Individual Selection with Mutation for Cooperative Group Formation

Thomas Smith
taes1g09@ecs.soton.ac.uk

January 9, 2013

1 Introduction

Powers, Penn, and Watson [2007] show that environmental conditions need not be externally imposed in order to promote the evolution of cooperative traits. They present a model which permits suitable conditions to arise via individual selection, and show that even in environments that initially select for selfish behaviour, a niche construction process can allow for cooperative behaviours to be ultimately successful. This paper reimplements the algorithm described, and extends the model to show not only that the process can be made more robust by the introduction of mutation to the model, but also that a side-effect of the niche construction process then favours selecting against individuals that are able to mutate, resulting in a more stable niche.

2 Reimplementation

In Powers et al. [2007], an algorithm is presented which demonstrates that under the right circumstances, individuals can select for environments that promote cooperation. Using the trait-group aggregation and dispersal model presented in Wilson [1975], individuals are permitted to select for control both over individual strategy and initial group size. Large groups receive a resource bonus [Allee, 1938], but are also more susceptible to exploitation by selfish individuals. Groups are formed according to individual size preferences, and with a random but globally proportionate composition of strategies. Individuals in groups are then reproduced over a number of generations, with shares of the group's resources (r_i) allocated according to the proportion of individuals with a particular genotype within that group. When using the parameters in Appendix A, this should result in individuals selecting away from the resource bonus of the larger groups in favour of an environment where the cooperative trait has greatest individual fitness. In small groups, stochastic between-group variation ultimately globally favours the growth of groups with a higher proportion of cooperative individuals.

After initialising the migrant pool with N individuals, Powers et al. [2007] present the following algorithm as an implementation of the trait-aggregation model. The steps represent a single cycle, and should be repeated for T generations.

1. **Aggregation:** Assign individuals in the migrant pool randomly to groups according to their size preference – each group will have a random composition of strategies, but the average should be proportionate to the global ratios. Surplus individuals may be discarded.
2. **Reproduction:** Perform reproduction within groups. Eq. 1 gives the resource share that each genotype population receives, and Eq. 2 gives the resulting change in population size. Repeat for t generations.
3. **Dispersal:** Return the progeny of each group to the migrant pool.
4. **Scaling:** Rescale the migrant pool back to size N, retaining the proportion of individuals with each genotype.

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

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

2.1 Representation

In the reimplementation, individuals are represented as aggregate populations by genotype. Each individual can specify two genes: one for size, and one for strategy, and each gene can take one of two values, large/small and cooperative/selfish, respectively. This results in four possible genotypes, and so a migrant pool is maintained of the total count of each of the four types. Each group formed also maintains only the total count of each genotype within it, and aggregation, reproduction, dispersal and scaling operations are all carried out by population rather than individually.

The algorithm has no explicit fitness function - rather, an individual's fitness may be approximated by the total resource share of its genotype within the population as a whole. There is no crossover, but the algorithm could be said to be generational, as after each aggregation/dispersal round the whole population is rescaled back to a nominal 'carrying capacity' (N). The rescaling can force populations below their minimum group sizes and hence to extinction when they are significantly outcompeted.

2.2 Results

When the reimplemented algorithm is run using the parameters in Appendix A, the output (Fig. 2) closely follows the original findings (Fig. 1).

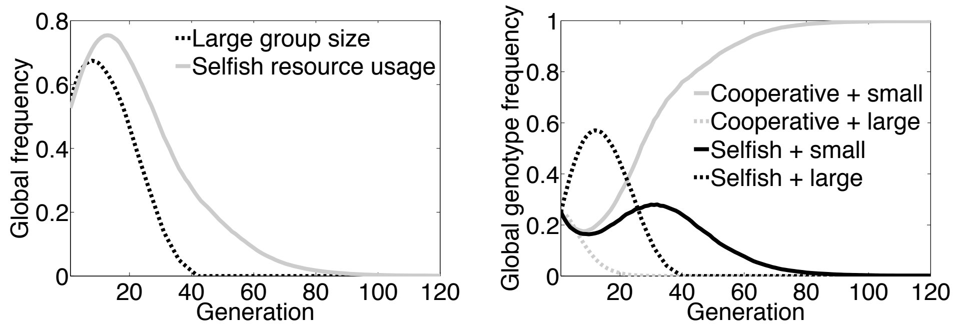

Figure 1: Left: average environment and strategy through time.
Right: change in genotype frequencies over time. [Powers et al., 2007]

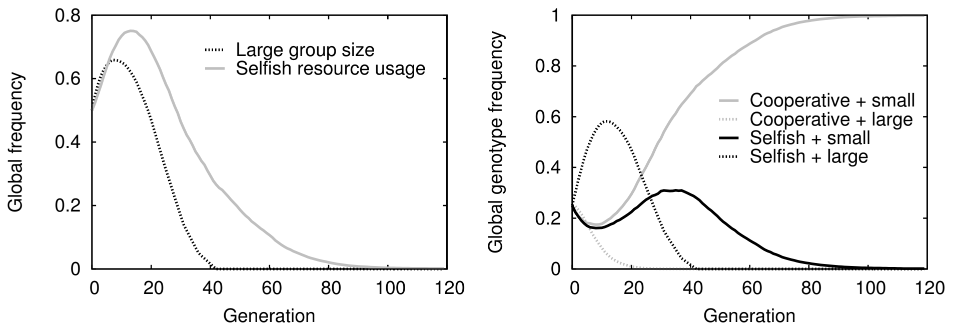

Figure 2: Results provided by reimplemented algorithm, as above. The plots match to a high degree of accuracy, showing accurate reimplementation.

As in Powers et al. [2007], large individuals initially benefit from the resource bonus, with large + selfish individuals flourishing as expected for the first few generations. Without a healthy population of large + cooperative individuals to exploit, the selfish individuals begin to decrease in frequency. Among the small groups, between-group variance exerts a selective pressure against groups containing selfish individuals [Powers and Watson, 2011], eventually driving the selfish allele extinct and allowing the cooperative + small genotype to fix in the population at equilibrium.

3 Extension

The original result can be shown to be quite robust: even in the case of severe imbalance in the initial genotype distribution, it is generally seen that the cooperative + small genotype successfully forms a niche and eventually fixes (Appendix B, figure 6). However, this is only possible when there exists a critical minimum population of cooperative + small individuals that between-groups variation can select in favour of. In the model as presented, if a particular genotype dies out or does not exist in the initial distribution, then it is never possible for it to (re)enter the gene pool. In certain situations, this can be seen as disadvantageous - for example, if in the original model the cooperative + small individuals were able to form large groups again once the selfish allele had been eradicated, then they could reap the resource bonus without exploitation by selfish individuals.

As an extension to the original model, this paper then presents analysis of cases where individuals may occasionally spontaneously change their size preference or strategy. An additional gene is introduced to each individual which both controls and is subject to these spontaneous changes, in a manner similar to a low rate of mutation across the whole population [Smith and Fogarty, 1996]. The expectation is that the addition of mutable individuals will make the niche construction process possible in a wider range of more initially hostile populations.

3.1 Representation

Once again, individuals are represented by genotype as populations of identical clones. As there are three distinct genes, there are now eight separate combinations of alleles representing populations in the migrant pool and in each group. The reproduction code has been modified to respect the possibility of newly mutated individuals being present in a group size different to that specified by their genotype. Mutation potentially occurs after each within-group reproduction cycle.

Ochoa et al. [2000] notes that 'optimal per-locus mutation rates depend mainly on $1/L$ (the reciprocal of the genotype length)'. In this model, the length of each individual's genotype is 3, and a mutation rate of $1/3$ is high enough to cause no significant solution to arise. However, the original reasoning behind the heuristic leads to a more effective value for mutation within the model. The use of a mutation rate of $1/L$ is intended to result in an average of one change to one gene in an individual per reproduction - in the model presented, the desired mutation rate is on the order of one change of one group per cycle of reproduction, and so a value of $1/num\_groups$ is more appropriate. Though the actual number of groups fluctuates based on prevailing group size preference, 550 groups may be taken as a reasonable approximation[1]. A mutation rate M of $\frac{1}{550} \approx 0.002$ is therefore used throughout the rest of the paper.

3.2 Results

In order to investigate the specific scenario mentioned above, individual mutation was initially restricted to act upon the size gene only.

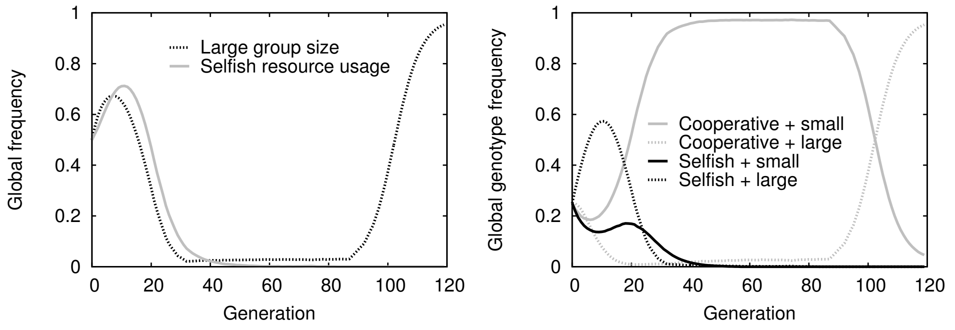

Figure 3: Mutation on the size gene allows the cooperative + large genotype to reappear and then eventually flourish once the selfish allele has died out.

The initial plot was then recreated using an equal distribution of all eight genotypes, and therefore an equal proportion of mutable to non-mutable individuals - shown on these plots as a separate aggregate proportion.

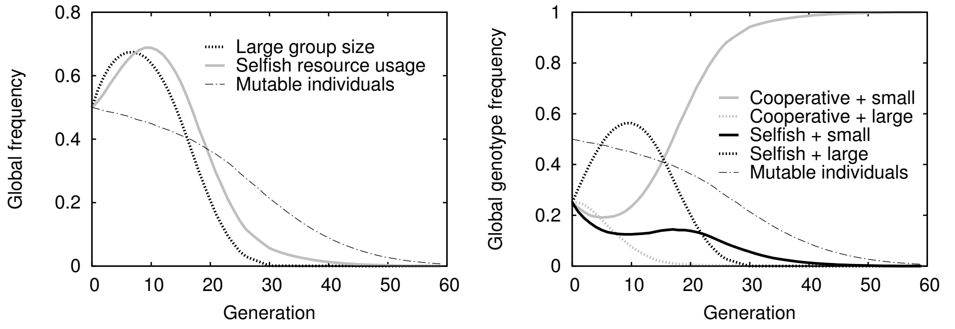

Figure 4: In this case, the addition of mutable individuals leads to the same outcome more quickly, and the mutable individuals die out.

[1] using the parameters in table 1 with equal distribution of genotypes: $\frac{2000}{40} + \frac{2000}{4} = 550$

Finally, with a high proportion of mutable individuals it is possible to start with an initial population distribution consisting of only a single genotype, and still demonstrate the niche creation process leading to the small + cooperative genotype dominating the population.

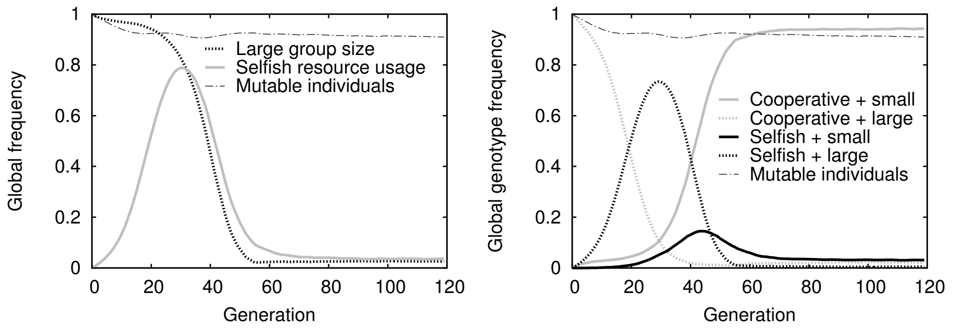

Figure 5: The population still contains a high proportion of mutable individuals after 120 generations, and so unlike in Figure 4 it is still possible that another genotype could later come to dominate, as in Figure 3.

4 Conclusion

The addition of mutable individuals to the algorithm as presented shows that the effects encouraging the niche construction process are robust, and largely independant of the initial genotype distribution. It demonstrates that the model is not only capable of selecting for small + cooperative groups despite unfavourable conditions - it is also capable of doing so when there are no small + cooperative individuals present in the original population.

One outcome is that the addition of mutation to the algorithm has increased the stochasticity of the output – previously, the process was largely deterministic save for the random group composition necessary for the between-group selection. Specifically, since in this model the gene for mutability has no direct effect on a populations' fitness[2], the proportion of mutable individuals after a few generations can vary greatly from run to run. A general trend towards non-mutable individuals is often observable – this is likely to be due to the one-way nature of the mutation from mutable to non-mutable individuals, and is most apparent when a genotype is close to fixing (as in Appendix B, figure 7).

[2] after a number of generations, a population of a mutable genotype is likely to be slightly smaller than an otherwise identical non-mutable genotype population, due to individuals mutating themselves out of the genotype

One area for further investigation would be to allow mutation on size preference to act in a more granular manner, as suggested in Powers [2010]. Under the current model, each gene has binary alleles. If mutation were able to affect the size preference in a continuous manner, it could demonstrate the development of a selfish population via mutation to create the correct conditions for cooperation in a more plausible, gradual manner. Other properties of the model, such as the time spent reproducing in groups before migration, could also become individually selectable and mutable.

References

W.C. Allee. *The social life of animals.* Norton New York, 1938.

G. Ochoa, I. Harvey, and H. Buxton. Optimal mutation rates and selection pressure in genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 315–322, 2000.

S. Powers, A. Penn, and R. Watson. Individual selection for cooperative group formation. *Advances in Artificial Life*, pages 585–594, 2007.

Simon T. Powers and Richard A. Watson. Evolution of individual group size preference can increase group-level selection and cooperation. In *Advances in Artificial Life. Darwin Meets von Neumann*, volume 5778 of *Lecture Notes in Computer Science*, pages 53–60. 2011.

S.T. Powers. *Social niche construction: evolutionary explanations for cooperative group formation.* PhD thesis, University of Southampton, 2010.

J. Smith and T.C. Fogarty. Self adaptation of mutation rates in a steady state genetic algorithm. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 318–323. IEEE, 1996.

D.S. Wilson. A theory of group selection. *Proceedings of the National Academy of Sciences*, 72(1):143–146, 1975.

Appendix A Algorithm parameters

Behaviour parameters	Cooperative	Selfish
Growth rate, G_i	0.018	0.02
Consumption rate, C_i	0.1	0.2

Size parameters	Large	Small
Group size, S_i	40	4
Resource influx, R_i	50	4

Global parameters	Value
Population size, N	4000
Generations, T	120
Reproductions, t	4
Death rate, K	0.1

Table 1: Parameters from Powers et al. [2007], used throughout.

Appendix B Additional graphs

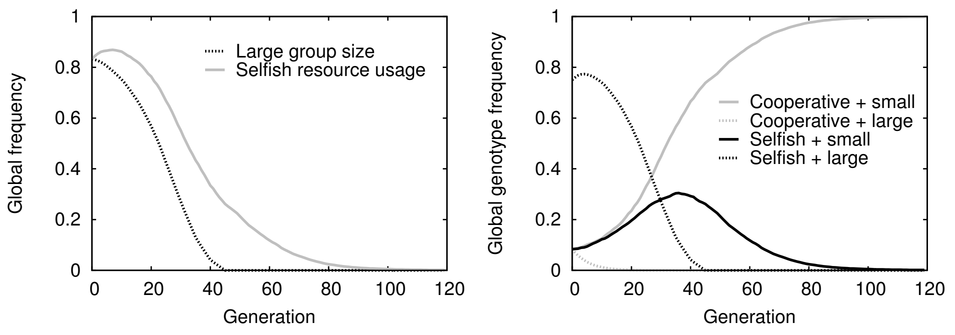

Figure 6: The model presented in Powers et al. [2007] is robust even in the face of severe imbalance in the initial genotype distribution.

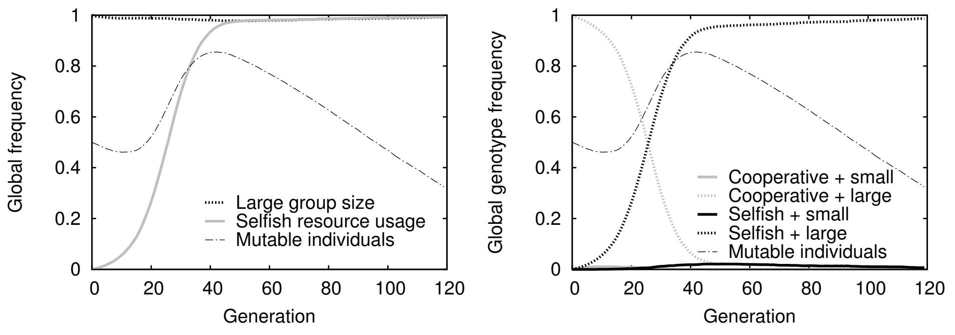

Figure 7: Compare to Figure 5. The initial distributions differ only in the proportion of mutable individuals, yet provide startlingly different outcomes.

Appendix C Source code

Two files are presented: `base.html` is the original reimplementation of the algorithm in Powers et al. [2007], and `further.html` is the implementation of the extension detailed in Section 3.

base.html

```
<html>
  <head>
    <title>COMP6026 Assignment 2: Individual selection for co-operative group formation</title>
    <script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>
  </head>
  <body>
    <div class="data">
    </div>
    <script>
    $(document).ready(function () {

      var LARGE = 0;
      var SMALL = 1;
      var COOP = 0;
      var SELF = 1;

      //from assignment notes page [large, small]
      var GROUP_SIZE = [40, 4];
      var DEATH_RATE = 0.1;
      var INFLUX_RATE = [50, 4];

      //from section 3 of paper [coop, self]
      var GROWTH_R = [0.018, 0.02];
      var CONSUME_R = [0.1, 0.2];
      var N = 4000;
      var T = 120;
      var t = 4;

      var kinds = 4;
      var LARGE_COOP = 0;
      var LARGE_SELF = 1;
      var SMALL_COOP = 2;
      var SMALL_SELF = 3;

      function Pool() {
        //initialise with N individuals
        var migrants = [N/kinds, N/kinds, N/kinds, N/kinds];

        var large_groups = [];
        var small_groups = [];

        function Group (groupSize) {
          var counts = [0,0];
          var size = groupSize;

          (function init() {
            var offset = groupSize * 2;     //small individuals are in indexes 2 and 3
            var total, index, selfish;
            for (var i = GROUP_SIZE[size] - 1; i >= 0; i--) {       //until this group is full
              total = migrants[offset] + migrants[offset+1];  //total size of pool remaining
              index = Math.random() * total;           //random index into pool
              selfish = (index > migrants[offset]);      //selfish individual at this offset?
              migrants[offset+Number(selfish)]--;        //decrement the appropriate migrant pool
              counts[Number(selfish)]++           //increment the pool in this group
            };

          })();

          function verify() {
            return (counts[0] + counts[1]) == GROUP_SIZE[size];
          }

          this.reproduce = function() {
```

```
          var shares = [0,0];
          var desire = [0,0];
          desire[0] = counts[0] * GROWTH_R[0] * CONSUME_R[0];
          desire[1] = counts[1] * GROWTH_R[1] * CONSUME_R[1];
          shares[0] = INFLUX_RATE[size] * desire[0] / (desire[0] + desire[1]);
          shares[1] = INFLUX_RATE[size] * desire[1] / (desire[0] + desire[1]);
          counts[0] = counts[0] + shares[0]/CONSUME_R[0] - DEATH_RATE*counts[0];
          counts[1] = counts[1] + shares[1]/CONSUME_R[1] - DEATH_RATE*counts[1];
        }

        this.disperse = function() {
          var offset = groupSize * 2;     //small individuals are in indexes 2 and 3
          migrants[offset] += counts[0];
          migrants[offset+1] += counts[1];
        }
      }

      //for T iterations
      while (T-- > 0) {

        //for individual.dat
        $(".data").append(migrants[0]/N + " " + migrants[1]/N + " " + migrants[2]/N + " " + migrants[3]/N + "<br />");

        //assign to groups (spares are unused)
        while (migrants[LARGE_COOP] + migrants[LARGE_SELF] >= GROUP_SIZE[LARGE]) {
          large_groups.push(new Group(LARGE));
        }
        while (migrants[SMALL_COOP] + migrants[SMALL_SELF] >= GROUP_SIZE[SMALL]) {
          small_groups.push(new Group(SMALL));
        }

        //reproduce each group for t steps
        for (var i = t - 1; i >= 0; i--) {
          $.each(large_groups, function (index, value){
            value.reproduce();
          });
          $.each(small_groups, function (index, value){
            value.reproduce();
          });
        };

        //return to pool
        migrants = [0, 0, 0, 0];
        $.each(large_groups, function (index, value){
          value.disperse();
        });
        $.each(small_groups, function (index, value){
          value.disperse();
        });
        large_groups.length = small_groups.length = 0;

        //rescale the pool
        var total = migrants.reduce(function(a,b) { return a+b; });
        $.each(migrants, function(index, value) {
          migrants[index] = value * N / total;
        });

      };
    }

    new Pool();
  });
  </script>
</body>
</html>
```

further.html

```
<html>
  <head>
    <title>COMP6026 Assignment 2: Individual selection for co-operative group formation</title>
    <script src="http://code.jquery.com/jquery-1.7.2.min.js"></script>
  </head>
  <body>
    <div class="data">
    </div>
    <script>
    $(document).ready(function () {

      var MUTIE = 0;
      var FIXED = 1;
      var LARGE = 0;
      var SMALL = 1;
      var COOP = 0;
      var SELF = 1;

      //from assignment notes page [large, small]
      var GROUP_SIZE = [40, 4];
      var DEATH_RATE = 0.1;
      var INFLUX_RATE = [50, 4];

      //from section 3 of paper [coop, self]
      var GROWTH_R = [0.018, 0.02];
      var CONSUME_R = [0.1, 0.2];
      var N = 4000;
      var T = 120;
      var t = 4;

      //extension
      var MUTATE_RATE = 0.002;

      function Pool() {
        function newDataStructure() {
          return [[[0, 0], [0, 0]],
              [[0, 0], [0, 0]]];
        }

        //initialise with N individuals
        var migrants = newDataStructure();
        migrants[MUTIE][LARGE][COOP] = N;

        //helper function - passes all appropriate indexes to given function
        multi_each = function (multi, func) {
          var total = 0;
          $.each(multi, function (adapt, val) {
            $.each(val, function (size, valu) {
              $.each(valu, function (plan, value) {
                total += func(adapt, size, plan);
              })
            })
          })
          return total; //inbuilt summation support
        }

        var large_groups = [];
        var small_groups = [];

        function Group (groupSize) {
          var counts = newDataStructure();
          var size = groupSize;

          (function init() {

            var total, index, selfish, mutie, offset;
            total = 0;
            total = multi_each(migrants, function(adapt, size_index, plan) {  //total of this size
              if (size_index == size) {
                return migrants[adapt][size_index][plan];
              } else {
                return 0; //only total correct size
              }
            });
            for (var i = GROUP_SIZE[size] - 1; i >= 0; i--) { //until this group is full
    index = Math.random() * total;           //random index into pool
    offset = 0;
    for (var j = 0; j < 4; j++) {          //search the four genotypes
      mutie = (j >= 2)?1:0;
      selfish = j % 2;
      if (index < offset + migrants[mutie][size][selfish]) {  //if of this kind
        migrants[mutie][size][selfish]--;          //move into group
        counts[mutie][size][selfish]++;            //and out of pool
        break;                         //stop looking
      } else {
        offset += migrants[mutie][size][selfish];     //look further
      }
    };
    total--;       //we know it's gone down by one, doesn't matter where
  };
})();

function verify() {       //use multi_each on counts to sum all groups, check correct
  return (multi_each(counts, function(adapt_index, size_index, plan_index) {
    return counts[adapt_index][size_index][plan_index];
  })) == GROUP_SIZE[size];
}

this.mutate = function() {
  var offspring = newDataStructure();
  multi_each(counts, function(adapt_index, size_index, plan_index) {  //for each genotype
    if (adapt_index == MUTIE) {                    //if you can mutate, do
      for (var i = counts[adapt_index][size_index][plan_index] - 1; i >= 0; i--) {
        if (Math.random() < MUTATE_RATE) {
          offspring[adapt_index][size_index][(plan_index)?0:1]++; //strategy
        } else {
          offspring[adapt_index][size_index][plan_index]++;
        }
      };
    } else {    //if you can't mutate, reproduce identically
      offspring[adapt_index][size_index][plan_index] += counts[adapt_index][size_index][plan_index];
    };
  })
  counts = offspring;
  offspring = newDataStructure();
  multi_each(counts, function(adapt_index, size_index, plan_index) {  //for each genotype
    if (adapt_index == MUTIE) {                    //if you can mutate, do
      for (var i = counts[adapt_index][size_index][plan_index] - 1; i >= 0; i--) {
        if (Math.random() < MUTATE_RATE) {
          offspring[adapt_index][(size_index)?0:1][plan_index]++; //size preference
        } else {
          offspring[adapt_index][size_index][plan_index]++;
        }
      };
    } else {    //if you can't mutate, reproduce identically
      offspring[adapt_index][size_index][plan_index] += counts[adapt_index][size_index][plan_index];
    };
  })
  counts = offspring;
  offspring = newDataStructure();
  multi_each(counts, function(adapt_index, size_index, plan_index) {  //for each genotype
    if (adapt_index == MUTIE) {                    //if you can mutate, do
      for (var i = counts[adapt_index][size_index][plan_index] - 1; i >= 0; i--) {
        if (Math.random() < MUTATE_RATE) {
          offspring[(adapt_index)?0:1][size_index][plan_index]++; //mutability
        } else {
          offspring[adapt_index][size_index][plan_index]++;
        }
      };
    } else {    //if you can't mutate, reproduce identically
      offspring[adapt_index][size_index][plan_index] += counts[adapt_index][size_index][plan_index];
    };
  })
  counts = offspring;
}

this.reproduce = function() {
  var shares = newDataStructure();
  var desire = newDataStructure();
  var totalDesire = 0;
  multi_each(counts, function(a_ix, s_ix, p_ix) {  //calulate Di = Ni*Gs*Cs or Ni*Gc*Cc
              desire[a_ix][s_ix][p_ix] = counts[a_ix][s_ix][p_ix] * GROWTH_R[p_ix] * CONSUME_R[p_ix];
            });
            totalDesire = multi_each(desire, function(a_ix, s_ix, p_ix) {   //total desire, natch
              return desire[a_ix][s_ix][p_ix];
            });
            multi_each(counts, function(a_ix, s_ix, p_ix) { //influx share Si = R*Di/  D i  (use the actual group size for influx
              shares[a_ix][s_ix][p_ix] = INFLUX_RATE[size] * desire[a_ix][s_ix][p_ix] / totalDesire;
            });
            multi_each(counts, function(a_ix, s_ix, p_ix) { //Ni(t+1) = Ni(t) + Si/C(c/s) - K*Ni(t)
              counts[a_ix][s_ix][p_ix] = Math.floor(counts[a_ix][s_ix][p_ix] + shares[a_ix][s_ix][p_ix]/CONSUME_R[p_ix] - DEATH_
            })
            this.mutate();    //after reproduction, mutate
          }

          this.disperse = function() {
            multi_each(counts, function(adapt_index, size_index, plan_index) {
              migrants[adapt_index][size_index][plan_index] += counts[adapt_index][size_index][plan_index];
            })
          }
        }

        //for T iterations
        while (T-- > 0) {

          //for further.dat
          multi_each(migrants, function(adapt, size, plan) {
            $(".data").append(migrants[adapt][size][plan]/N + " ");
          })
          $(".data").append("<br />");

          //assign to groups (spares are unused)
          while (migrants[MUTIE][LARGE][COOP] + migrants[MUTIE][LARGE][SELF] +
               migrants[FIXED][LARGE][COOP] + migrants[FIXED][LARGE][SELF] >= GROUP_SIZE[LARGE]) {
            large_groups.push(new Group(LARGE));
          }
          while (migrants[MUTIE][SMALL][COOP] + migrants[MUTIE][SMALL][SELF] +
               migrants[FIXED][SMALL][COOP] + migrants[FIXED][SMALL][SELF] >= GROUP_SIZE[SMALL]) {
            small_groups.push(new Group(SMALL));
          }

          // reproduce each group for t steps
          for (var i = t - 1; i >= 0; i--) {
            $.each(large_groups, function (index, value){
              value.reproduce();
            });
            $.each(small_groups, function (index, value){
              value.reproduce();
            });
          };

          //return to pool
          migrants = newDataStructure();
          $.each(large_groups, function (index, value){
            value.disperse();
          });
          $.each(small_groups, function (index, value){
            value.disperse();
          });
          large_groups.length = small_groups.length = 0;

          //rescale the pool
          var total = multi_each(migrants, function(adapt, size, plan) {
            return migrants[adapt][size][plan];
          })
          var scale = N/total;
          multi_each(migrants, function(adapt, size, plan) {
            migrants[adapt][size][plan] = Math.floor(migrants[adapt][size][plan] * scale);
          })
        };
      }

      new Pool();
    });
    </script>
  </body>
</html>
```