outlined how this research study has been reimplemented and extended to take into account a medium group size, mutations and a degree of selfishness and cooperativeness of the different individuals.

Additional results obtained while reimplementing Dr. Powers study can be found in Appendices A and B. Finally, all the code used in order to create these results is available in Appendix D.

In order to reproduce this study, the following notation has been used to represent the different genotypes parameters (Table 2).

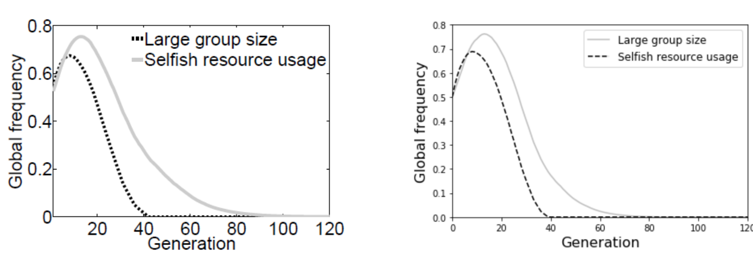
Genotype Parameters	Representation
Selfish and Large	SL
Selfish and Small	SS
Cooperative and Large	CL
Cooperative and Small	CS

Table 2: Genotypes Parameters Representation

2 Reimplementated Results

In Figures 1 and 2 are compared the original results with the reimplemented ones. Overall, it can be clearly seen that the different figures closely match.

In Figure 1, we can see that "Large group size" and "Selfish resource usage" starts with about the same global frequency and follow a similar increasing trend until they reach about the twentieth iteration. This shows that given these initial conditions small groups and cooperators are not favourite by any means (like large groups and selfish individuals instead are). Therefore, following the trend seen so far, we would easily expect to see the population converging to be composed just by large groups and selfish individuals. Nonetheless, after around the twentieth generation this trend gets reversed and small groups and cooperators individuals start taking over the population. By looking Figure 2, we can now try to understand what did lead to this inversion of trend.



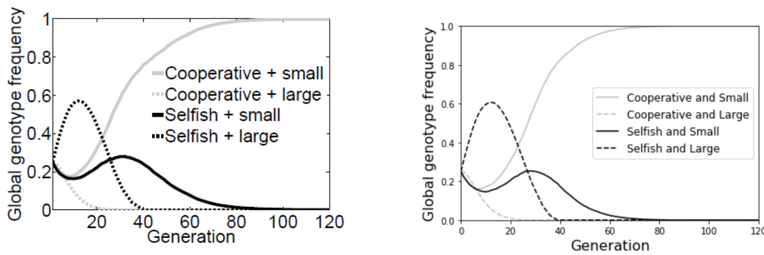
(a) Original

(b) Reimplemented

Figure 1: Average environment and strategy through time

Looking at Figure 2, we can observe how the overall number of each individual genotype parameter changes over time. At the beginning, we can see that large groups seems to have the greater global frequencies (due to their per capita resource advantage) and the selfish individuals resource usage is favoured (selfish individuals take advantage of the remaining resources created by the cooperator individuals). This can be seen by the fact that while the Selfish & Large type increases, it causes a reduction in frequency in the Cooperative & Large type. This demonstrates that because the selfish individuals are taking advantages of their resources, the Cooperative & Large type suffers for this and have less resources to maintain their frequency and reproduce. This makes therefore the selfish individuals strategy unsustainable since they are leading to extinction the same group they were benefiting from (leading finally to the decline in both the Selfish & Large and Cooperative & Large types).

This leaves as now with just two types left: Cooperative & Small and Selfish & Small. Although, in small groups the cooperative strategy is favourite since it drives selfish individuals extinct thanks to the fact that both the groups size and the number of time steps used to perform reproduction within groups are equal each other (to 4).



(a) Original (b) Reimplemented

Figure 2: Change in genotype frequencies over time

Figures 1 (b) and 2 (b) have been both reimplemented by storing the results of the simulation into a Python Pandas Datafarame [4] and then taking the different features to plot using Matplotlib [5]. Additionally, in Appendix A are also available more statistical information extrapolated from this analysis.

3 Extensions

In order to expand the scope of Dr. Powers research, in this section will be examined three different techniques which can be used to add complexity in the created environment and make it therefore look more realistic and applicable in different settings.

The considered complexity additions are:

- **Creating a third group of "middle" size** = Adding a third group, would make this experiment more realistic since individuals in real life usually end up forming groups of various sizes.
- **Adding the possibility of random mutations** = individuals in real life are not static and they might change over time their behaviour. Including random mutations would therefore help in adding this extra level of complexity by allowing for example selfish individuals to become cooperative over time (and vice-versa).
- **Dividing the population in not just selfish/cooperative individuals but in four different levels (from being extremely selfish to being extremely cooperative)** = this representation would resemble more closely real life populations, since different individuals might be more or less selfish/cooperative.

All the code used in order to create the extension for this research paper, is available in Appendix E.

3.1 Experiment 1

Research Question: Using two groups and no mutation, changing the Disposal Time has an important impact in determining if a selfish or cooperative group reaches fixation. Would this still hold true when creating more groups and adding mutations?

Hypothesis: In my belief, is expected to see that the Disposal Time would still play an important role when adding a third group and mutations. Although, when adding mutations, its effects might take longer to be evident.

3.2 Experiment 2

Research Question: Dividing individuals in different levels of selfishness might allows us to gain a better insight of the dynamics within the population. Will genotypes with a medium level of selfishness achieve a greater global frequency compared to the other genotypes?

Hypothesis: In my opinion, it is likely that the population will be composed mainly by genotypes either partially selfish or partially cooperative because they can represent a good evolutionary compromise between the two extremes.

4 Results of extension

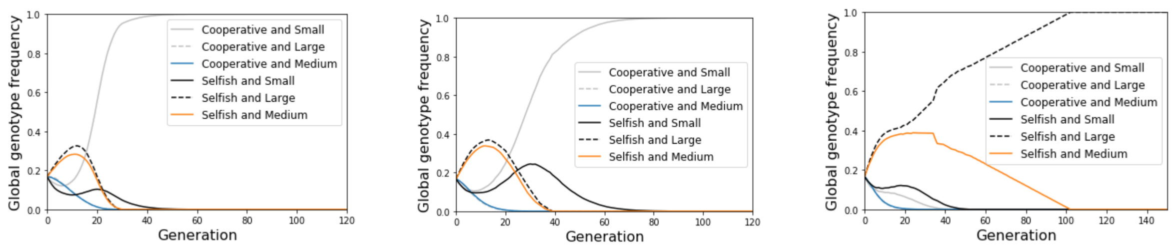
4.1 Experiment 1

In this experiment, has been created a third "middle" group and successively added also an option to mutate a random portion of the population so that the individuals could change from selfish to cooperative and from cooperative to selfish during the course of the simulation (as suggested by Dr. Powers at the end of his research [3]). Additionally, this experiment, has been tested using a range of different Disposal Times.

In order to create a "middle" group, its group size has been calculated to be the average of the big group size and the small group size (22 individuals per group), the same has been done also in order to calculate the associated group resources (Medium Group Resources = 27). All the other parameters remained unchanged.

In Figure 3, are available the results of the simulation with the added third group while varying the Disposal Time with 3 different values (2, 4, 6). As shown below, using a low value for the Disposal Time, facilitates the Cooperative & Small group to achieve an higher global genotype frequency. Instead, increasing the Disposal Time makes more likely for the Selfish & Large population to reach fixation.

This behaviour can be explained by the fact that, if also selfish individuals are present in the same groups, increasing the amount of time spent in groups before mixing inevitably leads to a decrease in the expression of cooperators.

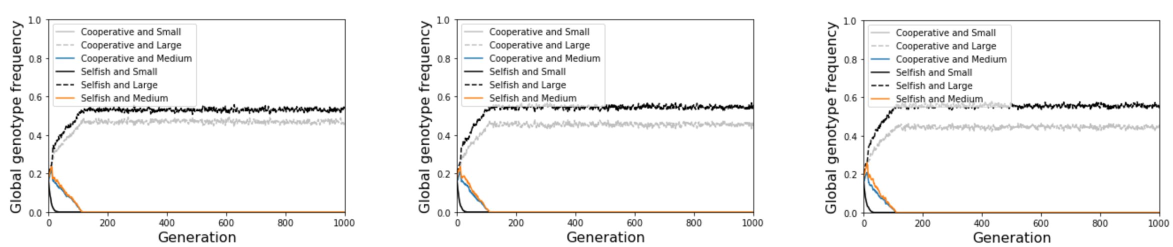


(a) Disposal Time = 2 (b) Disposal Time = 4 (c) Disposal Time = 6

Figure 3: Effects of adding a third "medium" group and varying the Disposal Time

In Figure 4, we can now see how adding mutations can affect the outcome of our simulation. Mutations, have been added at the end of the "Maintaining the global carrying capacity" stage. In this case, has been taken a sample of half the population and to each of the member of this half of the population has been assigned a fifthy percent probability to mutate (changing from being cooperative to selfish or vice-versa).

As shown in Figure 4, adding mutations made our results more consistent, even when varying the Disposal Time. In this case, even if increasing the number of generations to 1000, there is no group completely taking over the whole global genotype frequency, instead the Selfish & Large and Cooperative & Large groups both coexist. These results, therefore go partially against the initially proposed hypothesis. In fact, decreasing the Disposal Time when adding mutation can still increases the chances for Cooperative & Large to overtake Selfish & Large (even though at a reduced rate), but increasing the number of iterations doesn't seem to help to see these these changes happening. Additional information about coexistence dynamics, is available in "The Efficacy of Group Selection is Increased by Coexistence Dynamics within Groups" by Powers et al [6].



(a) Disposal Time = 2 (b) Disposal Time = 4 (c) Disposal Time = 6

Figure 4: Effects of adding a third "medium" group, adding mutation and varying the Disposal Time

Additionally, in Appendix C are available some additional analytical insights highlighting the mean, variance and correlation of the different genotypes when the 3 different groups are examined and no mutations takes place.

4.2 Experiment 2

In this experiment, has been created a population with individuals with 4 different levels of selfishness, by representing each individual by its selfishness level and belonging group, with 0 representing a selfish individual and 3 representing a cooperative individual (Table 3).

Genotype Parameters	Representation
Selfish and Large	0L
Selfish and Small	0S
Quite Selfish and Large	1L
Quite Selfish and Small	1S
Quite Cooperative and Large	2L
Quite Cooperative and Small	2S
Cooperative and Large	3L
Cooperative and Small	3S

Table 3: Extension: Genotypes Parameters Representation

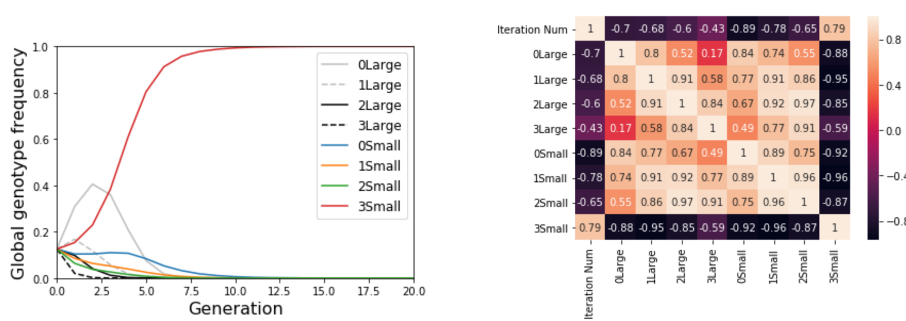
In order to make the different individual less or more selfish, new reproduction and consumption parameters had to be created. For Selfish and Cooperative individuals, have been used the original parameters used in Dr. Powers experiment. Instead, for Quite Selfish and Quite Cooperative, new parameters have been created by scaling respectively the Growth Rate and Consumption Rate of selfish individuals by 1.2 and 1.4. In this way, the overall total between Growth Rate and Consumption Rate almost linearly decreases every time an individual becomes a bit more cooperative (Table 4).

Parameters	Value
Growth Rate (selfish)	0.02
Growth Rate (quite selfish)	0.016
Growth Rate (quite cooperative)	0.014
Growth Rate (cooperative)	0.01
Consumption Rate (selfish)	0.2
Consumption Rate (quite selfish)	0.16
Consumption Rate (quite cooperative)	0.14
Consumption Rate (cooperative)	0.1

Table 4: Extension: Additional Growth and Consumption Rates

The results obtained while running the simulation are available in Figure 5 (a). In Figure 5 (b), is additionally shown a correlation matrix demonstrating how the different groups are correlated each other.

As shown in Figure 5 (a), group 3Small (Cooperative & Small) reached fixation in less than 20 generations.

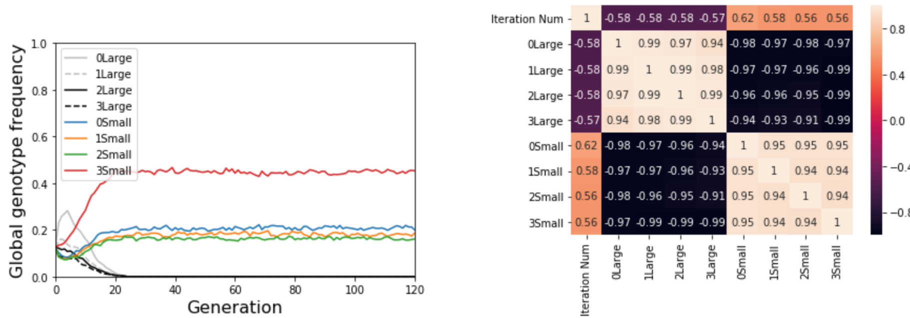


(a) Change in genotype frequencies over time (b) Correlation Matrix between genotypes

Figure 5: Including different levels of selfishness

Finally, mutation has been added in order to compare how the results would vary with this extra complexity element. In this case, mutation has been implemented at the end of the "Maintaining the global carrying capacity" stage by taking a sample of half the population and assigning a 3/4 probability of mutation to each individual. In this way, some of the individuals would have been able to arbitrarily become less or more cooperative.

The final results are shown in Figure 6 (a). Also in this case, group 3Small (Cooperative & Small) was the one achieving the greatest global genotype frequency (although, in coexistence with other 3 groups). In Figure 6 (b), is instead shown the correlation matrix resulting from this experiment (from which can be clearly examined the differences with our previous results in Figure 5 (b)).



(a) Change in genotype frequencies over time

(b) Correlation Matrix between genotypes

Figure 6: Including different levels of selfishness and mutations

Also in this case, the results obtained did not completely support the initial hypothesis (particularly in the simulation without mutations). In fact, in either simulations, genotypes with a medium level of selfishness did not achieve a greater global frequency compared to the other genotypes. Although, in the simulation with mutation, two groups with medium levels of selfishness arrived respectively third and fourth.

5 Further Developments

Overall, this project had a successful outcome providing multiple insights about different techniques which can be used in order to extend Dr. Powers research. Although, some additional features in order to enhance this analysis can still potentially be added. Some examples of further advancements which can be included in this project are:

- Try different medium group sizes and medium group resources.
- Experiment with different number of selfishness levels and consumption/growth rates.
- Observe how simulations can be affected by using different percentages of mutations or likelihoods of mutations in both experiments.

References

[1] Behavioural Economics: Introduction to Behavioral Game Theory and Game Experiments.University of Oxford, Michaelmas Term 2013 Vincent P. Crawford, University of Oxford, All Souls College, and University of California, San Diego. Accessed at: *https://econweb.ucsd.edu/ vcrawfor/BGTIntroductionSlides13.pdf*, Dec 2019.

[2] Evolution, Selfishness and Cooperation Francis HEYLIGHEN, CLEA, Free University of Brussels, Pleinlaan 2, B-1050 Brussels, Belgium. Accessed at: *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.160.7011rep=rep1type=pdf*, Dec 2019.

[3] Individual Selection for Cooperative Group Formation. In Proceedings of the 9th European Conference on Artificial Life (ECAL 2007), pp. 585-594, Lisbon, Portugal. e Costa, F. A., Rocha, L. M., Costa, E., Harvey, I. and Coutinho, A., Eds. Powers, S. T., Penn, A. S. and Watson, R. A. (2007) (2007).Accessed at: *https://eprints.soton.ac.uk/264277/*, Dec 2019.

[4] McKinney, W., others. (2010). Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference (Vol. 445, pp. 51–56). Accessed at: *https://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf*, Dec 2019.

[5] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science Engineering, vol. 9, no. 3, pp. 90-95, 2007 Accessed at: *https://ieeexplore.ieee.org/document/4160265*, Dec 2019.

[6] The Efficacy of Group Selection is Increased by Coexistence Dynamics within Groups, Powers et al. Accessed at: *https://eprints.soton.ac.uk/266450/*, Dec 2019.

Appendix A Reimplementated Results: Analytics

```
Selfish and Large
Mean:  56.327672327672325 Std:  321.4581421993898
Selfish and Small
Mean:  39.968031968031966 Std:  168.90004071423468
Cooperative and Large
Mean:  9.197802197802197 Std:  79.1346860525058
Cooperative and Small
Mean:  3894.3916083916083 Std:  512.7830068142116
```

Figure 1: Mean and Std Comparison of different groups

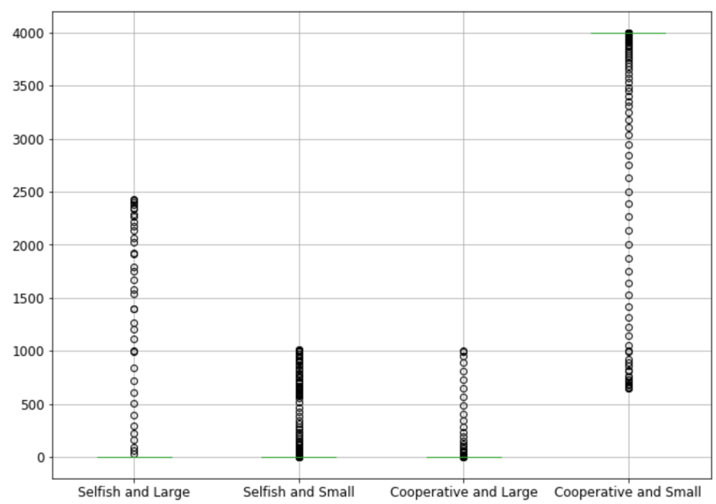


Figure 2: Mean and Variance Comparison of different groups

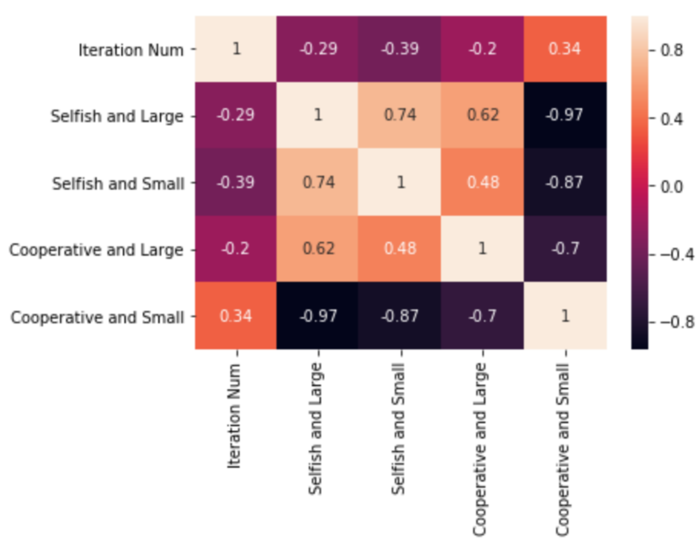
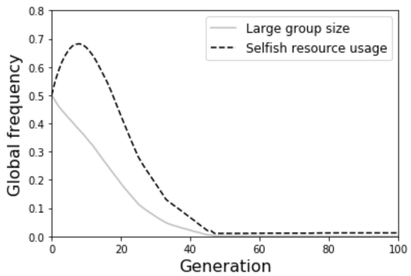


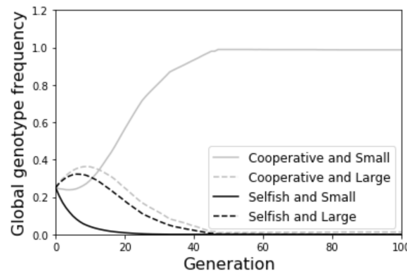
Figure 3: Correlation Matrix

Appendix B Reimplementated Results: Varying Parameters

B.1 Using same Growth Rate (Selfish and Cooperative)



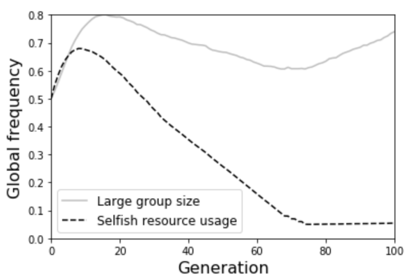
(a) Average environment and strategy through time



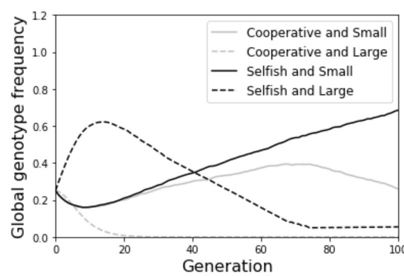
(b) Change in genotype frequencies over time

Figure 4: Using same Growth Rate and 100 iterations

B.2 Varying K (Death Rate = 0.2)



(a) Average environment and strategy through time



(b) Change in genotype frequencies over time

Figure 5: Varying K (Death Rate = 0.2) and using 100 iterations

Appendix C Extension Results: Experiment 1 Analytics (No Mutation)

```
Selfish and Large
Mean:  236.3708609271523 Std:  466.4830883683732
Selfish and Small
Mean:  227.76158940397352 Std:  312.78525965440946
Selfish and Medium
Mean:  211.0662251655629 Std:  425.0528236572439
Cooperative and Large
Mean:  37.23841059602649 Std:  122.19161527088029
Cooperative and Small
Mean:  3247.6754966887415 Std:  1241.8341836697473
Cooperative and Medium
Mean:  38.847682119205295 Std:  124.74146765572995
```

Figure 6: Mean and Std Comparison of different groups (Disposal Time = 4)

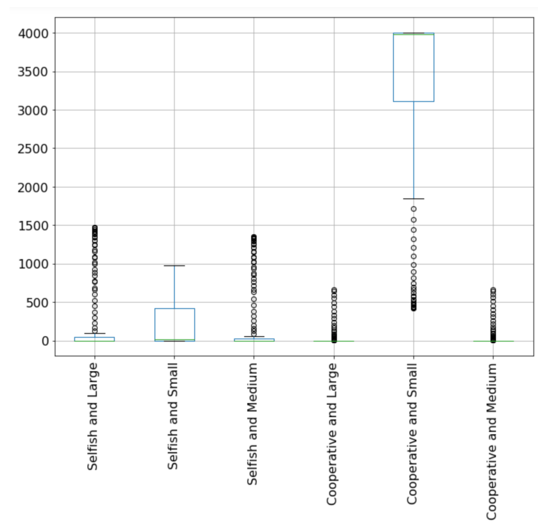


Figure 7: Mean and Variance Comparison of different groups (Disposal Time = 4)

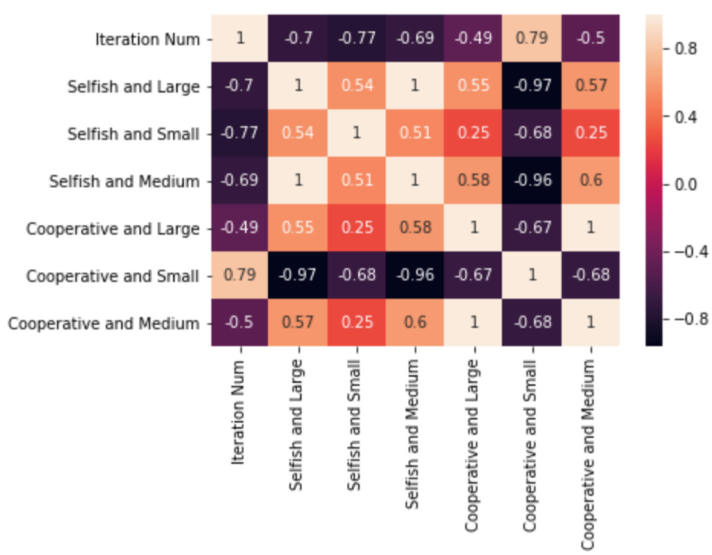


Figure 8: Correlation Matrix

Appendix D Code: Paper Implementation

```python
import pandas as pd
import numpy as np
import math
import random
import itertools
from functools import reduce
import matplotlib.pyplot as plt
import seaborn as sns


def create_pool(size):
    '''
    Creating an initial population containing 4 different types of
    individuals:

    SL: Selfish and Large
    SS: Selfish and Small
    CL: Cooperative and Large
    CS: Cooperative and Small
    '''
    res = np.repeat(['SL', 'SS', 'CL', 'CS'], size/4)
    random.shuffle(res)
    return res


def divide_in_groups(pool, large_g=40, small_g=4):
    '''
    Dividing the current population into two main divisions: one containing
    all the different types of large individuals and one containing instead
    all the types of small individuals. These two divisions are additionally
    splitted in multiple groups depending on the large_g and small_g parameters
    which represent respectively the fixed group size that each large and small
    group should have. If there are not enough individuals left to fill a group
    they are automatically discarded.
    '''
    large = [ind for ind in pool if ind[1] == 'L']
    small = [ind for ind in pool if ind[1] == 'S']
    discard_large = int(large_g*(len(large)/large_g - math.floor(len(large)/
    large_g)))
    discard_small = int(small_g*(len(small)/small_g - math.floor(len(small)/
    small_g)))
    try:
        groups_l = np.array(large[: len(large) - discard_large])
        groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
    except:
        if groups_l.size == 0:
            groups_s = np.array(small[: len(small) - discard_small])
            groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
            return groups_l, groups_s
        groups_l = np.array(large[: len(large) - discard_large - 1])
        groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
    try:
        groups_s = np.array(small[: len(small) - discard_small])
        groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
    except:
        if groups_s.size == 0:
            return groups_l, groups_s
        groups_s = np.array(small[: len(small) - discard_small -1])
        groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
    return groups_l, groups_s
```

```python
def reproduction(large_gs, small_gs, disposal_limit=4, large_r=50, small_r=4, self_g=0.02, coop_g=0.018, self_c=0.2,
                 coop_c=0.1, K=0.1):
    '''
    Reproduction takes place just within divisions and they are dependent
    on the magnitude of the share of the total group resource that the
    genotype receives and the replicator equations (shown above).
    Therefore, the reproduction results are highly dependent of the disposal
    time and the equations parameters.
    '''
    i = 0
    large_g_res = [[]] * len(large_gs)
    small_g_res = [[]] * len(small_gs)
    for large_g, small_g in itertools.zip_longest(large_gs, small_gs):
        if large_g is None:
            unique, counts = np.unique(small_g, return_counts=True)
            small_counts = dict(zip(unique, counts))
            disp_time = 0
            small_coop_individuals = small_counts.get('CS', 0)
            small_self_individuals = small_counts.get('SS', 0)
            while disp_time != disposal_limit:
                small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
                            small_coop_individuals * coop_g * coop_c + small_self_individuals * self_g * self_c) * small_r
                small_self_R_i = (small_self_individuals * self_g * self_c) / (
                            small_self_individuals * self_g * self_c + small_coop_individuals * coop_g * coop_c) * small_r
                small_coop_individuals = small_coop_individuals + small_coop_R_i / coop_c - K * small_coop_individuals
                small_self_individuals = small_self_individuals + small_self_R_i / self_c - K * small_self_individuals
                disp_time += 1

            small_g_res[i] = []
            small_g_res[i].extend(['CS'] * int(small_coop_individuals))
            small_g_res[i].extend(['SS'] * int(small_self_individuals))
            i += 1
        else:
            unique, counts = np.unique(large_g, return_counts=True)
            large_counts = dict(zip(unique, counts))
            disp_time = 0
            large_coop_individuals = large_counts.get('CL', 0)
            large_self_individuals = large_counts.get('SL', 0)
            unique, counts = np.unique(small_g, return_counts=True)
            small_counts = dict(zip(unique, counts))
            small_coop_individuals = small_counts.get('CS', 0)
            small_self_individuals = small_counts.get('SS', 0)
            while disp_time != disposal_limit:
                large_coop_R_i = (large_coop_individuals * coop_g * coop_c) / (
                            large_coop_individuals * coop_g * coop_c + large_self_individuals * self_g * self_c) * large_r
                large_self_R_i = (large_self_individuals * self_g * self_c) / (
                            large_self_individuals * self_g * self_c + large_coop_individuals * coop_g * coop_c) * large_r
                large_coop_individuals = large_coop_individuals + large_coop_R_i / coop_c - K * large_coop_individuals
                large_self_individuals = large_self_individuals + large_self_R_i / self_c - K * large_self_individuals
                small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
                            small_coop_individuals * coop_g * coop_c + small_self_individuals * self_g * self_c) * small_r
                small_self_R_i = (small_self_individuals * self_g * self_c) / (
                            small_self_individuals * self_g * self_c + small_coop_individuals * coop_g * coop_c) * small_r
```

```
                small_coop_individuals = small_coop_individuals + small_coop_R_i / coop_c - K * small_coop_individuals
                small_self_individuals = small_self_individuals + small_self_R_i / self_c - K * small_self_individuals
                disp_time += 1

            large_g_res[i] = []
            large_g_res[i].extend(['CL'] * int(large_coop_individuals))
            large_g_res[i].extend(['SL'] * int(large_self_individuals))
            small_g_res[i] = []
            small_g_res[i].extend(['CS'] * int(small_coop_individuals))
            small_g_res[i].extend(['SS'] * int(small_self_individuals))
            i += 1

    return large_g_res, small_g_res


def update_pool(large_gs, small_gs):
    '''
    In this case, are taken as input the two divisions which had undergone
    reproduction and are merged together to create a pool like the one created
    in the initialization step. In this situation, we need to make sure that
    overall population size remains the same.
    '''
    res = reduce(lambda x,y :x+y ,large_gs, []) + reduce(lambda x,y :x+y , small_gs, [])
    unique, counts = np.unique(res, return_counts=True)
    pop_counts = dict(zip(unique, counts))
    new_pop_elements = [int(((i/len(res))*pop)) for i in pop_counts.values()]
    res = np.repeat([*pop_counts.keys()], new_pop_elements).tolist()
    random.shuffle(res)
    return res


pop = 4000
iter_num = 1000
d = {'Iteration Num': [0], 'Selfish and Large': [1000], 'Selfish and Small': [1000],
     'Cooperative and Large': [1000], 'Cooperative and Small': [1000]}
df = pd.DataFrame(data=d)
migrant_pool = create_pool(pop)
for i in range(1, iter_num + 1):
    large_group, small_group = divide_in_groups(migrant_pool)
    large_group, small_group = reproduction(large_group, small_group)
    migrant_pool = update_pool(large_group, small_group)
    unique, counts = np.unique(migrant_pool, return_counts=True)
    res_counts = dict(zip(unique, counts))
    df = df.append({"Iteration Num": i,
                    "Selfish and Large": res_counts.get('SL', 0),
                    "Selfish and Small": res_counts.get('SS', 0),
                    "Cooperative and Large": res_counts.get('CL', 0),
                    "Cooperative and Small": res_counts.get('CS', 0),
                    }, ignore_index=True)
    if i % 50 == 0:
        print('Iteration Number:', i)


def stats(df):
    '''
    Statistical analysis summary of the experiment
    (eg. Mean and Std of each column, box plot showing
    data distribution of each column and correlation matrix of the dataframe)
    '''
    for i in range(1, len(df.columns)):
```

```
        print(df.columns.values[i])
        print('Mean: ', np.mean(df.iloc[:, i]), 'Std: ', np.std(df.iloc[:, i]))

    df.iloc[:, 1:].plot.box(rot=0, fontsize=12, figsize=(11, 8), grid=True)
    plt.show()

    sns.heatmap(df.corr(), annot=True)


stats(df)

tot = (df['Selfish and Small'] + df['Selfish and Large']+ df['Cooperative and
    Large'] + df['Cooperative and Small'])
plt.plot(df['Iteration Num'], (df['Selfish and Small'] + df['Selfish and Large'])/
    tot, '0.75',
         label = 'Large group size')
plt.plot(df['Iteration Num'], (df['Selfish and Large'] + df['Cooperative and Large
    '])/tot, 'k--',
         label = 'Selfish resource usage')
plt.xlim(0, 120)
plt.ylim(0, 0.8)
plt.xlabel("Generation")
plt.ylabel("Global frequency")
plt.legend()

plt.plot(df['Iteration Num'], df['Cooperative and Small']/tot, '0.75', label = '
    Cooperative and Small')
plt.plot(df['Iteration Num'], df['Cooperative and Large']/tot, '--', color= '0.75'
    , label = 'Cooperative and Large')
plt.plot(df['Iteration Num'], df['Selfish and Small']/tot, 'k', label = 'Selfish
    and Small')
plt.plot(df['Iteration Num'], df['Selfish and Large']/tot, 'k--', label = 'Selfish
     and Large')
plt.xlim(0, 120)
plt.ylim(0, 1)
plt.xlabel("Generation")
plt.ylabel("Global genotype frequency")
plt.legend()
```

Listing 1: Paper_4.py

Appendix E Code: Paper Extension

```python
import pandas as pd
import numpy as np
import math
import random
import itertools
from functools import reduce
import matplotlib.pyplot as plt
import seaborn as sns


def create_pool(size):
    '''
    Creating an initial population containing 4 different types of
    individuals:

    SL: Selfish and Large
    SS: Selfish and Small
    SM: Selfish and Medium
    CL: Cooperative and Large
    CS: Cooperative and Small
    CM: Cooperative and Medium
    '''
    res = np.repeat(['SL', 'SS', 'SM', 'CL', 'CS', 'CM'], size/6)
    random.shuffle(res)
    return res


def divide_in_groups(pool, large_g=40, small_g=4, medium_g=22):
    '''
    Dividing the current population into two main divisions: one containing
    all the different types of large individuals and one containing instead
    all the types of small individuals. These two divisions are additionally
    splitted in multiple groups depending on the large_g and small_g parameters
    which represent respectively the fixed group size that each large and small
    group should have. If there are not enough individuals left to fill a group
    they are automatically discarded.
    '''
    large = [ind for ind in pool if ind[1] == 'L']
    small = [ind for ind in pool if ind[1] == 'S']
    medium = [ind for ind in pool if ind[1] == 'M']
    discard_large = int(large_g*(len(large)/large_g - math.floor(len(large)/
    large_g)))
    discard_small = int(small_g*(len(small)/small_g - math.floor(len(small)/
    small_g)))
    discard_medium = int(medium_g*(len(medium)/medium_g - math.floor(len(medium)/
    medium_g)))
    try:
        groups_l = np.array(large[: len(large) - discard_large])
        groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
    except:
        if groups_l.size == 0:
            groups_s = np.array(small[: len(small) - discard_small])
            groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
            groups_m = np.array(medium[: len(medium) - discard_medium])
            if groups_m.size != 0 and groups_m.size != 1:
                groups_m = groups_m.reshape(math.floor(len(medium)/medium_g), -1)
            return groups_l, groups_s, groups_m
        groups_l = np.array(large[: len(large) - discard_large - 1])
        groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
    try:
        groups_m = np.array(medium[: len(medium) - discard_medium])
        groups_m = groups_m.reshape(math.floor(len(medium)/medium_g), -1)
```

```
    except:
        if groups_m.size == 0:
            groups_s = np.array(small[: len(small) - discard_small])
            if groups_s.size != 0 and groups_s.size != 1:
                groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
            return groups_l, groups_s, groups_m
        groups_m = np.array(medium[: len(medium) - discard_medium -1])
        groups_m = groups_m.reshape(math.floor(len(medium)/medium_g), -1)
    try:
        groups_s = np.array(small[: len(small) - discard_small])
        groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
    except:
        if groups_s.size == 0:
            return groups_l, groups_s, groups_m
        groups_s = np.array(small[: len(small) - discard_small -1])
        groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
    return groups_l, groups_s, groups_m


def reproduction(large_gs, small_gs, medium_gs, disposal_limit=4, large_r=50, small_r=4, medium_r=27, self_g=0.02, coop_g=0.018, self_c=0.2,
                 coop_c=0.1, K=0.1):
    '''
    Reproduction takes place just within divisions and they are dependent
    on the magnitude of the share of the total group resource that the
    genotype receives and the replicator equations (shown above).
    Therefore, the reproduction results are highly dependent of the disposal
    time and the equations parameters.
    '''
    i = 0
    large_g_res = [[]] * len(large_gs)
    small_g_res = [[]] * len(small_gs)
    medium_g_res = [[]] * len(medium_gs)
    for large_g, small_g, medium_g in itertools.zip_longest(large_gs, small_gs, medium_gs):
        if small_g is None and large_g is not None and medium_g is not None:
            unique, counts = np.unique(large_g, return_counts=True)
            large_counts = dict(zip(unique, counts))
            disp_time = 0
            large_coop_individuals = large_counts.get('CL', 0)
            large_self_individuals = large_counts.get('SL', 0)
            unique, counts = np.unique(medium_g, return_counts=True)
            medium_counts = dict(zip(unique, counts))
            medium_coop_individuals = medium_counts.get('CM', 0)
            medium_self_individuals = medium_counts.get('SM', 0)
            while disp_time != disposal_limit:
                large_coop_R_i = (large_coop_individuals * coop_g * coop_c) / (
                            large_coop_individuals * coop_g * coop_c + large_self_individuals * self_g * self_c) * large_r
                large_self_R_i = (large_self_individuals * self_g * self_c) / (
                            large_self_individuals * self_g * self_c + large_coop_individuals * coop_g * coop_c) * large_r
                large_coop_individuals = large_coop_individuals + large_coop_R_i / coop_c - K * large_coop_individuals
                large_self_individuals = large_self_individuals + large_self_R_i / self_c - K * large_self_individuals
                medium_coop_R_i = (medium_coop_individuals * coop_g * coop_c) / (
                            medium_coop_individuals * coop_g * coop_c + medium_self_individuals * self_g * self_c) * medium_r
                medium_self_R_i = (medium_self_individuals * self_g * self_c) / (
                            medium_self_individuals * self_g * self_c + medium_coop_individuals * coop_g * coop_c) * medium_r
                medium_coop_individuals = medium_coop_individuals + medium_coop_R_i / coop_c - K * medium_coop_individuals
```

```
                medium_self_individuals = medium_self_individuals + medium_self_R_i / self_c - K * medium_self_individuals
                disp_time += 1
            medium_g_res[i] = []
            medium_g_res[i].extend(['CM'] * int(medium_coop_individuals))
            medium_g_res[i].extend(['SM'] * int(medium_self_individuals))
            large_g_res[i] = []
            large_g_res[i].extend(['CL'] * int(large_coop_individuals))
            large_g_res[i].extend(['SL'] * int(large_self_individuals))
            i += 1
            continue
        if small_g is None and large_g is not None:
            unique, counts = np.unique(large_g, return_counts=True)
            large_counts = dict(zip(unique, counts))
            disp_time = 0
            large_coop_individuals = large_counts.get('CL', 0)
            large_self_individuals = large_counts.get('SL', 0)
            while disp_time != disposal_limit:
                large_coop_R_i = (large_coop_individuals * coop_g * coop_c) / (
                            large_coop_individuals * coop_g * coop_c + large_self_individuals * self_g * self_c) * large_r
                large_self_R_i = (large_self_individuals * self_g * self_c) / (
                            large_self_individuals * self_g * self_c + large_coop_individuals * coop_g * coop_c) * large_r
                large_coop_individuals = large_coop_individuals + large_coop_R_i / coop_c - K * large_coop_individuals
                large_self_individuals = large_self_individuals + large_self_R_i / self_c - K * large_self_individuals
                disp_time += 1
            large_g_res[i] = []
            large_g_res[i].extend(['CL'] * int(large_coop_individuals))
            large_g_res[i].extend(['SL'] * int(large_self_individuals))
            i += 1
            continue
        if small_g is None and medium_g is not None:
            unique, counts = np.unique(medium_g, return_counts=True)
            medium_counts = dict(zip(unique, counts))
            medium_coop_individuals = medium_counts.get('CM', 0)
            medium_self_individuals = medium_counts.get('SM', 0)
            while disp_time != disposal_limit:
                medium_coop_R_i = (medium_coop_individuals * coop_g * coop_c) / (
                            medium_coop_individuals * coop_g * coop_c + medium_self_individuals * self_g * self_c) * medium_r
                medium_self_R_i = (medium_self_individuals * self_g * self_c) / (
                            medium_self_individuals * self_g * self_c + medium_coop_individuals * coop_g * coop_c) * medium_r
                medium_coop_individuals = medium_coop_individuals + medium_coop_R_i / coop_c - K * medium_coop_individuals
                medium_self_individuals = medium_self_individuals + medium_self_R_i / self_c - K * medium_self_individuals
                disp_time += 1
            medium_g_res[i] = []
            medium_g_res[i].extend(['CM'] * int(medium_coop_individuals))
            medium_g_res[i].extend(['SM'] * int(medium_self_individuals))
            i += 1
            continue
        if large_g is None and medium_g is not None:
            unique, counts = np.unique(medium_g, return_counts=True)
            medium_counts = dict(zip(unique, counts))
            disp_time = 0
            medium_coop_individuals = medium_counts.get('CM', 0)
            medium_self_individuals = medium_counts.get('SM', 0)
            unique, counts = np.unique(small_g, return_counts=True)
            small_counts = dict(zip(unique, counts))
```

```
            small_coop_individuals = small_counts.get('CS', 0)
            small_self_individuals = small_counts.get('SS', 0)
            while disp_time != disposal_limit:
                medium_coop_R_i = (medium_coop_individuals * coop_g * coop_c) / (
                            medium_coop_individuals * coop_g * coop_c + medium_self_individuals * self_g * self_c) * medium_r
                medium_self_R_i = (medium_self_individuals * self_g * self_c) / (
                            medium_self_individuals * self_g * self_c + medium_coop_individuals * coop_g * coop_c) * medium_r
                medium_coop_individuals = medium_coop_individuals + medium_coop_R_i / coop_c - K * medium_coop_individuals
                medium_self_individuals = medium_self_individuals + medium_self_R_i / self_c - K * medium_self_individuals
                small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
                            small_coop_individuals * coop_g * coop_c + small_self_individuals * self_g * self_c) * small_r
                small_self_R_i = (small_self_individuals * self_g * self_c) / (
                            small_self_individuals * self_g * self_c + small_coop_individuals * coop_g * coop_c) * small_r
                small_coop_individuals = small_coop_individuals + small_coop_R_i / coop_c - K * small_coop_individuals
                small_self_individuals = small_self_individuals + small_self_R_i / self_c - K * small_self_individuals
                disp_time += 1

            medium_g_res[i] = []
            medium_g_res[i].extend(['CM'] * int(medium_coop_individuals))
            medium_g_res[i].extend(['SM'] * int(medium_self_individuals))
            small_g_res[i] = []
            small_g_res[i].extend(['CS'] * int(small_coop_individuals))
            small_g_res[i].extend(['SS'] * int(small_self_individuals))
            i += 1
            continue
        if medium_g is None and large_g is None:
            unique, counts = np.unique(small_g, return_counts=True)
            small_counts = dict(zip(unique, counts))
            disp_time = 0
            small_coop_individuals = small_counts.get('CS', 0)
            small_self_individuals = small_counts.get('SS', 0)
            while disp_time != disposal_limit:
                small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
                            small_coop_individuals * coop_g * coop_c + small_self_individuals * self_g * self_c) * small_r
                small_self_R_i = (small_self_individuals * self_g * self_c) / (
                            small_self_individuals * self_g * self_c + small_coop_individuals * coop_g * coop_c) * small_r
                small_coop_individuals = small_coop_individuals + small_coop_R_i / coop_c - K * small_coop_individuals
                small_self_individuals = small_self_individuals + small_self_R_i / self_c - K * small_self_individuals
                disp_time += 1

            small_g_res[i] = []
            small_g_res[i].extend(['CS'] * int(small_coop_individuals))
            small_g_res[i].extend(['SS'] * int(small_self_individuals))
            i += 1
            continue
        if medium_g is not None and large_g is not None:
            unique, counts = np.unique(large_g, return_counts=True)
            large_counts = dict(zip(unique, counts))
            disp_time = 0
            large_coop_individuals = large_counts.get('CL', 0)
            large_self_individuals = large_counts.get('SL', 0)
            unique, counts = np.unique(small_g, return_counts=True)
```

```
            small_counts = dict(zip(unique, counts))
            small_coop_individuals = small_counts.get('CS', 0)
            small_self_individuals = small_counts.get('SS', 0)
            unique, counts = np.unique(medium_g, return_counts=True)
            medium_counts = dict(zip(unique, counts))
            medium_coop_individuals = medium_counts.get('CM', 0)
            medium_self_individuals = medium_counts.get('SM', 0)
            while disp_time != disposal_limit:
                large_coop_R_i = (large_coop_individuals * coop_g * coop_c) / (
                            large_coop_individuals * coop_g * coop_c + large_self_individuals * self_g * self_c) * large_r
                large_self_R_i = (large_self_individuals * self_g * self_c) / (
                            large_self_individuals * self_g * self_c + large_coop_individuals * coop_g * coop_c) * large_r
                large_coop_individuals = large_coop_individuals + large_coop_R_i / coop_c - K * large_coop_individuals
                large_self_individuals = large_self_individuals + large_self_R_i / self_c - K * large_self_individuals
                small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
                            small_coop_individuals * coop_g * coop_c + small_self_individuals * self_g * self_c) * small_r
                small_self_R_i = (small_self_individuals * self_g * self_c) / (
                            small_self_individuals * self_g * self_c + small_coop_individuals * coop_g * coop_c) * small_r
                small_coop_individuals = small_coop_individuals + small_coop_R_i / coop_c - K * small_coop_individuals
                small_self_individuals = small_self_individuals + small_self_R_i / self_c - K * small_self_individuals
                medium_coop_R_i = (medium_coop_individuals * coop_g * coop_c) / (
                            medium_coop_individuals * coop_g * coop_c + medium_self_individuals * self_g * self_c) * medium_r
                medium_self_R_i = (medium_self_individuals * self_g * self_c) / (
                            medium_self_individuals * self_g * self_c + medium_coop_individuals * coop_g * coop_c) * medium_r
                medium_coop_individuals = medium_coop_individuals + medium_coop_R_i / coop_c - K * medium_coop_individuals
                medium_self_individuals = medium_self_individuals + medium_self_R_i / self_c - K * medium_self_individuals
                disp_time += 1

            large_g_res[i] = []
            large_g_res[i].extend(['CL'] * int(large_coop_individuals))
            large_g_res[i].extend(['SL'] * int(large_self_individuals))
            small_g_res[i] = []
            small_g_res[i].extend(['CS'] * int(small_coop_individuals))
            small_g_res[i].extend(['SS'] * int(small_self_individuals))
            medium_g_res[i] = []
            medium_g_res[i].extend(['CM'] * int(medium_coop_individuals))
            medium_g_res[i].extend(['SM'] * int(medium_self_individuals))
            i += 1
            continue

    return large_g_res, small_g_res, medium_g_res


def update_pool(large_gs, small_gs, medium_gs, mutation=False):
    '''
    In this case, are taken as input the two divisions which had undergone
    reproduction and are merged together to create a pool like the one created
    in the initialization step. In this situation, we need to make sure that
    overall population size remains the same.
    '''
    res = reduce(lambda x,y :x+y ,large_gs, []) + reduce(lambda x,y :x+y , small_gs, []) + reduce(lambda x,y :x+y , medium_gs, [])
```

```python
    unique, counts = np.unique(res, return_counts=True)
    pop_counts = dict(zip(unique, counts))
    new_pop_elements = [int(((i/len(res))*pop)) for i in pop_counts.values()]
    res = np.repeat([*pop_counts.keys()], new_pop_elements).tolist()
    random.shuffle(res)
    if mutation is True:
        for i in range(0, int(len(res)/2)):
            res[i] = (np.random.choice(['C' + res[i][1], 'S' + res[i][1]],
                                p=[1/2, 1/2]))
    return res


pop = 4000
iter_num = 150
d = {'Iteration Num': [0], 'Selfish and Large': [666], 'Selfish and Small': [666],
     'Selfish and Medium': [666],
      'Cooperative and Large': [666], 'Cooperative and Small': [666], 'Cooperative
    and Medium': [666]}
df = pd.DataFrame(data=d)
migrant_pool = create_pool(pop)
for i in range(1, iter_num + 1):
    large_group, small_group, medium_group = divide_in_groups(migrant_pool)
    large_group, small_group, medium_group = reproduction(large_group, small_group
    , medium_group)
    migrant_pool = update_pool(large_group, small_group, medium_group)
    unique, counts = np.unique(migrant_pool, return_counts=True)
    res_counts = dict(zip(unique, counts))
    df = df.append({"Iteration Num": i,
                    "Selfish and Large": res_counts.get('SL', 0),
                    "Selfish and Small": res_counts.get('SS', 0),
                    "Selfish and Medium": res_counts.get('SM', 0),
                    "Cooperative and Large": res_counts.get('CL', 0),
                    "Cooperative and Small": res_counts.get('CS', 0),
                    "Cooperative and Medium": res_counts.get('CM', 0)
                    }, ignore_index=True)
    if i % 50 == 0:
        print('Iteration Number:', i)
```

Listing 2: Three_groups_and_mutation.py

```python
import pandas as pd
import numpy as np
import math
import random
import itertools
from functools import reduce
import matplotlib.pyplot as plt
import seaborn as sns


def create_pool(size):
    '''
    Creating an initial population containing 4 different types of
    individuals:

    0 -> Selfish
    1 -> Quite Selfish
    2 -> Quite Cooperative
    3 -> Cooperative
    '''
    res = np.repeat(['0L', '0S', '1L', '1S', '2L', '2S', '3L', '3S'], size/4)
    random.shuffle(res)
    return res


```

```python
def divide_in_groups(pool, large_g=40, small_g=4):
    '''
    Dividing the current population into two main divisions: one containing
    all the different types of large individuals and one containing instead
    all the types of small individuals. These two divisions are additionally
    splitted in multiple groups depending on the large_g and small_g parameters
    which represent respectively the fixed group size that each large and small
    group should have. If there are not enough individuals left to fill a group
    they are automatically discarded.
    '''
    large = [ind for ind in pool if ind[1] == 'L']
    small = [ind for ind in pool if ind[1] == 'S']
    discard_large = int(large_g*(len(large)/large_g - math.floor(len(large)/large_g)))
    discard_small = int(small_g*(len(small)/small_g - math.floor(len(small)/small_g)))
    try:
        groups_l = np.array(large[: len(large) - discard_large])
        groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
    except:
        if groups_l.size == 0:
            groups_s = np.array(small[: len(small) - discard_small])
            groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
            return groups_l, groups_s
        groups_l = np.array(large[: len(large) - discard_large - 1])
        groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
    try:
        groups_s = np.array(small[: len(small) - discard_small])
        groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
    except:
        if groups_s.size == 0:
            return groups_l, groups_s
        groups_s = np.array(small[: len(small) - discard_small -1])
        groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
    return groups_l, groups_s


def reproduction(large_gs, small_gs, disposal_limit=4, large_r=50, small_r=4, self_g=0.02, coop_g=0.018, self_c=0.2,
                 coop_c=0.1, K=0.1):
    '''
    Reproduction takes place just within divisions and they are dependent
    on the magnitude of the share of the total group resource that the
    genotype receives and the replicator equations (shown above).
    Therefore, the reproduction results are highly dependent of the disposal
    time and the equations parameters.
    '''
    i = 0
    large_g_res = [[]] * len(large_gs)
    small_g_res = [[]] * len(small_gs)
    #print(len(large_gs), len(small_gs))
    for large_g, small_g in itertools.zip_longest(large_gs, small_gs):
        if large_g is None and small_g is not None:
            unique, counts = np.unique(small_g, return_counts=True)
            small_counts = dict(zip(unique, counts))
            disp_time = 0
            small_1 = small_counts.get('0S', 0)
            small_2 = small_counts.get('1S', 0)
            small_3 = small_counts.get('2S', 0)
            small_4 = small_counts.get('3S', 0)
            dem = (small_1 * self_g * self_c + small_2 * ((self_g/1.2) * (self_c/1.2)) + small_3 * ((self_g/1.4) * (self_c/1.4)) + small_4 * coop_g * coop_c)
            while disp_time != disposal_limit:
                small_1_R_i = (small_1 * self_g * self_c) / dem * small_r
```

```python
            small_2_R_i = (small_2 * ((self_g/1.2) * (self_c/1.2))) / dem * small_r
            small_3_R_i = (small_3 * ((self_g/1.4) * (self_c/1.4))) / dem * small_r
            small_4_R_i = (small_4 * coop_g * coop_c) / dem * small_r
            small_1 = small_1 + small_1_R_i / self_c - K * small_1
            small_2 = small_2 + (small_2_R_i / (self_c/1.2)) - K * small_2
            small_3 = small_3 + (small_3_R_i / (self_c/1.4)) - K * small_3
            small_4 = small_4 + small_4_R_i / coop_c - K * small_4
            disp_time += 1

        small_g_res[i] = []
        small_g_res[i].extend(['0S'] * int(small_1))
        small_g_res[i].extend(['1S'] * int(small_2))
        small_g_res[i].extend(['2S'] * int(small_3))
        small_g_res[i].extend(['3S'] * int(small_4))
        i += 1
    elif small_g is None and large_g is not None:
        disp_time = 0
        unique, counts = np.unique(large_g, return_counts=True)
        large_counts = dict(zip(unique, counts))
        large_1 = large_counts.get('0L', 0)
        large_2 = large_counts.get('1L', 0)
        large_3 = large_counts.get('2L', 0)
        large_4 = large_counts.get('3L', 0)
        dem2 = (large_1 * self_g * self_c + large_2 * ((self_g/1.2) * (self_c/1.2)) + large_3 * ((self_g/1.4) * (self_c/1.4)) + large_4 * coop_g * coop_c)
        while disp_time != disposal_limit:
            large_1_R_i = (large_1 * self_g * self_c) / dem2 * large_r
            large_2_R_i = (large_2 * ((self_g/1.2) * (self_c/1.2))) / dem2 * large_r
            large_3_R_i = (large_3 * ((self_g/1.4) * (self_c/1.4))) / dem2 * large_r
            large_4_R_i = (large_4 * coop_g * coop_c) / dem2 * large_r
            large_1 = large_1 + large_1_R_i / self_c - K * large_1
            large_2 = large_2 + (large_2_R_i / (self_c/1.2)) - K * large_2
            large_3 = large_3 + (large_3_R_i / (self_c/1.4)) - K * large_3
            large_4 = large_4 + large_4_R_i / self_c - K * large_4
            disp_time += 1

        large_g_res[i] = []
        large_g_res[i].extend(['0L'] * int(large_1))
        large_g_res[i].extend(['1L'] * int(large_2))
        large_g_res[i].extend(['2L'] * int(large_3))
        large_g_res[i].extend(['3L'] * int(large_4))
        i += 1
    else:
        unique, counts = np.unique(small_g, return_counts=True)
        small_counts = dict(zip(unique, counts))
        disp_time = 0
        small_1 = small_counts.get('0S', 0)
        small_2 = small_counts.get('1S', 0)
        small_3 = small_counts.get('2S', 0)
        small_4 = small_counts.get('3S', 0)
        dem = (small_1 * self_g * self_c + small_2 * ((self_g/1.2) * (self_c/1.2)) + small_3 * ((self_g/1.4) * (self_c/1.4)) + small_4 * coop_g * coop_c)
        unique, counts = np.unique(large_g, return_counts=True)
        large_counts = dict(zip(unique, counts))
        large_1 = large_counts.get('0L', 0)
        large_2 = large_counts.get('1L', 0)
        large_3 = large_counts.get('2L', 0)
        large_4 = large_counts.get('3L', 0)
        dem2 = (large_1 * self_g * self_c + large_2 * ((self_g/1.2) * (self_c/1.2)) + large_3 * ((self_g/1.4) * (self_c/1.4)) + large_4 * coop_g * coop_c)
```

```
            while disp_time != disposal_limit:
                large_1_R_i = (large_1 * self_g * self_c) / dem2 * large_r
                large_2_R_i = (large_2 * ((self_g/1.2) * (self_c/1.2))) / dem2 * large_r
                large_3_R_i = (large_3 * ((self_g/1.4) * (self_c/1.4))) / dem2 * large_r
                large_4_R_i = (large_4 * coop_g * coop_c) / dem2 * large_r
                large_1 = large_1 + large_1_R_i / self_c - K * large_1
                large_2 = large_2 + (large_2_R_i / (self_c/1.2)) - K * large_2
                large_3 = large_3 + (large_3_R_i / (self_c/1.4)) - K * large_3
                large_4 = large_4 + large_4_R_i / self_c - K * large_4
                small_1_R_i = (small_1 * self_g * self_c) / dem * small_r
                small_2_R_i = (small_2 * ((self_g/1.2) * (self_c/1.2))) / dem * small_r
                small_3_R_i = (small_3 * ((self_g/1.4) * (self_c/1.4))) / dem * small_r
                small_4_R_i = (small_4 * coop_g * coop_c) / dem * small_r
                small_1 = small_1 + small_1_R_i / self_c - K * small_1
                small_2 = small_2 + (small_2_R_i / (self_c/1.2)) - K * small_2
                small_3 = small_3 + (small_3_R_i / (self_c/1.4)) - K * small_3
                small_4 = small_4 + small_4_R_i / coop_c - K * small_4
                disp_time += 1

            large_g_res[i] = []
            large_g_res[i].extend(['0L'] * int(large_1))
            large_g_res[i].extend(['1L'] * int(large_2))
            large_g_res[i].extend(['2L'] * int(large_3))
            large_g_res[i].extend(['3L'] * int(large_4))
            small_g_res[i] = []
            small_g_res[i].extend(['0S'] * int(small_1))
            small_g_res[i].extend(['1S'] * int(small_2))
            small_g_res[i].extend(['2S'] * int(small_3))
            small_g_res[i].extend(['3S'] * int(small_4))
            i += 1

    return large_g_res, small_g_res


def update_pool(large_gs, small_gs, mutation=False):
    '''
    In this case, are taken as input the two divisions which had undergone
    reproduction and are merged together to create a pool like the one created
    in the initialization step. In this situation, we need to make sure that
    overall population size remains the same.
    '''
    res = reduce(lambda x,y :x+y ,large_gs, []) + reduce(lambda x,y :x+y , small_gs, [])
    unique, counts = np.unique(res, return_counts=True)
    pop_counts = dict(zip(unique, counts))
    new_pop_elements = [int(((i/len(res))*pop)) for i in pop_counts.values()]
    res = np.repeat([*pop_counts.keys()], new_pop_elements).tolist()
    random.shuffle(res)
    if mutation is True:
        for i in range(0, int(len(res)/2)):
            res[i] = (np.random.choice(['0' + res[i][1], '1' + res[i][1], '2' + res[i][1], '3' + res[i][1]],
                            p=[1/4, 1/4, 1/4, 1/4]))
    return res


pop = 4000
iter_num = 120
d = {'Iteration Num': [0], '0Large': [500], '1Large': [500],
     '2Large': [500], '3Large': [500], '0Small': [500],
```

```
      '1Small': [500], '2Small': [500], '3Small': [500]}
df = pd.DataFrame(data=d)
migrant_pool = create_pool(pop)
for i in range(1, iter_num + 1):
    large_group, small_group = divide_in_groups(migrant_pool)
    large_group, small_group = reproduction(large_group, small_group)
    migrant_pool = update_pool(large_group, small_group, mutation=True)
    unique, counts = np.unique(migrant_pool, return_counts=True)
    res_counts = dict(zip(unique, counts))
    df = df.append({"Iteration Num": i,
                    "0Large": res_counts.get('0L', 0),
                    "1Large": res_counts.get('1L', 0),
                    "2Large": res_counts.get('2L', 0),
                    "3Large": res_counts.get('3L', 0),
                    "0Small": res_counts.get('0S', 0),
                    "1Small": res_counts.get('1S', 0),
                    "2Small": res_counts.get('2S', 0),
                    "3Small": res_counts.get('3S', 0),
                    }, ignore_index=True)
    if i % 5 == 0:
        print('Iteration Number:', i)
```
Listing 3: Including_levels_of_selfishness.py