Cooperative and Selfish Group Evolvability

Dimitri Diomaiuta

University of Southampton

Abstract. In this paper we analyze and reproduce the experiments discussed in the paper *Individual Selection For Cooperative Group Formation* [1]. In the first part of this paper we discuss and reimplement the model described in [1] that aims at evolving cooperative and selfish traits. In the second part we extend the research by applying the niche construction process to the t parameter, the time spent for within group reproduction.

1 Introduction

The biological inspiration of the research is bacteria growth. In this biological scenario we can observe the different types of bacteria that, under limited resources, behave in a cooperative way (growing slower and consuming less resources) or in a selfish one (growing faster and consuming more resources). The originality of Powers research [1] lies in the fact that cooperation is not favoured by environmental conditions. Techniques like spatially structured population, described in [2–5], and not sharing resources with the rest of the group are avoided. Another approach that is avoided in the paper is the trait group aggregation shown in [6–8]. Trait group aggregation favours the cooperators by exploiting the differential group productivity. Other environmental settings that facilitates the growth of the cooperative trait is shown in [7–9]. Here the individuals with similar genetic traits are grouped together, promoting the cooperative trait. The main research point of the paper is, hence, to observe which genetic trait will emerge when environmental conditions that favours cheaters or cooperator are evolved as well.

The technique used to evolve the environmental settings is niche construction [10]. The model is composed by individuals that specify two genetic traits:

- Strategy trait: cooperative or selfish
- Group size trait: large or small

Following the settings of the model we can see that there are four different kind of individuals: *Cooperative + Large*, *Cooperative + Small*, *Selfish + Large* and *Selfish + Small*. The model settings favours large groups, having a larger assigned resource, and selfish ones, having a faster growth rate. This should lead *Selfish + large* to win over the other species. Instead, when all the four possible individual types are present, we can observe *Cooperative + Small* reaching fixation.

2 Model Details

We describe in this section the settings of the model we reproduced. As Powers describes [1] the model has a population of size N where the 4 types of individuals have, initially, an equal distribution. The strategy genotype specifies the amount of resources that an individual consumes and its growth rate. The group size genotype specifies the type of group that the individual will be part of. After the initialization phase the main routine of the program starts. In this routine the individuals of the initial population are in one group, called migrant pool. The second phase, called aggregation, corresponds to randomly divide individuals into groups according to their group size genotype, discarding the unassigned individuals. The third phase is about performing reproduction within the groups, as described in subsection 2.1. In Powers' model reproduction occurs only within groups for t generations. After the reproduction step the offspring joins again the migrant pool, this is the fourth step. The last step is about rescaling the population back to its original size N. This is done by maintaining the proportion between the individuals. This routine continues for T generations.

2.1 reproduction step

Reproduction, in this model, depends on R, a limited resource influx that each group is given. The amount of resource depends on the group size. Large groups have a 5% per capita more resource than their smaller counterpart. Reproduction, in this model, happen asexually and the offspring are not affected by mutation. The offspring size for each individual type is determined only by the consumption and growth rate of their parents given a limited resource influx and the distribution of other individuals inside their group. The amount of R that each genotype receives is given by equation 1.

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

The genotype i receives r_i resource share. n_i represents the number of individuals of genotype i in the current generation and G_i and C_i represents respectively the growth and consumption rate. At the denominator, genotype j represents all the other genotypes, i included. Given the resource share of a genotype for a given group, its offspring size is determined by the replicator equation 2.

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

The current generation is represented by t. The replicator equation 2 introduces also a constant mortality term represented by K.

3 Implementation reproduction

In this section we discuss the reimplementation of Powers' research [1] and the results we obtained.

3.1 Implementation details

The first step towards reproduction was to gather all the parameters used throughout the paper. We were able to reproduce the paper using the exact same parameter used by Powers. The parameters that were missing from the original paper, like the resource share assigned to the groups and the death rate, were given by the University of Southampton through assignment specifications. Table 1 contains all the parameters used for our reimplementation. The program

Table 1. The parameters settings of our reproduced model.

Parameters	**Value**
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
Small group size	4
Large group size	40
Small group resource	4
Large group resource	50
Number of generations within group t	4
Death rate K	0.1

we reimplemented consists of four Java classes. The *Individual* object (see code at appendix 6.2) contains the two genotypes and is used to represents the bacteria. The *IndividualFrequencies* object (see code at appendix 6.3) represents the frequencies for a given individual type (among the four possible types). The *GroupFrequencies* object (see code at appendix 6.4) describes the frequencies of all the four types for a given aggregation. Lastly, the *Main* class of the program (see code at appendix 6.1) is responsible for the aggregation in groups, the reproduction, the dispersal and the normalization steps. We reimplemented the resource allocation (1) and the replicator equation (2) exactly how reported in the original paper. The only relevant implementation choice that we took independently from the original paper, due to not appearing in Powers' research, was the rounding of the offspring. Since the replicator equation (2) is not producing integers, after some testing, we decided to round the offspring during the normalization step.

3.2 Results

In this section we analyze the reimplementation results. Figure 1 summarizes the obtained results. The code that produced figure 1 can be found in appendix 6.7. Our reimplementation, as the results confirm, works as expected and reproduces

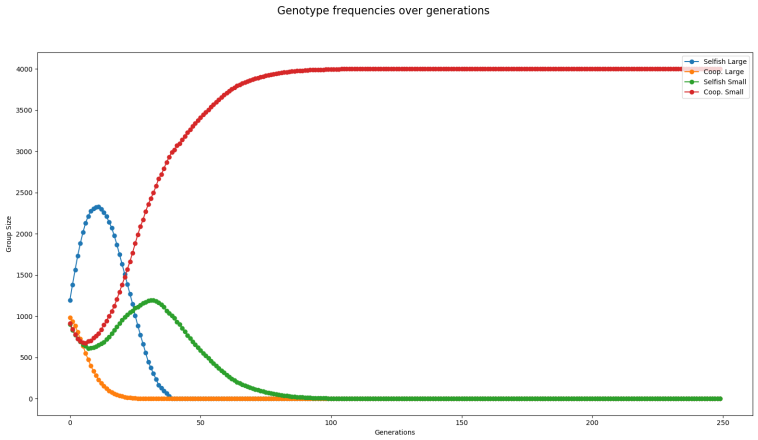

Fig. 1. The reimplementation results.

the results obtained by Powers. We can observe that in our reimplementation as well the individual type *Cooperators + Small* wins over the other species. We can see that, around generation 20, the selfish large starts diminishing and both the individuals with the small group gene start rising. Once the *Selfish + Large* species is extinct the *Cooperators + Small* wins over the other small group individual type. The *Selfish + Large* individuals extinction is caused because, after driving the *Cooperators + Large* species extinct, they loose the advantage of consuming the cooperators resources. Meanwhile between the small species the cooperators win over the cheaters since they have an advantage when group size is 4 and the generations spent reproducing within the group is 4.

4 Extension

In this section we consider an extension to the Powers' research. In the conclusion section Powers states that future research should consider applying the process of niche construction to other parameters as well. We decided to apply niche construction to t, the time spent for the within group reproduction step. We want to test whether the original model environmental assumption about t could have played a role in facilitating a group proliferation over the others.

4.1 Design

The first step was to modify Powers model in order to meet our new research question. We introduced a new gene that expresses the preferred time value t for within group reproduction. The initialization phase changes in such a way that all the possible individuals types, according to this new gene, have the same equal distribution. During the aggregation we draw groups from random

individuals specifying the same group size and time gene. The dispersal and the normalization phase remain the same as in the original model.

4.2 Implementation

We implemented this extension part by modifying the Main class of the existing Java sources written to test Powers' research reproducibility (see code at appendix 6.5). We used the exact same parameters described in table 1, with the exception of the t value. For testing purposes, we limited t to take only two different values per execution. This means that every execution will consist of eight species, four with t_1 value and four with t_2 value for the t gene. We implemented two different migrant pools according to the t gene. This, however, does not affect the results since the normalization phase rescales the distribution like there is only one population.

4.3 Results

In this section we analyze the results obtained by testing different values for the t gene against each other. We first consider the case when $t_1 = 4$ and $t_2 = 5$. Figures 2 and 3 summarize the results. We can observe that giving the same parameters we obtain two different results. This happens because of the randomness of the aggregation phase. Since reproduction is applied only within groups, how these groups are formed can influence the number of the species' offspring. The important data to analyze, however, is not only the genotype that reaches fixation, but how the small groups with $t = 5$ are behaving. We can see that in both cases the *Selfish + Small* + t_5 drive the *Cooperative + Small* + t_5 type extinct. We can see the same pattern in figures 4 and 5. When $t <= 4$ in small groups the cooperators are favoured, but when $t > 4$ the cheaters have an higher growth in the small groups. This indicates that the fixed environmental value of t in Powers research played a role in facilitating the small cooperators to emerge against the small selfish individuals. Furthermore, we can see the model limitations in the fact that, as seen in all the aforementioned figures (2, 3, 4, 5), applying only reproduction between groups facilitates the individuals with a larger t value.

5 Conclusion

In the extension part we have investigated whether applying the niche construction process in regards to the t parameter, specifying the time spent for within group reproduction, could reveal some environmental assumption about the model. We discovered that the model described by Powers works as expected only when $t <= 4$. We also observed the model's limits regarding the reproduction step. Applying reproduction only in between groups causes the species with a larger t trait to reach fixation.

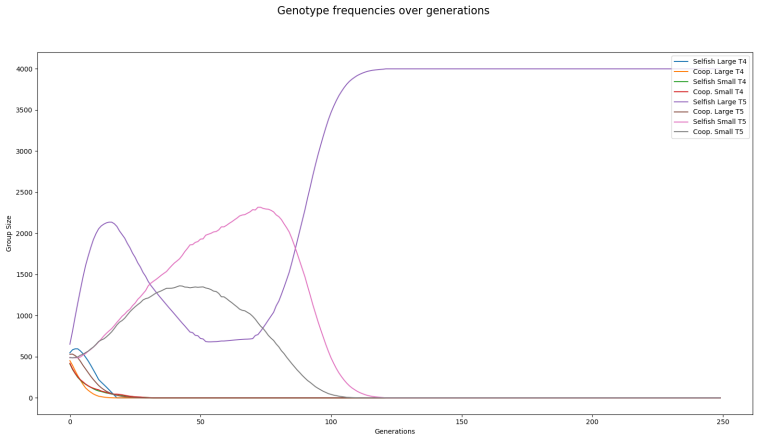

Fig. 2. Population with $t_1 = 4$ and $t_2 = 5$.

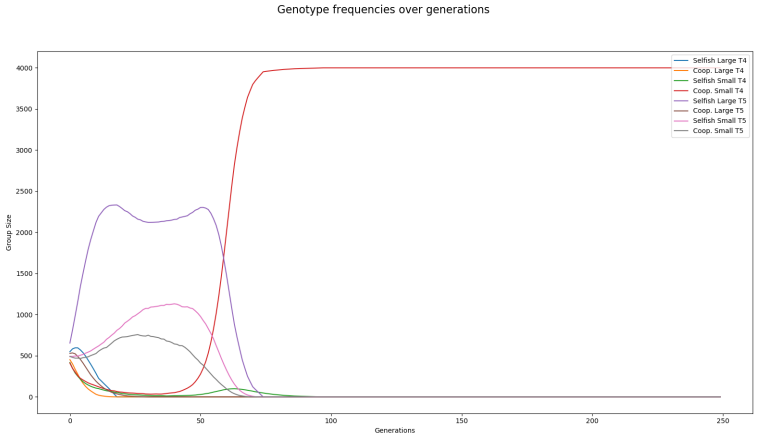

Fig. 3. Population with $t_1 = 4$ and $t_2 = 5$.

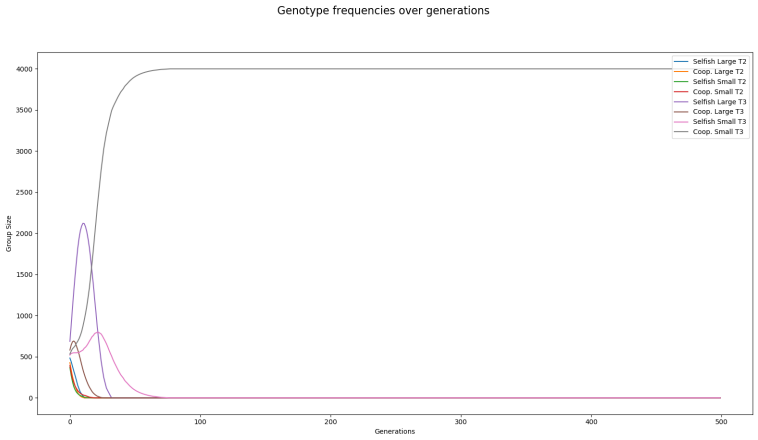

Fig. 4. Population with $t_1 = 2$ and $t_2 = 3$.

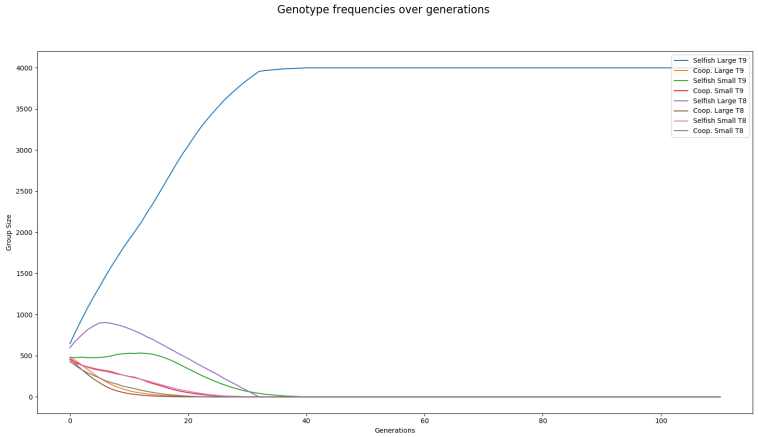

Fig. 5. Population with $t_1 = 9$ and $t_2 = 8$.

References

1. Powers, S.T., Penn, A.S. and Watson, R.A., 2007, September. Individual selection for cooperative group formation. In European Conference on Artificial Life (pp. 585-594). Springer, Berlin, Heidelberg.
2. Pfeiffer, T., Schuster, S., Bonhoeffer, S.: Cooperation and competition in the evolution of ATP-producing pathways. Science 292(5516) (2001) 504507
3. Pfeiffer, T., Bonhoeffer, S.: An evolutionary scenario for the transition to undifferentiated multicellularity. PNAS 100(3) (2003) 10951098
4. Kreft, J.U.: Biofilms promote altruism. Microbiology 150 (2004) 27512760
5. Nowak, M.A., May, R.M.: The spatial dilemmas of evolution. International Journal of Bifurcation and Chaos 3(1) (1993) 3578
6. Wilson, D.S.: A theory of group selection. PNAS 72(1) (1975) 143146
7. Wilson, D.S.: The Natural Selection of Populations and Communities. Benjamin/Cummings (1980)
8. Sober, E., Wilson, D.S.: Unto Others: The Evolution and Psychology of Unselfish Behavior. Harvard University Press, Cambridge, MA (1998)
9. Wilson, D.S., Dugatkin, L.A.: Group selection and assortative interactions. The American Naturalist 149(2) (1997) 336351
10. Odling-Smee, F.J., Laland, K.N., Feldman, M.W.: Niche construction: the neglected process in evolution. Monographs in population biology; no. 37. Princeton University Press (2003)

6 Appendix A: source code

This appendix section contains the source code of the program:

- Compile with: *javac *.java*
- Run with: *java Main*
- To plot the results run with: *java Main | xargs python3 plotBacteria.py*

6.1 Main.java

```
import java.lang.reflect.Array;
import java.math.MathContext;
import java.util.ArrayList;
import java.util.Collections;

public class Main {

    private static final int POPULATION_SIZE = 4000;
    private static final int GENERATIONS = 1000;
    private static final int TIME_IN_GROUP = 4;
    private static final int SMALL_GROUP_SIZE = 4;
    private static final int LARGE_GROUP_SIZE = 40;
    private static final double SMALL_GROUP_RESOURCE = 4;
    private static final double LARGE_GROUP_RESOURCE = 50;
```

```
private static final double DEATH_RATE = 0.1;
private static final Individual SELFISH_SMALL = new Individual
    (true, false);
private static final Individual SELFISH_LARGE = new Individual
    (true, true);
private static final Individual COOPERATIVE_SMALL = new
    Individual(false, false);
private static final Individual COOPERATIVE_LARGE = new
    Individual(false, true);
private static final Individual [] ALL_INDIVIDUALS = new
    Individual[] {SELFISH_LARGE, COOPERATIVE_LARGE,
    SELFISH_SMALL, COOPERATIVE_SMALL};

private static ArrayList<Individual> generatePopulation() {
    ArrayList<Individual> population = new ArrayList<>();
    for (int i = 0; i < POPULATION_SIZE / ALL_INDIVIDUALS.
        length; i++) {
        for (Individual individual : ALL_INDIVIDUALS) {
            population.add(individual.clone());
        }
    }
    return population;
}

private static ArrayList<ArrayList> computeGroups(ArrayList<
    Individual> population, int groupSize, boolean largeGroup)
     {
    ArrayList<ArrayList> allGroups = new ArrayList<>();
    ArrayList<Individual> group = new ArrayList<>();
    for (Individual individual : population) {
        if(individual.isLargeGroupGene() == largeGroup) {
            group.add(individual.clone());
        }
        if(group.size() == groupSize) {
            allGroups.add(group);
            group = new ArrayList<>();
        }
    }
    return allGroups;
}

private static double computeNextGeneration(double
    thisGeneSize, double thisGeneGrowth, double
    thisGeneConsumption,
```

```
                                        double thatGeneSize,
                                            double thatGeneGrowth
                                            , double
                                            thatGeneConsumption,
                                            double resources) {
        double resourceShare = ( (thisGeneSize * thisGeneGrowth *
            thisGeneConsumption) /
                (thatGeneSize * thatGeneGrowth *
                    thatGeneConsumption + thisGeneSize *
                    thisGeneGrowth * thisGeneConsumption) ) *
                    resources;
        return thisGeneSize + (resourceShare / thisGeneConsumption
            ) - DEATH_RATE * thisGeneSize;
    }

    private static ArrayList<GroupFrequencies>
        translateGroupsToFrequencies(ArrayList<ArrayList> groups,
        Individual cheater, Individual cooperator) {
        ArrayList<GroupFrequencies> result = new ArrayList<>();
        for (ArrayList<Individual> group : groups) {
            result.add(new GroupFrequencies(new
                IndividualFrequencies(Collections.frequency(group,
                cheater), cheater), new IndividualFrequencies(
                Collections.frequency(group, cooperator),
                cooperator)));
        }
        return result;
    }

    private static ArrayList<Individual>
        generatePopulationGivenFrequencies(IndividualFrequencies
        ... individualFrequencies) {
        ArrayList<Individual> result = new ArrayList<>();
        for (IndividualFrequencies ind : individualFrequencies) {
            for (int i = 0; i < ind.getFrequency(); i++) {
                result.add(ind.getIndividualType().clone());
            }
        }
        return result;
    }

    private static GroupFrequencies reproductionStep(ArrayList<
        ArrayList> groups, boolean largeGroup) {
        Individual cheater = largeGroup ? SELFISH_LARGE :
            SELFISH_SMALL;
```

```
Individual cooperator = largeGroup ? COOPERATIVE_LARGE :
    COOPERATIVE_SMALL;
ArrayList<GroupFrequencies> groupsWithFrequencies =
    translateGroupsToFrequencies(groups, cheater,
    cooperator);
double groupResources = largeGroup ? LARGE_GROUP_RESOURCE
    : SMALL_GROUP_RESOURCE;
double totalNumberOfCheaters = 0;
double totalNumberOfCooperators = 0;
for (int i = 0; i < TIME_IN_GROUP; i++) {
    totalNumberOfCheaters = 0;
    totalNumberOfCooperators = 0;
    int index = 0;
    for (GroupFrequencies group : groupsWithFrequencies) {
        double numberOfCheaters = group.getSizeOfGenotype(
            cheater);
        double numberOfCooperators = group.
            getSizeOfGenotype(cooperator);
        double nextNumberOfCheaters = computeNextGeneration
            (numberOfCheaters, Individual.GROWTH_SELFISH,
            Individual.CONSUMPTION_SELFISH,
               numberOfCooperators, Individual.
                   GROWTH_COOPERATIVE, Individual.
                   CONSUMPTION_COOPERATIVE, groupResources)
                   ;
        double nextNumberOfCooperators =
            computeNextGeneration(numberOfCooperators,
            Individual.GROWTH_COOPERATIVE, Individual.
            CONSUMPTION_COOPERATIVE,
               numberOfCheaters, Individual.GROWTH_SELFISH,
                    Individual.CONSUMPTION_SELFISH,
                   groupResources);
        groupsWithFrequencies.set(index, new
            GroupFrequencies(new IndividualFrequencies(
            nextNumberOfCheaters, cheater), new
            IndividualFrequencies(nextNumberOfCooperators,
            cooperator)));
        index++;
        totalNumberOfCheaters += nextNumberOfCheaters;
        totalNumberOfCooperators += nextNumberOfCooperators
            ;
    }
}
return new GroupFrequencies(new IndividualFrequencies(
    totalNumberOfCheaters, cheater), new
```

```
            IndividualFrequencies(totalNumberOfCooperators,
            cooperator));
    }

    private static void printGenerations(ArrayList<
        GroupFrequencies> generations) {
        for (GroupFrequencies group : generations) {
            for (Individual ind : ALL_INDIVIDUALS) {
                System.out.printf("%.0f␣", group.
                    getIndividualFrequency(ind).getFrequency());
            }
        }
    }

    public static void main(String[] args) {
        // 1: Generate population of N individuals. Equiprobable
            distribution of genotypes.
        ArrayList<Individual> population = generatePopulation();
        ArrayList<GroupFrequencies> allGenerationsFrequencies =
            new ArrayList<>();
        for (int i = 0; i < GENERATIONS / TIME_IN_GROUP; i++) {
            Collections.shuffle(population);
            // 2: Divide into groups
            ArrayList<ArrayList> smallGroups = computeGroups(
                population, SMALL_GROUP_SIZE, false);
            ArrayList<ArrayList> largeGroups = computeGroups(
                population, LARGE_GROUP_SIZE, true);
            // 3: Reproduction step
            GroupFrequencies largeGroupsFrequencies =
                reproductionStep(largeGroups, true);
            GroupFrequencies smallGroupFrequencies =
                reproductionStep(smallGroups, false);
            // 4: Return progeny to the migrant pool
            GroupFrequencies currentGeneration =
                smallGroupFrequencies.mergeWithGroup(
                largeGroupsFrequencies);
            currentGeneration.rescaleGroup(POPULATION_SIZE);
            allGenerationsFrequencies.add(currentGeneration);
            // 6: Repeat from step 2 for N generations.
            population = generatePopulationGivenFrequencies(
                currentGeneration.getFrequencies());
        }
        printGenerations(allGenerationsFrequencies);
    }
}
```

6.2 Individual.java

```
import java.util.Objects;

public class Individual {

    public static final double GROWTH_COOPERATIVE = 0.018;
    public static final double GROWTH_SELFISH = 0.02;
    public static final double CONSUMPTION_COOPERATIVE = 0.1;
    public static final double CONSUMPTION_SELFISH = 0.2;
    private boolean selfishGene;
    private boolean largeGroupGene;

    public Individual() {

    }

    public Individual(boolean selfishGene, boolean largeGroupGene)
         {
        this.selfishGene = selfishGene;
        this.largeGroupGene = largeGroupGene;
    }

    @Override
    public String toString() {
        return "Individual{" +
                "selfishGene=" + selfishGene +
                ",␣largeGroupGene=" + largeGroupGene +
                '}';
    }

    public boolean isSelfishGene() {
        return selfishGene;
    }

    public boolean isLargeGroupGene() {
        return largeGroupGene;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Individual that = (Individual) o;
        return selfishGene == that.isSelfishGene() &&
                largeGroupGene == that.isLargeGroupGene();
```

```
    }

    @Override
    public int hashCode() {
        return Objects.hash(selfishGene, largeGroupGene);
    }

    public Individual clone() {
        return new Individual(this.isSelfishGene(), this.
            isLargeGroupGene());
    }

}
```

6.3 IndividualFrequencies.java

```
public class IndividualFrequencies {

    private Individual individualType;
    private double frequency;

    public IndividualFrequencies() {

    }

    public IndividualFrequencies(double frequency, Individual
        individualType) {
        this.individualType = individualType;
        this.frequency = frequency;
    }

    public Individual getIndividualType() {
        return individualType;
    }

    public double getFrequency() {
        return frequency;
    }

    public void setFrequency(long frequency) {
        this.frequency = frequency;
    }

    @Override
    public String toString() {
        return "IndividualFrequencies{" +
```

```
            "individualType=" + individualType +
            ",␣frequency=" + frequency +
            '}';
    }

    public IndividualFrequencies clone() {
        return new IndividualFrequencies(this.frequency,
            individualType.clone());
    }
}
```

6.4 GroupFrequencies.java

```
public class GroupFrequencies {
    private IndividualFrequencies[] groupFrequencies;

    public GroupFrequencies() {

    }

    public GroupFrequencies(IndividualFrequencies ...
        groupFrequencies) {
        this.groupFrequencies = groupFrequencies;
    }

    public int getSize() {
        int result = 0;
        for (IndividualFrequencies ind : groupFrequencies) {
            result += ind.getFrequency();
        }
        return result;
    }

    public double getSizeOfGenotype(Individual individual) {
        for (IndividualFrequencies ind : groupFrequencies) {
            if (ind.getIndividualType().equals(individual)) {
                return ind.getFrequency();
            }
        }
        return 0;
    }

    public int getNumberOfGenotypes() {
        return this.groupFrequencies.length;
    }
```

```
    public IndividualFrequencies getIndividualFrequency(Individual
         individualType) {
        for (IndividualFrequencies ind:groupFrequencies) {
            if(ind.getIndividualType().equals(individualType)) {
                return ind;
            }
        }
        return new IndividualFrequencies();
    }

    public IndividualFrequencies[] getFrequencies() {
        return groupFrequencies;
    }

    public GroupFrequencies mergeWithGroup(GroupFrequencies other)
         {
        IndividualFrequencies [] individualFrequencies = new
            IndividualFrequencies[this.groupFrequencies.length +
            other.getNumberOfGenotypes()];
        for (int i = 0; i < this.groupFrequencies.length; i++) {
            individualFrequencies[i] = this.groupFrequencies[i].
                clone();
        }
        IndividualFrequencies[] otherIndividualFrequencies = other
            .getFrequencies();
        for (int i = 0; i < otherIndividualFrequencies.length; i
            ++) {
            individualFrequencies[i + this.groupFrequencies.length]
                 = otherIndividualFrequencies[i].clone();
        }
        return new GroupFrequencies(individualFrequencies);
    }

    public void rescaleGroup(int populationSize) {
        int totalCurrentSize = this.getSize();
        for (IndividualFrequencies ind : this.groupFrequencies) {
            ind.setFrequency((long) ind.getFrequency() *
                populationSize / totalCurrentSize);
        }
    }
}
```

6.5 Main.java (extended version)

```
import java.util.ArrayList;
import java.util.Collections;
```

```
public class Main {

    private static final int POPULATION_SIZE = 4000;
    private static final int GENERATIONS = 1000;
    private static final int TIME_IN_GROUP = 4;
    private static final int TIME_IN_GROUP2 = 5;
    private static final int SMALL_GROUP_SIZE = 4;
    private static final int LARGE_GROUP_SIZE = 40;
    private static final double SMALL_GROUP_RESOURCE = 4;
    private static final double LARGE_GROUP_RESOURCE = 50;
    private static final double DEATH_RATE = 0.1;
    private static final Individual SELFISH_SMALL = new Individual
        (true, false);
    private static final Individual SELFISH_LARGE = new Individual
        (true, true);
    private static final Individual COOPERATIVE_SMALL = new
        Individual(false, false);
    private static final Individual COOPERATIVE_LARGE = new
        Individual(false, true);
    private static final Individual [] ALL_INDIVIDUALS = new
        Individual[] {SELFISH_LARGE, COOPERATIVE_LARGE,
        SELFISH_SMALL, COOPERATIVE_SMALL};

    private static ArrayList<Individual> generatePopulation() {
        ArrayList<Individual> population = new ArrayList<>();
        for (int i = 0; i < POPULATION_SIZE / ALL_INDIVIDUALS.
            length; i++) {
            for (Individual individual : ALL_INDIVIDUALS) {
                population.add(individual.clone());
            }
        }
        return population;
    }

    private static ArrayList<ArrayList> computeGroups(ArrayList<
        Individual> population, int groupSize, boolean largeGroup)
         {
        ArrayList<ArrayList> allGroups = new ArrayList<>();
        ArrayList<Individual> group = new ArrayList<>();
        for (Individual individual : population) {
            if(individual.isLargeGroupGene() == largeGroup) {
                group.add(individual.clone());
            }
            if(group.size() == groupSize) {
```

```
            allGroups.add(group);
            group = new ArrayList<>();
        }
    }
    return allGroups;
}

private static double computeNextGeneration(double
    thisGeneSize, double thisGeneGrowth, double
    thisGeneConsumption,
                                    double thatGeneSize,
                                        double thatGeneGrowth
                                        , double
                                        thatGeneConsumption,
                                        double resources) {
    double resourceShare = ( (thisGeneSize * thisGeneGrowth *
        thisGeneConsumption) /
            (thatGeneSize * thatGeneGrowth *
                thatGeneConsumption + thisGeneSize *
                thisGeneGrowth * thisGeneConsumption) ) *
                resources;
    return thisGeneSize + (resourceShare / thisGeneConsumption
        ) - DEATH_RATE * thisGeneSize;
}

private static ArrayList<GroupFrequencies>
    translateGroupsToFrequencies(ArrayList<ArrayList> groups,
    Individual cheater, Individual cooperator) {
    ArrayList<GroupFrequencies> result = new ArrayList<>();
    for (ArrayList<Individual> group : groups) {
        result.add(new GroupFrequencies(new
            IndividualFrequencies(Collections.frequency(group,
            cheater), cheater), new IndividualFrequencies(
            Collections.frequency(group, cooperator),
            cooperator)));
    }
    return result;
}

private static ArrayList<Individual>
    generatePopulationGivenFrequencies(IndividualFrequencies
    ... individualFrequencies) {
    ArrayList<Individual> result = new ArrayList<>();
    for (IndividualFrequencies ind : individualFrequencies) {
        for (int i = 0; i < ind.getFrequency(); i++) {
```

```
            result.add(ind.getIndividualType().clone());
        }
    }
    return result;
}

private static GroupFrequencies reproductionStep(ArrayList<
    ArrayList> groups, boolean largeGroup, int timeInGroups) {
    Individual cheater = largeGroup ? SELFISH_LARGE :
        SELFISH_SMALL;
    Individual cooperator = largeGroup ? COOPERATIVE_LARGE :
        COOPERATIVE_SMALL;
    ArrayList<GroupFrequencies> groupsWithFrequencies =
        translateGroupsToFrequencies(groups, cheater,
        cooperator);
    double groupResources = largeGroup ? LARGE_GROUP_RESOURCE
        : SMALL_GROUP_RESOURCE;
    double totalNumberOfCheaters = 0;
    double totalNumberOfCooperators = 0;
    for (int i = 0; i < timeInGroups; i++) {
        totalNumberOfCheaters = 0;
        totalNumberOfCooperators = 0;
        int index = 0;
        for (GroupFrequencies group : groupsWithFrequencies) {
            double numberOfCheaters = group.getSizeOfGenotype(
                cheater);
            double numberOfCooperators = group.
                getSizeOfGenotype(cooperator);
            double nextNumberOfCheaters = computeNextGeneration
                (numberOfCheaters, Individual.GROWTH_SELFISH,
                Individual.CONSUMPTION_SELFISH,
                    numberOfCooperators, Individual.
                        GROWTH_COOPERATIVE, Individual.
                        CONSUMPTION_COOPERATIVE, groupResources)
                        ;
            double nextNumberOfCooperators =
                computeNextGeneration(numberOfCooperators,
                Individual.GROWTH_COOPERATIVE, Individual.
                CONSUMPTION_COOPERATIVE,
                    numberOfCheaters, Individual.GROWTH_SELFISH,
                         Individual.CONSUMPTION_SELFISH,
                        groupResources);
            groupsWithFrequencies.set(index, new
                GroupFrequencies(new IndividualFrequencies(
                nextNumberOfCheaters, cheater), new
```

```
                IndividualFrequencies(nextNumberOfCooperators,
                cooperator)));
            index++;
            totalNumberOfCheaters += nextNumberOfCheaters;
            totalNumberOfCooperators += nextNumberOfCooperators
                ;
        }
    }
    return new GroupFrequencies(new IndividualFrequencies(
        totalNumberOfCheaters, cheater), new
        IndividualFrequencies(totalNumberOfCooperators,
        cooperator));
}

private static void printGenerations(ArrayList<
    GroupFrequencies> generations) {
    for (GroupFrequencies group : generations) {
        for (Individual ind : ALL_INDIVIDUALS) {
            System.out.printf("%.0f␣", group.
                getIndividualFrequency(ind).getFrequency());
        }
    }
}

public static void main(String[] args) {
    // 1: Generate population of N individuals. Equiprobable
        distribution of genotypes.
    ArrayList<Individual> population = generatePopulation();
    ArrayList<Individual> populationT5 = generatePopulation();
    ArrayList<GroupFrequencies> allGenerationsFrequencies =
        new ArrayList<>();
    ArrayList<GroupFrequencies> allGenerationsFrequenciesT5 =
        new ArrayList<>();
    for (int i = 0; i < GENERATIONS / TIME_IN_GROUP; i++) {
        Collections.shuffle(population);
        Collections.shuffle(populationT5);
        // 2: Divide into groups
        ArrayList<ArrayList> smallGroups = computeGroups(
            population, SMALL_GROUP_SIZE, false);
        ArrayList<ArrayList> largeGroups = computeGroups(
            population, LARGE_GROUP_SIZE, true);
        ArrayList<ArrayList> smallGroupsT5 = computeGroups(
            populationT5, SMALL_GROUP_SIZE, false);
        ArrayList<ArrayList> largeGroupsT5 = computeGroups(
            populationT5, LARGE_GROUP_SIZE, true);
```

```
        // 3: Reproduction step
        GroupFrequencies largeGroupsFrequencies =
            reproductionStep(largeGroups, true, TIME_IN_GROUP);
        GroupFrequencies smallGroupFrequencies =
            reproductionStep(smallGroups, false, TIME_IN_GROUP)
            ;
        GroupFrequencies largeGroupsFrequenciesT5 =
            reproductionStep(largeGroupsT5, true,
            TIME_IN_GROUP2);
        GroupFrequencies smallGroupFrequenciesT5 =
            reproductionStep(smallGroupsT5, false,
            TIME_IN_GROUP2);
        // 4: Return progeny to the migrant pool
        GroupFrequencies currentGeneration =
            smallGroupFrequencies.mergeWithGroup(
            largeGroupsFrequencies);
        GroupFrequencies currentGenerationT5 =
            smallGroupFrequenciesT5.mergeWithGroup(
            largeGroupsFrequenciesT5);
        int totalSize = currentGeneration.getSize() +
            currentGenerationT5.getSize();
        if(currentGeneration.getSize() > 0) {
            int populationMagnitudeT4 = currentGeneration.
                getSize() * POPULATION_SIZE / totalSize;
            currentGeneration.rescaleGroup(
                populationMagnitudeT4);
        }
        if(currentGenerationT5.getSize() > 0) {
            int populationMagnitudeT5 = currentGenerationT5.
                getSize() * POPULATION_SIZE / totalSize;
            currentGenerationT5.rescaleGroup(
                populationMagnitudeT5);
        }
        allGenerationsFrequencies.add(currentGeneration);
        allGenerationsFrequenciesT5.add(currentGenerationT5);
        // 6: Repeat from step 2 for N generations.
        population = generatePopulationGivenFrequencies(
            currentGeneration.getFrequencies());
        populationT5 = generatePopulationGivenFrequencies(
            currentGenerationT5.getFrequencies());
    }
    printGenerations(allGenerationsFrequencies);
    printGenerations(allGenerationsFrequenciesT5);
    System.out.printf("%d␣%d", TIME_IN_GROUP, TIME_IN_GROUP2);
}
```

```
}
```

6.6 plotBacteria.py

```
#!/home/dimitri/anaconda3/bin/python
import numpy as np
import matplotlib.pyplot as plt
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='the␣frequencies␣of␣the␣bacteria')
args = parser.parse_args()
bacteria_frequencies = np.array(args.integers)
columns = 4
print(bacteria_frequencies.shape)
bacteria_frequencies = bacteria_frequencies.reshape(int(
    bacteria_frequencies.shape[0] / columns), columns)
print(bacteria_frequencies)
print(bacteria_frequencies[:, 0])
sl, cl, ss, cs = bacteria_frequencies[:, 0], bacteria_frequencies
    [:, 1], bacteria_frequencies[:, 2], bacteria_frequencies[:, 3]
plt.plot(sl, '-o', label='Selfish␣Large')
plt.plot(cl, '-o', label='Coop.␣Large')
plt.plot(ss, '-o', label='Selfish␣Small')
plt.plot(cs, '-o', label='Coop.␣Small')
plt.suptitle("Genotype␣frequencies␣over␣generations", fontsize=16)
plt.ylabel('Group␣Size')
plt.xlabel('Generations')
plt.legend(loc='upper␣right')
plt.show()
```

6.7 plotBacteria.py (extended version)

```
#!/home/dimitri/anaconda3/bin/python
import numpy as np
import matplotlib.pyplot as plt
import argparse


parser = argparse.ArgumentParser()
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='the␣frequencies␣of␣the␣bacteria')
args = parser.parse_args()
bacteriaT2 = args.integers.pop()
bacteriaT1 = args.integers.pop()
print(bacteriaT1)
```

```
print(bacteriaT2)
bacteria_frequencies = np.array(args.integers[:len(args.integers)
    // 2])
bacteria_frequencies2 = np.array(args.integers[len(args.integers)
    // 2:])
#bacteria_frequencies = np.array(args.integers)
columns = 4
# print(bacteria_frequencies.shape)
bacteria_frequencies = bacteria_frequencies.reshape(int(
    bacteria_frequencies.shape[0] / columns), columns)
bacteria_frequencies2 = bacteria_frequencies2.reshape(int(
    bacteria_frequencies2.shape[0] / columns), columns)
# print(bacteria_frequencies)
print(bacteria_frequencies)
print(bacteria_frequencies2)
# print(bacteria_frequencies[:, 0])
sl, cl, ss, cs = bacteria_frequencies[:, 0], bacteria_frequencies
    [:, 1], bacteria_frequencies[:, 2], bacteria_frequencies[:, 3]
plt.plot(sl, '-', label="Selfish␣Large␣T{}".format(bacteriaT1))
plt.plot(cl, '-', label='Coop.␣Large␣T{}'.format(bacteriaT1))
plt.plot(ss, '-', label='Selfish␣Small␣T{}'.format(bacteriaT1))
plt.plot(cs, '-', label='Coop.␣Small␣T{}'.format(bacteriaT1))
sl2, cl2, ss2, cs2 = bacteria_frequencies2[:, 0],
    bacteria_frequencies2[:, 1], bacteria_frequencies2[:, 2],
    bacteria_frequencies2[:, 3]
plt.plot(sl2, '-', label='Selfish␣Large␣T{}'.format(bacteriaT2))
plt.plot(cl2, '-', label='Coop.␣Large␣T{}'.format(bacteriaT2))
plt.plot(ss2, '-', label='Selfish␣Small␣T{}'.format(bacteriaT2))
plt.plot(cs2, '-', label='Coop.␣Small␣T{}'.format(bacteriaT2))
plt.suptitle("Genotype␣frequencies␣over␣generations", fontsize=16)
plt.ylabel('Group␣Size')
plt.xlabel('Generations')
plt.legend(loc='upper␣right')
plt.show()
```