
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 Reimplemented 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.

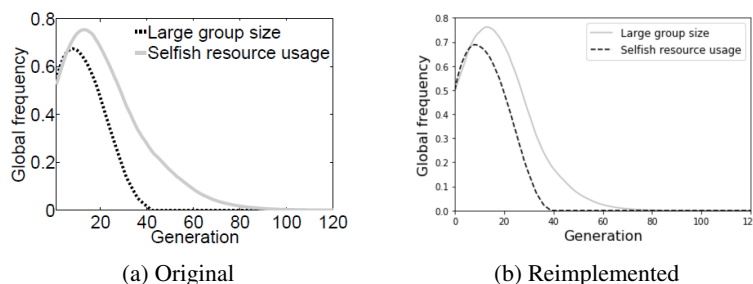


Figure 1: Average environment and strategy through time

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).

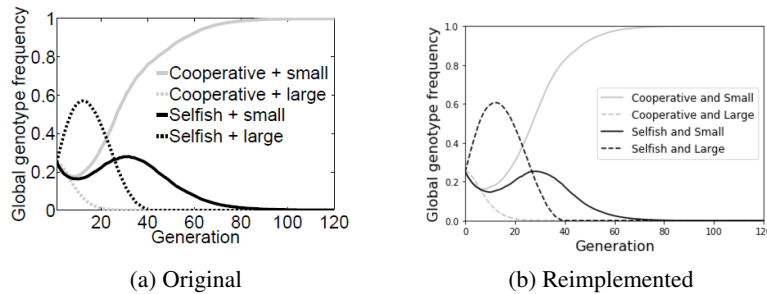


Figure 2: Change in genotype frequencies over time

Figures 1 (b) and 2 (b) have been both reimplemented by storing the results of the simulation into a Python Pandas Dataframe [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.

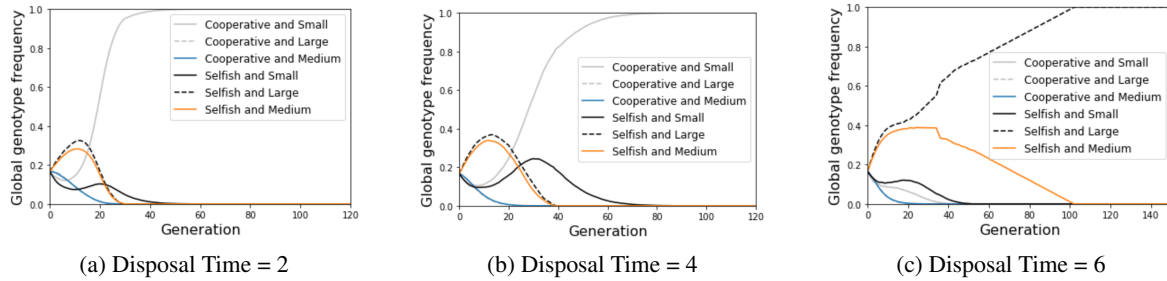


Figure 3: Effects of adding a third "medium" group and varying the Disposal Time

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 fifty 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 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].

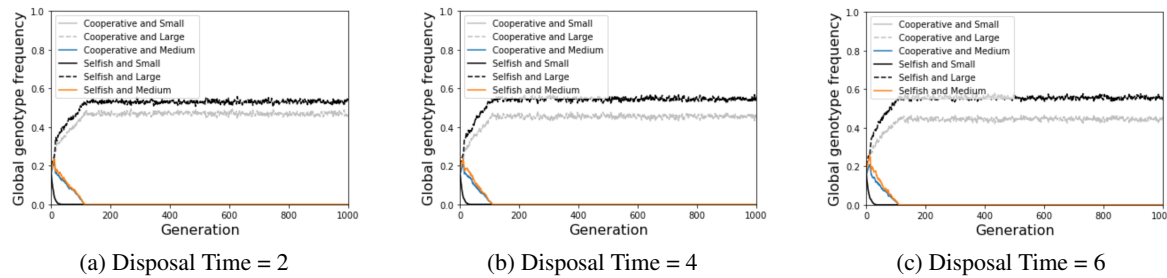


Figure 4: Effects of adding a third "medium" group, adding mutation and varying the Disposal Time

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.

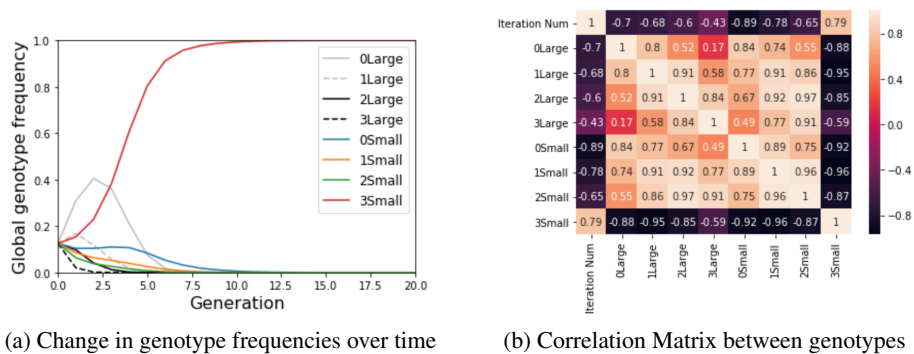


Figure 5: Including different levels of selfishness

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)).

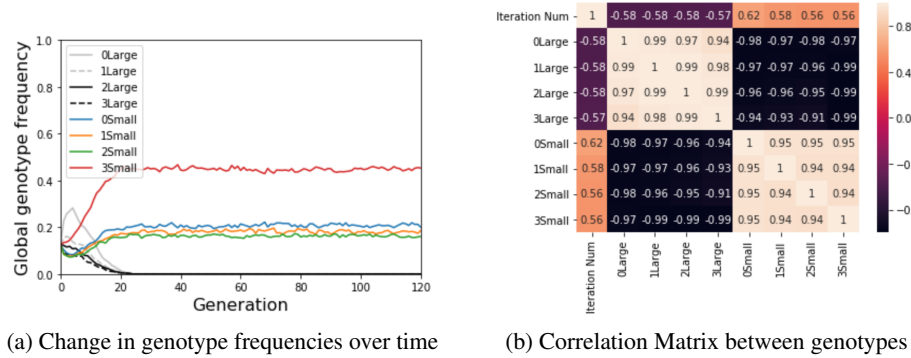


Figure 6: Including different levels of selfishness and mutations

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 Reimplemented 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

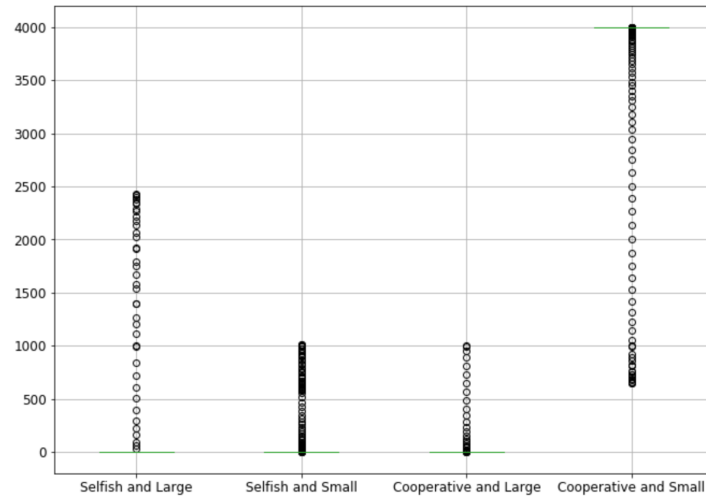


Figure 2: Mean and Variance Comparison of different groups

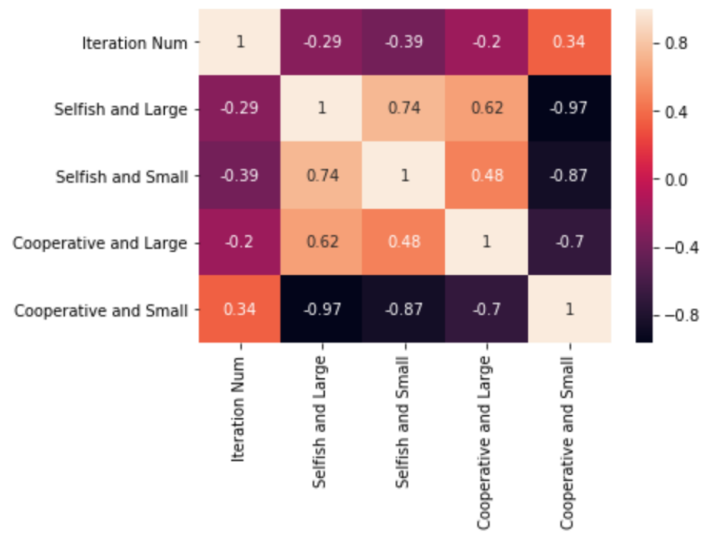
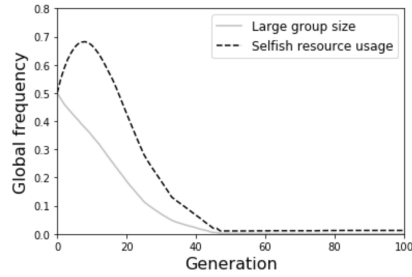


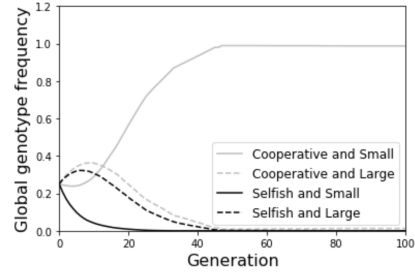
Figure 3: Correlation Matrix

Appendix B Reimplemented Results: Varying Parameters

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



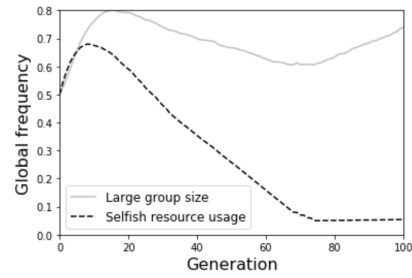
(a) Average environment and strategy through time



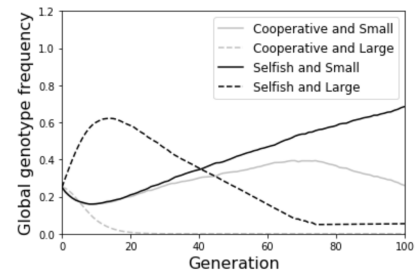
(b) Change in genotype frequencies over time

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)

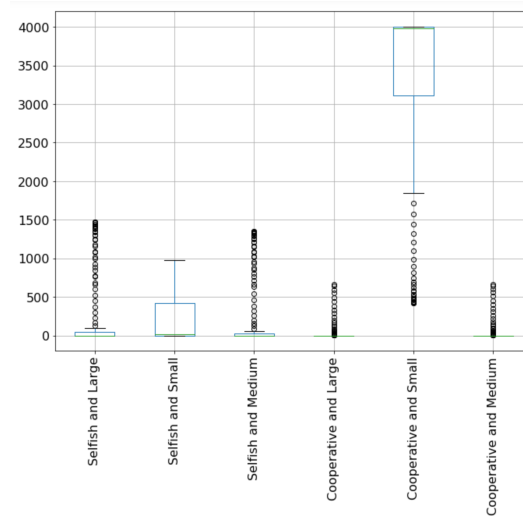


Figure 7: Mean and Variance Comparison of different groups (Disposal Time = 4)

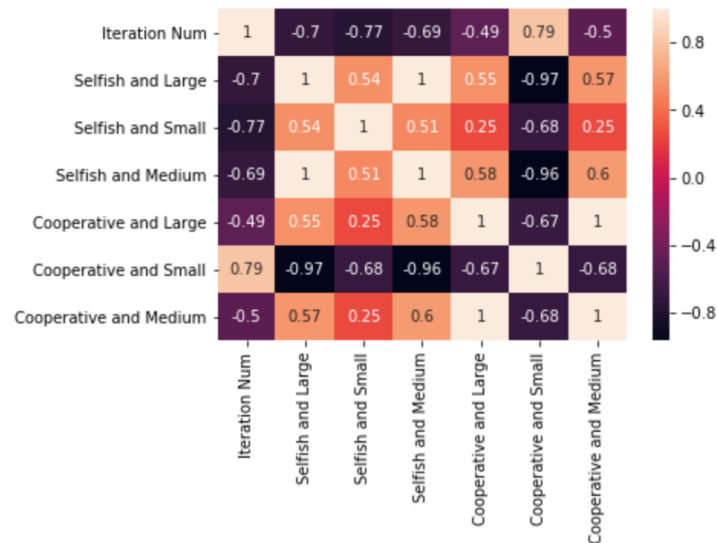


Figure 8: Correlation Matrix

Appendix D Code: Paper Implementation

```
1 import pandas as pd
2 import numpy as np
3 import math
4 import random
5 import itertools
6 from functools import reduce
7 import matplotlib.pyplot as plt
8 import seaborn as sns
9
10
11 def create_pool(size):
12     '''
13     Creating an initial population containing 4 different types of
14     individuals:
15
16     SL: Selfish and Large
17     SS: Selfish and Small
18     CL: Cooperative and Large
19     CS: Cooperative and Small
20     '''
21     res = np.repeat(['SL', 'SS', 'CL', 'CS'], size/4)
22     random.shuffle(res)
23     return res
24
25
26 def divide_in_groups(pool, large_g=40, small_g=4):
27     '''
28     Dividing the current population into two main divisions: one containing
29     all the different types of large individuals and one containing instead
30     all the types of small individuals. These two divisions are additionally
31     splitted in multiple groups depending on the large_g and small_g parameters
32     which represent respectively the fixed group size that each large and small
33     group should have. If there are not enough individuals left to fill a group
34     they are automatically discarded.
35     '''
36     large = [ind for ind in pool if ind[1] == 'L']
37     small = [ind for ind in pool if ind[1] == 'S']
38     discard_large = int(large_g*(len(large)/large_g - math.floor(len(large)/
39     large_g)))
40     discard_small = int(small_g*(len(small)/small_g - math.floor(len(small)/
41     small_g)))
42     try:
43         groups_l = np.array(large[: len(large) - discard_large])
44         groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
45     except:
46         if groups_l.size == 0:
47             groups_s = np.array(small[: len(small) - discard_small])
48             groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
49             return groups_l, groups_s
50         groups_l = np.array(large[: len(large) - discard_large - 1])
51         groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
52     try:
53         groups_s = np.array(small[: len(small) - discard_small])
54         groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
55     except:
56         if groups_s.size == 0:
57             return groups_l, groups_s
58         groups_s = np.array(small[: len(small) - discard_small - 1])
59         groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
60     return groups_l, groups_s
```

```

61 def reproduction(large_gs, small_gs, disposal_limit=4, large_r=50, small_r=4,
62                 self_g=0.02, coop_g=0.018, self_c=0.2,
63                 coop_c=0.1, K=0.1):
64     '''
65     Reproduction takes place just within divisions and they are dependent
66     on the magnitude of the share of the total group resource that the
67     genotype receives and the replicator equations (shown above).
68     Therefore, the reproduction results are highly dependent of the disposal
69     time and the equations parameters.
70     '''
71     i = 0
72     large_g_res = [[] * len(large_gs)]
73     small_g_res = [[] * len(small_gs)]
74     for large_g, small_g in itertools.zip_longest(large_gs, small_gs):
75         if large_g is None:
76             unique, counts = np.unique(small_g, return_counts=True)
77             small_counts = dict(zip(unique, counts))
78             disp_time = 0
79             small_coop_individuals = small_counts.get('CS', 0)
80             small_self_individuals = small_counts.get('SS', 0)
81             while disp_time != disposal_limit:
82                 small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
83                     small_coop_individuals * coop_g * coop_c +
84                     small_self_individuals * self_g * self_c) * small_r
85                 small_self_R_i = (small_self_individuals * self_g * self_c) / (
86                     small_coop_individuals * coop_g * coop_c +
87                     small_self_individuals * self_g * self_c) * small_r
88                 small_coop_individuals = small_coop_individuals + small_coop_R_i /
89                 coop_c - K * small_coop_individuals
90                 small_self_individuals = small_self_individuals + small_self_R_i /
91                 self_c - K * small_self_individuals
92                 disp_time += 1
93
94             small_g_res[i] = []
95             small_g_res[i].extend(['CS'] * int(small_coop_individuals))
96             small_g_res[i].extend(['SS'] * int(small_self_individuals))
97             i += 1
98         else:
99             unique, counts = np.unique(large_g, return_counts=True)
100             large_counts = dict(zip(unique, counts))
101             disp_time = 0
102             large_coop_individuals = large_counts.get('CL', 0)
103             large_self_individuals = large_counts.get('SL', 0)
104             unique, counts = np.unique(small_g, return_counts=True)
105             small_counts = dict(zip(unique, counts))
106             small_coop_individuals = small_counts.get('CS', 0)
107             small_self_individuals = small_counts.get('SS', 0)
108             while disp_time != disposal_limit:
109                 large_coop_R_i = (large_coop_individuals * coop_g * coop_c) / (
110                     large_coop_individuals * coop_g * coop_c +
111                     large_self_individuals * self_g * self_c) * large_r
112                 large_self_R_i = (large_self_individuals * self_g * self_c) / (
113                     large_coop_individuals * coop_g * coop_c +
114                     large_self_individuals * self_g * self_c) * large_r
115                 large_coop_individuals = large_coop_individuals + large_coop_R_i /
116                 coop_c - K * large_coop_individuals
117                 large_self_individuals = large_self_individuals + large_self_R_i /
118                 self_c - K * large_self_individuals
119                 small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
120                     small_coop_individuals * coop_g * coop_c +
121                     small_self_individuals * self_g * self_c) * small_r
122                 small_self_R_i = (small_self_individuals * self_g * self_c) / (
123                     small_coop_individuals * coop_g * coop_c +
124                     small_self_individuals * self_g * self_c) * small_r
125                 small_coop_individuals = small_coop_individuals + small_coop_R_i /
126                 coop_c - K * small_coop_individuals
127                 small_self_individuals = small_self_individuals + small_self_R_i /
128                 self_c - K * small_self_individuals
129                 disp_time += 1

```

```

114         small_coop_individuals = small_coop_individuals + small_coop_R_i /
coop_c - K * small_coop_individuals
115         small_self_individuals = small_self_individuals + small_self_R_i /
self_c - K * small_self_individuals
116         disp_time += 1
117
118         large_g_res[i] = []
119         large_g_res[i].extend(['CL'] * int(large_coop_individuals))
120         large_g_res[i].extend(['SL'] * int(large_self_individuals))
121         small_g_res[i] = []
122         small_g_res[i].extend(['CS'] * int(small_coop_individuals))
123         small_g_res[i].extend(['SS'] * int(small_self_individuals))
124         i += 1
125
126     return large_g_res, small_g_res
127
128
129 def update_pool(large_gs, small_gs):
130     '''
131     In this case, are taken as input the two divisions which had undergone
132     reproduction and are merged together to create a pool like the one created
133     in the initialization step. In this situation, we need to make sure that
134     overall population size remains the same.
135     '''
136     res = reduce(lambda x,y :x+y ,large_gs, []) + reduce(lambda x,y :x+y ,
small_gs, [])
137     unique, counts = np.unique(res, return_counts=True)
138     pop_counts = dict(zip(unique, counts))
139     new_pop_elements = [int(((i/len(res))*pop)) for i in pop_counts.values()]
140     res = np.repeat([*pop_counts.keys()], new_pop_elements).tolist()
141     random.shuffle(res)
142     return res
143
144
145 pop = 4000
146 iter_num = 1000
147 d = {'Iteration Num': [0], 'Selfish and Large': [1000], 'Selfish and Small':
[1000],
148      'Cooperative and Large': [1000], 'Cooperative and Small': [1000]}
149 df = pd.DataFrame(data=d)
150 migrant_pool = create_pool(pop)
151 for i in range(1, iter_num + 1):
152     large_group, small_group = divide_in_groups(migrant_pool)
153     large_group, small_group = reproduction(large_group, small_group)
154     migrant_pool = update_pool(large_group, small_group)
155     unique, counts = np.unique(migrant_pool, return_counts=True)
156     res_counts = dict(zip(unique, counts))
157     df = df.append({"Iteration Num": i,
158                   "Selfish and Large": res_counts.get('SL', 0),
159                   "Selfish and Small": res_counts.get('SS', 0),
160                   "Cooperative and Large": res_counts.get('CL', 0),
161                   "Cooperative and Small": res_counts.get('CS', 0),
162                   }, ignore_index=True)
163     if i % 50 == 0:
164         print('Iteration Number:', i)
165
166
167 def stats(df):
168     '''
169     Statistical analysis summary of the experiment
170     (eg. Mean and Std of each column, box plot showing
171     data distribution of each column and correlation matrix of the dataframe)
172     '''
173     for i in range(1, len(df.columns)):

```

```

174         print(df.columns.values[i])
175         print('Mean: ', np.mean(df.iloc[:, i]), 'Std: ', np.std(df.iloc[:, i]))
176
177     df.iloc[:, 1:].plot.box(rot=0, fontsize=12, figsize=(11, 8), grid=True)
178     plt.show()
179
180     sns.heatmap(df.corr(), annot=True)
181
182
183 stats(df)
184
185 tot = (df['Selfish and Small'] + df['Selfish and Large'] + df['Cooperative and
186         Large'] + df['Cooperative and Small'])
187 plt.plot(df['Iteration Num'], (df['Selfish and Small'] + df['Selfish and Large'])/
188         tot, '0.75',
189         label = 'Large group size')
190 plt.plot(df['Iteration Num'], (df['Selfish and Large'] + df['Cooperative and Large
191         '])/tot, 'k--',
192         label = 'Selfish resource usage')
193 plt.xlim(0, 120)
194 plt.ylim(0, 0.8)
195 plt.xlabel("Generation")
196 plt.ylabel("Global frequency")
197 plt.legend()
198
199 plt.plot(df['Iteration Num'], df['Cooperative and Small']/tot, '0.75', label = '
200         Cooperative and Small')
201 plt.plot(df['Iteration Num'], df['Cooperative and Large']/tot, '--', color= '0.75'
202         , label = 'Cooperative and Large')
203 plt.plot(df['Iteration Num'], df['Selfish and Small']/tot, 'k', label = 'Selfish
204         and Small')
205 plt.plot(df['Iteration Num'], df['Selfish and Large']/tot, 'k--', label = 'Selfish
206         and Large')
207 plt.xlim(0, 120)
208 plt.ylim(0, 1)
209 plt.xlabel("Generation")
210 plt.ylabel("Global genotype frequency")
211 plt.legend()

```

Listing 1: Paper_4.py

Appendix E Code: Paper Extension

```
1 import pandas as pd
2 import numpy as np
3 import math
4 import random
5 import itertools
6 from functools import reduce
7 import matplotlib.pyplot as plt
8 import seaborn as sns
9
10
11 def create_pool(size):
12     '''
13     Creating an initial population containing 4 different types of
14     individuals:
15
16     SL: Selfish and Large
17     SS: Selfish and Small
18     SM: Selfish and Medium
19     CL: Cooperative and Large
20     CS: Cooperative and Small
21     CM: Cooperative and Medium
22     '''
23     res = np.repeat(['SL', 'SS', 'SM', 'CL', 'CS', 'CM'], size/6)
24     random.shuffle(res)
25     return res
26
27
28 def divide_in_groups(pool, large_g=40, small_g=4, medium_g=22):
29     '''
30     Dividing the current population into two main divisions: one containing
31     all the different types of large individuals and one containing instead
32     all the types of small individuals. These two divisions are additionally
33     splitted in multiple groups depending on the large_g and small_g parameters
34     which represent respectively the fixed group size that each large and small
35     group should have. If there are not enough individuals left to fill a group
36     they are automatically discarded.
37     '''
38     large = [ind for ind in pool if ind[1] == 'L']
39     small = [ind for ind in pool if ind[1] == 'S']
40     medium = [ind for ind in pool if ind[1] == 'M']
41     discard_large = int(large_g*(len(large)/large_g - math.floor(len(large)/
42     large_g)))
43     discard_small = int(small_g*(len(small)/small_g - math.floor(len(small)/
44     small_g)))
45     discard_medium = int(medium_g*(len(medium)/medium_g - math.floor(len(medium)/
46     medium_g)))
47     try:
48         groups_l = np.array(large[: len(large) - discard_large])
49         groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
50     except:
51         if groups_l.size == 0:
52             groups_s = np.array(small[: len(small) - discard_small])
53             groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
54             groups_m = np.array(medium[: len(medium) - discard_medium])
55             if groups_m.size != 0 and groups_m.size != 1:
56                 groups_m = groups_m.reshape(math.floor(len(medium)/medium_g), -1)
57             return groups_l, groups_s, groups_m
58         groups_l = np.array(large[: len(large) - discard_large - 1])
59         groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
60     try:
61         groups_m = np.array(medium[: len(medium) - discard_medium])
62         groups_m = groups_m.reshape(math.floor(len(medium)/medium_g), -1)
```

```

60 except:
61     if groups_m.size == 0:
62         groups_s = np.array(small[: len(small) - discard_small])
63         if groups_s.size != 0 and groups_s.size != 1:
64             groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
65         return groups_l, groups_s, groups_m
66     groups_m = np.array(medium[: len(medium) - discard_medium -1])
67     groups_m = groups_m.reshape(math.floor(len(medium)/medium_g), -1)
68 try:
69     groups_s = np.array(small[: len(small) - discard_small])
70     groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
71 except:
72     if groups_s.size == 0:
73         return groups_l, groups_s, groups_m
74     groups_s = np.array(small[: len(small) - discard_small -1])
75     groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
76 return groups_l, groups_s, groups_m
77
78
79 def reproduction(large_gs, small_gs, medium_gs, disposal_limit=4, large_r=50,
80                 small_r=4, medium_r=27, self_g=0.02, coop_g=0.018, self_c=0.2,
81                 coop_c=0.1, K=0.1):
82     '''
83     Reproduction takes place just within divisions and they are dependent
84     on the magnitude of the share of the total group resource that the
85     genotype receives and the replicator equations (shown above).
86     Therefore, the reproduction results are highly dependent of the disposal
87     time and the equations parameters.
88     '''
89     i = 0
90     large_g_res = [[]] * len(large_gs)
91     small_g_res = [[]] * len(small_gs)
92     medium_g_res = [[]] * len(medium_gs)
93     for large_g, small_g, medium_g in itertools.zip_longest(large_gs, small_gs,
94                                                             medium_gs):
95         if small_g is None and large_g is not None and medium_g is not None:
96             unique, counts = np.unique(large_g, return_counts=True)
97             large_counts = dict(zip(unique, counts))
98             disp_time = 0
99             large_coop_individuals = large_counts.get('CL', 0)
100             large_self_individuals = large_counts.get('SL', 0)
101             unique, counts = np.unique(medium_g, return_counts=True)
102             medium_counts = dict(zip(unique, counts))
103             medium_coop_individuals = medium_counts.get('CM', 0)
104             medium_self_individuals = medium_counts.get('SM', 0)
105             while disp_time != disposal_limit:
106                 large_coop_R_i = (large_coop_individuals * coop_g * coop_c) / (
107                     large_coop_individuals * coop_g * coop_c +
108                     large_self_individuals * self_g * self_c) * large_r
109                 large_self_R_i = (large_self_individuals * self_g * self_c) / (
110                     large_coop_individuals * coop_g * coop_c +
111                     large_self_individuals * self_g * self_c) * large_r
112                 large_coop_individuals = large_coop_individuals + large_coop_R_i /
113                 coop_c - K * large_coop_individuals
114                 large_self_individuals = large_self_individuals + large_self_R_i /
115                 self_c - K * large_self_individuals
116                 medium_coop_R_i = (medium_coop_individuals * coop_g * coop_c) / (
117                     medium_coop_individuals * coop_g * coop_c +
118                     medium_self_individuals * self_g * self_c) * medium_r
119                 medium_self_R_i = (medium_self_individuals * self_g * self_c) / (
120                     medium_coop_individuals * coop_g * coop_c +
121                     medium_self_individuals * self_g * self_c) * medium_r
122                 medium_coop_individuals = medium_coop_individuals +
123                 medium_coop_R_i / coop_c - K * medium_coop_individuals

```

```

115         medium_self_individuals = medium_self_individuals +
medium_self_R_i / self_c - K * medium_self_individuals
116         disp_time += 1
117         medium_g_res[i] = []
118         medium_g_res[i].extend(['CM'] * int(medium_coop_individuals))
119         medium_g_res[i].extend(['SM'] * int(medium_self_individuals))
120         large_g_res[i] = []
121         large_g_res[i].extend(['CL'] * int(large_coop_individuals))
122         large_g_res[i].extend(['SL'] * int(large_self_individuals))
123         i += 1
124         continue
125     if small_g is None and large_g is not None:
126         unique, counts = np.unique(large_g, return_counts=True)
127         large_counts = dict(zip(unique, counts))
128         disp_time = 0
129         large_coop_individuals = large_counts.get('CL', 0)
130         large_self_individuals = large_counts.get('SL', 0)
131         while disp_time != disposal_limit:
132             large_coop_R_i = (large_coop_individuals * coop_g * coop_c) / (
133                 large_coop_individuals * coop_g * coop_c +
large_self_individuals * self_g * self_c) * large_r
134             large_self_R_i = (large_self_individuals * self_g * self_c) / (
135                 large_self_individuals * self_g * self_c +
large_coop_individuals * coop_g * coop_c) * large_r
136             large_coop_individuals = large_coop_individuals + large_coop_R_i /
coop_c - K * large_coop_individuals
137             large_self_individuals = large_self_individuals + large_self_R_i /
self_c - K * large_self_individuals
138             disp_time += 1
139             large_g_res[i] = []
140             large_g_res[i].extend(['CL'] * int(large_coop_individuals))
141             large_g_res[i].extend(['SL'] * int(large_self_individuals))
142             i += 1
143             continue
144     if small_g is None and medium_g is not None:
145         unique, counts = np.unique(medium_g, return_counts=True)
146         medium_counts = dict(zip(unique, counts))
147         medium_coop_individuals = medium_counts.get('CM', 0)
148         medium_self_individuals = medium_counts.get('SM', 0)
149         while disp_time != disposal_limit:
150             medium_coop_R_i = (medium_coop_individuals * coop_g * coop_c) / (
151                 medium_coop_individuals * coop_g * coop_c +
medium_self_individuals * self_g * self_c) * medium_r
152             medium_self_R_i = (medium_self_individuals * self_g * self_c) / (
153                 medium_self_individuals * self_g * self_c +
medium_coop_individuals * coop_g * coop_c) * medium_r
154             medium_coop_individuals = medium_coop_individuals +
medium_coop_R_i / coop_c - K * medium_coop_individuals
155             medium_self_individuals = medium_self_individuals +
medium_self_R_i / self_c - K * medium_self_individuals
156             disp_time += 1
157             medium_g_res[i] = []
158             medium_g_res[i].extend(['CM'] * int(medium_coop_individuals))
159             medium_g_res[i].extend(['SM'] * int(medium_self_individuals))
160             i += 1
161             continue
162     if large_g is None and medium_g is not None:
163         unique, counts = np.unique(medium_g, return_counts=True)
164         medium_counts = dict(zip(unique, counts))
165         disp_time = 0
166         medium_coop_individuals = medium_counts.get('CM', 0)
167         medium_self_individuals = medium_counts.get('SM', 0)
168         unique, counts = np.unique(small_g, return_counts=True)
169         small_counts = dict(zip(unique, counts))

```



```

170         small_coop_individuals = small_counts.get('CS', 0)
171         small_self_individuals = small_counts.get('SS', 0)
172         while disp_time != disposal_limit:
173             medium_coop_R_i = (medium_coop_individuals * coop_g * coop_c) / (
174                 medium_coop_individuals * coop_g * coop_c +
medium_self_individuals * self_g * self_c) * medium_r
175             medium_self_R_i = (medium_self_individuals * self_g * self_c) / (
176                 medium_self_individuals * self_g * self_c +
medium_coop_individuals * coop_g * coop_c) * medium_r
177             medium_coop_individuals = medium_coop_individuals +
medium_coop_R_i / coop_c - K * medium_coop_individuals
178             medium_self_individuals = medium_self_individuals +
medium_self_R_i / self_c - K * medium_self_individuals
179             small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
180                 small_coop_individuals * coop_g * coop_c +
small_self_individuals * self_g * self_c) * small_r
181             small_self_R_i = (small_self_individuals * self_g * self_c) / (
182                 small_self_individuals * self_g * self_c +
small_coop_individuals * coop_g * coop_c) * small_r
183             small_coop_individuals = small_coop_individuals + small_coop_R_i /
coop_c - K * small_coop_individuals
184             small_self_individuals = small_self_individuals + small_self_R_i /
self_c - K * small_self_individuals
185             disp_time += 1
186
187         medium_g_res[i] = []
188         medium_g_res[i].extend(['CM'] * int(medium_coop_individuals))
189         medium_g_res[i].extend(['SM'] * int(medium_self_individuals))
190         small_g_res[i] = []
191         small_g_res[i].extend(['CS'] * int(small_coop_individuals))
192         small_g_res[i].extend(['SS'] * int(small_self_individuals))
193         i += 1
194         continue
195     if medium_g is None and large_g is None:
196         unique, counts = np.unique(small_g, return_counts=True)
197         small_counts = dict(zip(unique, counts))
198         disp_time = 0
199         small_coop_individuals = small_counts.get('CS', 0)
200         small_self_individuals = small_counts.get('SS', 0)
201         while disp_time != disposal_limit:
202             small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
203                 small_coop_individuals * coop_g * coop_c +
small_self_individuals * self_g * self_c) * small_r
204             small_self_R_i = (small_self_individuals * self_g * self_c) / (
205                 small_self_individuals * self_g * self_c +
small_coop_individuals * coop_g * coop_c) * small_r
206             small_coop_individuals = small_coop_individuals + small_coop_R_i /
coop_c - K * small_coop_individuals
207             small_self_individuals = small_self_individuals + small_self_R_i /
self_c - K * small_self_individuals
208             disp_time += 1
209
210         small_g_res[i] = []
211         small_g_res[i].extend(['CS'] * int(small_coop_individuals))
212         small_g_res[i].extend(['SS'] * int(small_self_individuals))
213         i += 1
214         continue
215     if medium_g is not None and large_g is not None:
216         unique, counts = np.unique(large_g, return_counts=True)
217         large_counts = dict(zip(unique, counts))
218         disp_time = 0
219         large_coop_individuals = large_counts.get('CL', 0)
220         large_self_individuals = large_counts.get('SL', 0)
221         unique, counts = np.unique(small_g, return_counts=True)

```

```

222     small_counts = dict(zip(unique, counts))
223     small_coop_individuals = small_counts.get('CS', 0)
224     small_self_individuals = small_counts.get('SS', 0)
225     unique, counts = np.unique(medium_g, return_counts=True)
226     medium_counts = dict(zip(unique, counts))
227     medium_coop_individuals = medium_counts.get('CM', 0)
228     medium_self_individuals = medium_counts.get('SM', 0)
229     while disp_time != disposal_limit:
230         large_coop_R_i = (large_coop_individuals * coop_g * coop_c) / (
231             large_coop_individuals * coop_g * coop_c +
232             large_self_individuals * self_g * self_c) * large_r
233         large_self_R_i = (large_self_individuals * self_g * self_c) / (
234             large_self_individuals * self_g * self_c +
235             large_coop_individuals * coop_g * coop_c) * large_r
236         large_coop_individuals = large_coop_individuals + large_coop_R_i /
237         coop_c - K * large_coop_individuals
238         large_self_individuals = large_self_individuals + large_self_R_i /
239         self_c - K * large_self_individuals
240         small_coop_R_i = (small_coop_individuals * coop_g * coop_c) / (
241             small_coop_individuals * coop_g * coop_c +
242             small_self_individuals * self_g * self_c) * small_r
243         small_self_R_i = (small_self_individuals * self_g * self_c) / (
244             small_self_individuals * self_g * self_c +
245             small_coop_individuals * coop_g * coop_c) * small_r
246         small_coop_individuals = small_coop_individuals + small_coop_R_i /
247         coop_c - K * small_coop_individuals
248         small_self_individuals = small_self_individuals + small_self_R_i /
249         self_c - K * small_self_individuals
250         medium_coop_R_i = (medium_coop_individuals * coop_g * coop_c) / (
251             medium_coop_individuals * coop_g * coop_c +
252             medium_self_individuals * self_g * self_c) * medium_r
253         medium_self_R_i = (medium_self_individuals * self_g * self_c) / (
254             medium_self_individuals * self_g * self_c +
255             medium_coop_individuals * coop_g * coop_c) * medium_r
256         medium_coop_individuals = medium_coop_individuals +
257         medium_coop_R_i / coop_c - K * medium_coop_individuals
258         medium_self_individuals = medium_self_individuals +
259         medium_self_R_i / self_c - K * medium_self_individuals
260         disp_time += 1
261
262         large_g_res[i] = []
263         large_g_res[i].extend(['CL'] * int(large_coop_individuals))
264         large_g_res[i].extend(['SL'] * int(large_self_individuals))
265         small_g_res[i] = []
266         small_g_res[i].extend(['CS'] * int(small_coop_individuals))
267         small_g_res[i].extend(['SS'] * int(small_self_individuals))
268         medium_g_res[i] = []
269         medium_g_res[i].extend(['CM'] * int(medium_coop_individuals))
270         medium_g_res[i].extend(['SM'] * int(medium_self_individuals))
271         i += 1
272         continue
273
274     return large_g_res, small_g_res, medium_g_res
275
276 def update_pool(large_gs, small_gs, medium_gs, mutation=False):
277     '''
278     In this case, are taken as input the two divisions which had undergone
279     reproduction and are merged together to create a pool like the one created
280     in the initialization step. In this situation, we need to make sure that
281     overall population size remains the same.
282     '''
283     res = reduce(lambda x,y :x+y ,large_gs, []) + reduce(lambda x,y :x+y ,
284         small_gs, []) + reduce(lambda x,y :x+y , medium_gs, [])

```

```

273 unique, counts = np.unique(res, return_counts=True)
274 pop_counts = dict(zip(unique, counts))
275 new_pop_elements = [int(((i/len(res))*pop)) for i in pop_counts.values()]
276 res = np.repeat([*pop_counts.keys()], new_pop_elements).tolist()
277 random.shuffle(res)
278 if mutation is True:
279     for i in range(0, int(len(res)/2)):
280         res[i] = (np.random.choice(['C' + res[i][1], 'S' + res[i][1]],
281                                   p=[1/2, 1/2]))
282     return res
283
284
285 pop = 4000
286 iter_num = 150
287 d = {'Iteration Num': [0], 'Selfish and Large': [666], 'Selfish and Small': [666],
288      'Selfish and Medium': [666],
289      'Cooperative and Large': [666], 'Cooperative and Small': [666], 'Cooperative
290      and Medium': [666]}
291 df = pd.DataFrame(data=d)
292 migrant_pool = create_pool(pop)
293 for i in range(1, iter_num + 1):
294     large_group, small_group, medium_group = divide_in_groups(migrant_pool)
295     large_group, small_group, medium_group = reproduction(large_group, small_group
296     , medium_group)
297     migrant_pool = update_pool(large_group, small_group, medium_group)
298     unique, counts = np.unique(migrant_pool, return_counts=True)
299     res_counts = dict(zip(unique, counts))
300     df = df.append({"Iteration Num": i,
301                    "Selfish and Large": res_counts.get('SL', 0),
302                    "Selfish and Small": res_counts.get('SS', 0),
303                    "Selfish and Medium": res_counts.get('SM', 0),
304                    "Cooperative and Large": res_counts.get('CL', 0),
305                    "Cooperative and Small": res_counts.get('CS', 0),
306                    "Cooperative and Medium": res_counts.get('CM', 0)
307                    }, ignore_index=True)
308
309     if i % 50 == 0:
310         print('Iteration Number:', i)

```

Listing 2: Three_groups_and_mutation.py

```

1 import pandas as pd
2 import numpy as np
3 import math
4 import random
5 import itertools
6 from functools import reduce
7 import matplotlib.pyplot as plt
8 import seaborn as sns
9
10
11 def create_pool(size):
12     '''
13     Creating an initial population containing 4 different types of
14     individuals:
15
16     0 -> Selfish
17     1 -> Quite Selfish
18     2 -> Quite Cooperative
19     3 -> Cooperative
20     '''
21     res = np.repeat(['0L', '0S', '1L', '1S', '2L', '2S', '3L', '3S'], size/4)
22     random.shuffle(res)
23     return res
24
25

```

```

26 def divide_in_groups(pool, large_g=40, small_g=4):
27     '''
28     Dividing the current population into two main divisions: one containing
29     all the different types of large individuals and one containing instead
30     all the types of small individuals. These two divisions are additionally
31     splitted in multiple groups depending on the large_g and small_g parameters
32     which represent respectively the fixed group size that each large and small
33     group should have. If there are not enough individuals left to fill a group
34     they are automatically discarded.
35     '''
36     large = [ind for ind in pool if ind[1] == 'L']
37     small = [ind for ind in pool if ind[1] == 'S']
38     discard_large = int(large_g*(len(large)/large_g - math.floor(len(large)/
39     large_g)))
40     discard_small = int(small_g*(len(small)/small_g - math.floor(len(small)/
41     small_g)))
42     try:
43         groups_l = np.array(large[: len(large) - discard_large])
44         groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
45     except:
46         if groups_l.size == 0:
47             groups_s = np.array(small[: len(small) - discard_small])
48             groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
49             return groups_l, groups_s
50         groups_l = np.array(large[: len(large) - discard_large - 1])
51         groups_l = groups_l.reshape(math.floor(len(large)/large_g), -1)
52     try:
53         groups_s = np.array(small[: len(small) - discard_small])
54         groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
55     except:
56         if groups_s.size == 0:
57             return groups_l, groups_s
58         groups_s = np.array(small[: len(small) - discard_small - 1])
59         groups_s = groups_s.reshape(math.floor(len(small)/small_g), -1)
60     return groups_l, groups_s
61
62 def reproduction(large_gs, small_gs, disposal_limit=4, large_r=50, small_r=4,
63 self_g=0.02, coop_g=0.018, self_c=0.2,
64 coop_c=0.1, K=0.1):
65     '''
66     Reproduction takes place just within divisions and they are dependent
67     on the magnitude of the share of the total group resource that the
68     genotype receives and the replicator equations (shown above).
69     Therefore, the reproduction results are highly dependent of the disposal
70     time and the equations parameters.
71     '''
72     i = 0
73     large_g_res = [[]] * len(large_gs)
74     small_g_res = [[]] * len(small_gs)
75     #print(len(large_gs), len(small_gs))
76     for large_g, small_g in itertools.zip_longest(large_gs, small_gs):
77         if large_g is None and small_g is not None:
78             unique, counts = np.unique(small_g, return_counts=True)
79             small_counts = dict(zip(unique, counts))
80             disp_time = 0
81             small_1 = small_counts.get('0S', 0)
82             small_2 = small_counts.get('1S', 0)
83             small_3 = small_counts.get('2S', 0)
84             small_4 = small_counts.get('3S', 0)
85             dem = (small_1 * self_g * self_c + small_2 * ((self_g/1.2) * (self_c
86             /1.2)) + small_3 * ((self_g/1.4) * (self_c/1.4)) + small_4 * coop_g * coop_c)
87             while disp_time != disposal_limit:
88                 small_1_R_i = (small_1 * self_g * self_c) / dem * small_r

```

```

86         small_2_R_i = (small_2 * ((self_g/1.2) * (self_c/1.2))) / dem *
small_r
87         small_3_R_i = (small_3 * ((self_g/1.4) * (self_c/1.4))) / dem *
small_r
88         small_4_R_i = (small_4 * coop_g * coop_c) / dem * small_r
89         small_1 = small_1 + small_1_R_i / self_c - K * small_1
90         small_2 = small_2 + (small_2_R_i / (self_c/1.2)) - K * small_2
91         small_3 = small_3 + (small_3_R_i / (self_c/1.4)) - K * small_3
92         small_4 = small_4 + small_4_R_i / coop_c - K * small_4
93         disp_time += 1
94
95         small_g_res[i] = []
96         small_g_res[i].extend(['0S'] * int(small_1))
97         small_g_res[i].extend(['1S'] * int(small_2))
98         small_g_res[i].extend(['2S'] * int(small_3))
99         small_g_res[i].extend(['3S'] * int(small_4))
100        i += 1
101        elif small_g is None and large_g is not None:
102            disp_time = 0
103            unique, counts = np.unique(large_g, return_counts=True)
104            large_counts = dict(zip(unique, counts))
105            large_1 = large_counts.get('0L', 0)
106            large_2 = large_counts.get('1L', 0)
107            large_3 = large_counts.get('2L', 0)
108            large_4 = large_counts.get('3L', 0)
109            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)
110            while disp_time != disposal_limit:
111                large_1_R_i = (large_1 * self_g * self_c) / dem2 * large_r
112                large_2_R_i = (large_2 * ((self_g/1.2) * (self_c/1.2))) / dem2 *
large_r
113                large_3_R_i = (large_3 * ((self_g/1.4) * (self_c/1.4))) / dem2 *
large_r
114                large_4_R_i = (large_4 * coop_g * coop_c) / dem2 * large_r
115                large_1 = large_1 + large_1_R_i / self_c - K * large_1
116                large_2 = large_2 + (large_2_R_i / (self_c/1.2)) - K * large_2
117                large_3 = large_3 + (large_3_R_i / (self_c/1.4)) - K * large_3
118                large_4 = large_4 + large_4_R_i / self_c - K * large_4
119                disp_time += 1
120
121            large_g_res[i] = []
122            large_g_res[i].extend(['0L'] * int(large_1))
123            large_g_res[i].extend(['1L'] * int(large_2))
124            large_g_res[i].extend(['2L'] * int(large_3))
125            large_g_res[i].extend(['3L'] * int(large_4))
126            i += 1
127        else:
128            unique, counts = np.unique(small_g, return_counts=True)
129            small_counts = dict(zip(unique, counts))
130            disp_time = 0
131            small_1 = small_counts.get('0S', 0)
132            small_2 = small_counts.get('1S', 0)
133            small_3 = small_counts.get('2S', 0)
134            small_4 = small_counts.get('3S', 0)
135            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)
136            unique, counts = np.unique(large_g, return_counts=True)
137            large_counts = dict(zip(unique, counts))
138            large_1 = large_counts.get('0L', 0)
139            large_2 = large_counts.get('1L', 0)
140            large_3 = large_counts.get('2L', 0)
141            large_4 = large_counts.get('3L', 0)
142            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)

```

```

143         while disp_time != disposal_limit:
144             large_1_R_i = (large_1 * self_g * self_c) / dem2 * large_r
145             large_2_R_i = (large_2 * ((self_g/1.2) * (self_c/1.2))) / dem2 *
large_r
146             large_3_R_i = (large_3 * ((self_g/1.4) * (self_c/1.4))) / dem2 *
large_r
147             large_4_R_i = (large_4 * coop_g * coop_c) / dem2 * large_r
148             large_1 = large_1 + large_1_R_i / self_c - K * large_1
149             large_2 = large_2 + (large_2_R_i / (self_c/1.2)) - K * large_2
150             large_3 = large_3 + (large_3_R_i / (self_c/1.4)) - K * large_3
151             large_4 = large_4 + large_4_R_i / self_c - K * large_4
152             small_1_R_i = (small_1 * self_g * self_c) / dem * small_r
153             small_2_R_i = (small_2 * ((self_g/1.2) * (self_c/1.2))) / dem *
small_r
154             small_3_R_i = (small_3 * ((self_g/1.4) * (self_c/1.4))) / dem *
small_r
155             small_4_R_i = (small_4 * coop_g * coop_c) / dem * small_r
156             small_1 = small_1 + small_1_R_i / self_c - K * small_1
157             small_2 = small_2 + (small_2_R_i / (self_c/1.2)) - K * small_2
158             small_3 = small_3 + (small_3_R_i / (self_c/1.4)) - K * small_3
159             small_4 = small_4 + small_4_R_i / coop_c - K * small_4
160             disp_time += 1
161
162             large_g_res[i] = []
163             large_g_res[i].extend(['0L'] * int(large_1))
164             large_g_res[i].extend(['1L'] * int(large_2))
165             large_g_res[i].extend(['2L'] * int(large_3))
166             large_g_res[i].extend(['3L'] * int(large_4))
167             small_g_res[i] = []
168             small_g_res[i].extend(['0S'] * int(small_1))
169             small_g_res[i].extend(['1S'] * int(small_2))
170             small_g_res[i].extend(['2S'] * int(small_3))
171             small_g_res[i].extend(['3S'] * int(small_4))
172             i += 1
173
174         return large_g_res, small_g_res
175
176
177 def update_pool(large_gs, small_gs, mutation=False):
178     '''
179     In this case, are taken as input the two divisions which had undergone
180     reproduction and are merged together to create a pool like the one created
181     in the initialization step. In this situation, we need to make sure that
182     overall population size remains the same.
183     '''
184     res = reduce(lambda x,y :x+y ,large_gs, []) + reduce(lambda x,y :x+y ,
small_gs, [])
185     unique, counts = np.unique(res, return_counts=True)
186     pop_counts = dict(zip(unique, counts))
187     new_pop_elements = [int(((i/len(res))*pop)) for i in pop_counts.values()]
188     res = np.repeat([*pop_counts.keys()], new_pop_elements).tolist()
189     random.shuffle(res)
190     if mutation is True:
191         for i in range(0, int(len(res)/2)):
192             res[i] = (np.random.choice(['0' + res[i][1], '1' + res[i][1], '2' +
res[i][1], '3' + res[i][1]],
193                                     p=[1/4, 1/4, 1/4, 1/4]))
194     return res
195
196
197 pop = 4000
198 iter_num = 120
199 d = {'Iteration Num': [0], '0Large': [500], '1Large': [500],
200      '2Large': [500], '3Large': [500], '0Small': [500],

```

```

201     '1Small': [500], '2Small': [500], '3Small': [500]}
202 df = pd.DataFrame(data=d)
203 migrant_pool = create_pool(pop)
204 for i in range(1, iter_num + 1):
205     large_group, small_group = divide_in_groups(migrant_pool)
206     large_group, small_group = reproduction(large_group, small_group)
207     migrant_pool = update_pool(large_group, small_group, mutation=True)
208     unique, counts = np.unique(migrant_pool, return_counts=True)
209     res_counts = dict(zip(unique, counts))
210     df = df.append({"Iteration Num": i,
211                    "0Large": res_counts.get('0L', 0),
212                    "1Large": res_counts.get('1L', 0),
213                    "2Large": res_counts.get('2L', 0),
214                    "3Large": res_counts.get('3L', 0),
215                    "0Small": res_counts.get('0S', 0),
216                    "1Small": res_counts.get('1S', 0),
217                    "2Small": res_counts.get('2S', 0),
218                    "3Small": res_counts.get('3S', 0),
219                    }, ignore_index=True)
220 if i % 5 == 0:
221     print('Iteration Number:', i)

```

Listing 3: Including_levels_of_selfishness.py