Genetic Algorithms: Using Evolution to Search for Strategies for the Prisoners' Dilemma

Paul Scurek

May 16, 2014

MATH 599

1 Introduction

A **genetic algorithm (GA)** is an algorithm that uses ideas from evolution to search for solutions to some specified problem. The purpose of this paper is to introduce the basic principles employed in genetic algorithms, and then to demonstrate how they work with a practical example. First, we will develop some important terminology and describe the essential steps involved in a simple genetic algorithm. After the basics are covered, we will implement a genetic algorithm that searches for strategies for the prisoners' dilemma, a game which is studied in game theory.

2 Genetic Algorithms

Genetic algorithms are meant to mimic evolution. Thus, they will all start with some specified population of individuals, called **candidate solutions**, and they will all have a way of determining which individuals are the fittest in the population. In all cases, the **fitness** of an individual will be a measure of how well that individual solves the problem at hand. Additionally, each genetic algorithm will have a way of choosing which individuals from the current population will reproduce to form **offspring**, and it will also specify exactly how the traits of the chosen individuals, called **parents**, will be passed on to their offspring. This is roughly how a genetic algorithm works. What follows is a more detailed development of these ideas.

2.1 Biological Terminology

Since evolution is the model for genetic algorithms, it is not surprising that many terms and concepts have been borrowed from biology. The following is a list of terms, along with their meanings in the context of genetic algorithms.

Chromosome: A chromosome is a candidate solution (i.e. an individual in the population). When a genetic algorithm is implemented on a computer, a chromosome is commonly *represented* as a string of bits (0's and 1's).

Loci: Given the bit string representation of a chromosome, the loci of a particular bit is that bit's position in the string. It can be though of as the *index* of the bit in question.

Gene: A gene is a single bit or a group of bits at a particular loci of a chromosome, which encodes a particular feature of the candidate solution.

Allele: Given a bit string, an allele is either 0 or 1. Thus, there is an allele at each loci of a chromosome.

Crossover: Crossover is the process by which genetic material from two parent chromosomes is exchanged to form two new offspring chromosomes. Another name for crossover is **recombination**. In a GA, crossover takes

place between two chromosomes, x and y, when a randomly selected loci is chosen, and then the bits before that loci are switched between x and y. This produces two new chromosomes. For example, if $x = 1101$ and $y = 1010$, and the loci at which the crossover takes place is the second position from the right, then the two offspring would be 1001 and 1110.

Mutation: Mutation occurs when randomly chosen bits in a chromosome are switched from 0 to 1 or from 1 to 0. In a GA, mutation generally takes place in offspring before they are added to the population.

Genotype: The genotype of an individual chromosome is the arrangement of its bits. Thus, two chromosomes have the same genotype if they have the same allele at each loci.

Note 1. *In the definition of* **chromosome***, above, I mentioned that they are commonly represented as strings of* 0*'s and* 1*'s. In this section, we will assume that this is how all chromosomes are represented. In practice, however, they can be represented by strings of whatever symbols the programmer chooses. The symbols need not be* 0*'s and* 1*'s.*

Before we are ready to give a more precise description of a genetic algorithm, we must first explain some of the mathematical ideas involved.

2.2 Mathematical Concepts

In the first paragraph of section 2, I said that the fitness of an individual is a measure of how well that individual solves the problem of interest. The word measure indicates that something mathematical is going on behind the scenes. In this section, we will discuss some of the more mathematical aspects of genetic algorithms.

Search Space: A search space is a collection of candidate solutions to a given problem, along with some notion of *distance* between the candidate solutions. A *point* in a search space is a chromosome. For example, suppose the chromosomes in a search space, $\mathcal{S}$, are represented as bit strings, and that the distance between chromosomes is given by the hamming distance, denoted $d_H(x, y)$. That is, the distance between two chromosomes is equal to the number of positions at which their bits differ. Then the search space is actually a *metric space* since $\forall x, y, z \in \mathcal{S}$, the following hold.

1. $d_H(x, y) \geq 0$, and $d_H(x, y) = 0 \Leftrightarrow x = y$
2. $d_H(x, y) = d_H(y, x)$
3. $d_H(x, z) \leq d_H(x, y) + d_H(y, z)$

Fitness Function: Let $\mathcal{S}$ be a search space. Then a fitness function is a function $f : S \to \mathbb{R}$. That is, f maps each chromosome to a real number, called its **fitness**. In practice, the fitness of a given chromosome should

represent how good it is at solving the problem. For example, if a genetic algorithm is being used to find roots of a polynomial $p(x)$, then f could be the map defined by $f(x) \mapsto -|p(x)|$. Then the fittest individuals would be the ones with the greatest images under f.

Fitness-Proportionate Selection: In the first paragraph of section 2, I mentioned that every GA has a method of *choosing* which individuals in the population will reproduce. This process of choosing parent chromosomes is called **selection**. When Fitness-proportionate selection is used, the number of times an individual from the population is expected to reproduce is equal to it's fitness divided by the average fitness of the entire population. More formally, let $x_1, x_2, \ldots, x_n$ be the candidate solutions in a population of size n, and suppose that fitness-proportionate selection is utilized. Then the expected number of times that a given individual, x_0, will reproduce is

$$\frac{f(x_0)}{\frac{1}{n}\sum_{i=1}^{n} f(x_i)} \ ,$$

where f is the fitness function for the search space.

Roulette-Wheel Sampling: This is a method for implementing fitness-proportionate selection in a GA. Conceptually, roulette-wheel sampling is like assigning to each individual in the population a slice of a roulette-wheel equal in area to its fitness. Then, the selection process is like spinning the wheel. If the search space has n individuals, then the probability that a spin will land on a given chromosome, x_0, is

$$P(\text{spin lands on } x_0) = \frac{f(x_o)}{\sum_{i=1}^{n} f(x_i)} \ ,$$

where the denominator represents the total area of the roulette-wheel and the numerator represents the area of the slice associated with x_0.

Fitness Landscape: This is simply a representation of all the candidate solutions to a problem (i.e. chromosomes) along with their fitness. Formally, the fitness landscape of a given search space (i.e. population), $\mathcal{S}$, is the set of ordered pairs $\mathcal{F} = \{(x, f(x)) \mid x \in \mathcal{S}\}$.

Crossover Rate: After two parent chromosomes have been selected for reproduction, the probability that crossover actually takes place is called the crossover rate, denoted p_c. The crossover rate is a constant determined by the programmer, and the same rate applies to every pair of parents in the GA.

Mutation Rate: After two parent chromosomes have reproduced, the probability that mutation will occur at each loci in each of the offspring is called the mutation rate, denoted p_m. The mutation rate is a constant value determined by the programmer.

2.3 Skeleton of a Genetic Algorithm

In sections 2.1 and 2.2, we laid out the concepts and terminology of genetic algorithms that will be used throughout the rest of the paper. In this section, we will bring everything together and explain how basic genetic algorithms work.

The purpose of a genetic algorithm is to search for solutions to some specified problem. In the terminology we have just learned, searching for solutions to a problem amounts to searching the fitness landscape for highly fit chromosomes, where a chromosome, x, is highly fit if $|f(x)|$ is large relative to the fitness of the other chromosomes in the space. However, to say that a genetic algorithm *searches* may be a bit misleading because it does not actively seek out the fittest chromosomes. Instead, the searching process is facilitated by repeated random selection and recombination, where the fitter chromosomes are more likely to be chosen as parents. Thus, it is an indirect type of search.

Every genetic algorithm has four basic elements. At this point, we have introduced all four, but we list them below in an order that is indicative of the logical flow of a GA.

1. Population of chromosomes
2. Selection according to fitness
3. Crossover to produce new offspring
4. Random mutation of new offspring

Genetic algorithms work by successively replacing the current population with a new population consisting of offspring, and the hope is that the process of random selection and recombination has produced fitter chromosomes. Keeping these ideas in mind, we will now present the skeleton of a basic GA.

Given a clearly defined problem and a symbol representation for candidate solutions,

1. Randomly generate a population of n l-bit chromosomes (i.e. candidate solutions), and specify the number of iterations for the algorithm, N.
2. For each chromosome x in the current population, calculate its fitness $f(x)$.
3. Repeat the following three steps until n offspring are created.
   (a) Select a pair chromosomes from the current population to be parents. [***note:*** The probability of selection should be an increasing function of fitness. Also, selection is done with replacement, so that a given chromosome could parent multiple offspring.]
   (b) At a randomly chosen loci, perform crossover on the pair to form two offspring. For each pair of parents, crossover should occur with probability (crossover rate) p_c. If crossover does not occur, then the two offspring are simply copies of the two parent chromosomes.

(c) At each locus, mutate the two offspring with probability (mutation rate) p_m, and then add them to the *new* population.

4. If n (the original population size) is odd, then delete one member of the new population at random. [***note:*** This is because, in step 3, the offspring are created in pairs, so the only way to end up with an odd number of them is to make one extra and then delete one.]

5. Replace the current population with the new population.

6. Repeat steps 2-5 $N - 1$ times.

We call each complete iteration through steps 2-5 a **generation**. A **run** is defined to be the entire set of N generations, where N is the number of iterations specified in step 1.

Now that we have general idea of how a genetic algorithm should work, we are almost ready to implement our own. First, however, we must introduce the problem that we wish to find solutions to.

3 The Prisoners' Dilemma

The prisoners' dilemma is a two-person game studied in game theory in which strategies for maximizing short term payoff differ from strategies for maximizing long term payoff. In this section, we will describe the prisoners' dilemma in detail. Then, in the next section, we will implement a genetic algorithm to search for strategies for the game.

The scenario of the prisoners' dilemma is formulated as follows. Two people, A and B, are arrested under the suspicion of committing a crime together. They are held in different jail cells and interrogated separately, so they do not have a chance to come up with a false alibi together. While being interrogated, person A is presented with the following information:

- If person A admits to committing the crime and agrees to testify as a whiteness against person B, then A will be let free and B will get 5 years in prison.

- If person A confesses and agrees to testify against B, but person B also confesses and agrees to testify against A, then both of them will be put in prison for 4 years for pleading guilty.

- Person B is being offered exactly the same deal in his interrogation.

- If neither person agrees to testify against the other, then they will both likely only get 2 years in prison.

In the event that one person decides to testify against the other, we will say that person **defects** against the other. On the other hand, if one person decides not to testify against the other, we will say he **cooperates** with the other person.

For now, we will consider a game to be an occurrence of each person deciding to either cooperate or defect.

Thus, if we take the position of one of the players, say A, the question arises as to whether we should cooperate or defect. In order to analyze the situation in terms of maximum payoff, we first need to decide how payoff should be determined. Given the information contained in the bullet points above, one way to think about the payoff of a given decision is to subtract the resulting amount of time in prison from the total possible time in prison. Thus, if we consider a game where A defects and B cooperates, the payoff for A, denoted payoff(A), is 5 - 0 = 5. Likewise, payoff(B) = 5 - 5 = 0.

Assuming this is how we determine payoff, we see that a game consists of each player receiving a score based on their decision to either cooperate of defect. Thus, we can now define a **game** more formally as the ordered pair (payoff(A), payoff(B)) that results from each player making a decision. With these ideas in mind, we can construct the payoff matrix for the prisoners' dilemma, where each entry in the matrix is of the form (payoff(A), payoff(B)).

Payoff Matrix for Prisoners' Dilemma

		Player B	
		Cooperate	Defect
Player A	Cooperate	(3, 3)	(0, 5)
	Defect	(5, 0)	(1, 1)

Taking on the point of view of player A, let us first consider what happens if we play a single game. Suppose that our opponent, B, decides to cooperate. Then, referring to the payoff matrix, we will receive a payoff of 3 if we cooperate and a payoff of 5 if we defect. Thus, it is in our best interest to defect. If we hypothesize that player B will defect, then we will receive a payoff of 0 for cooperating and a payoff of 1 for defecting, so we are better off defecting in this scenario as well. Hence, in any single game, player A will ensure the greatest possible payoff (relative to B's decision) by defecting. Another way to analyze A's situation is by calculating the expected payoff, denoted E[payoff(A)], for each of his possible decisions (assuming that there is a 50/50 chance of B either defecting or cooperating). If player A cooperates, then E[payoff(A)] = $.5 \cdot 3 + .5 \cdot 0 = 1.5$. On the other hand, if player A defects, then E[payoff(A)] = $.5 \cdot 5 + .5 \cdot 1 = 3$. Thus, no matter how we look at it, player A is better off defecting in any given game.

Now let us consider what might happen if A plays multiple games against B. From the considerations in the previous paragraph, it is reasonable that player A might adopt the strategy of defecting in every game. To see what happens in the long run, we need to assume that player B is somewhat rational. If A defects, then B receives a payoff of 0 for cooperating and 1 for defecting. If B notices A's strategy, then he will almost certainly begin to defect every game as well. Then, with both players consistently defecting, each is receiving a steady

payoff rate of 1 per game. But if A and B are both rational and concerned with securing the highest payoff, it seems that they should eventually realize that they are better off cooperating with one another in the long run, so that they each receive a payoff of 3 each game. But is this what would happen? Maybe A and B would not want to take the risk of the other player not reciprocating cooperation, in which case the sequence of games would stabilize at (1, 1).

As we can see, the prisoners' dilemma is much harder to analyze in the long run. Clearly if maximizing payoff is the highest concern of the players, then the best strategy for a single game (defecting) is not the best strategy in the long run. On the other hand, it is not clear that the best strategy is to cooperate every time either. Maybe it would be best to cooperate most of the time, but then to defect once in a while in order to pick up an extra 2 payoff points (in the hopes that your opponent does not retaliate with a defection of his own).

According to Mitchell ('96), Robert Axelrod from the University of Michigan was very interested in long-run strategies for the prisoners' dilemma. In the 1980's, he organized multiple tournaments in which researches in various academic disciplines submitted computer programs that implemented different strategies. Axelrod pitted the strategies against one another, and took note of the ones that scored the best. He then implemented a genetic algorithm which played multiple games against four of the best strategies from the tournament. The goal was for the GA to evolve new strategies that were better than the ones submitted for the tournament. In the end, Axelrod found that most of best strategies that evolved were variations of the "TIT FOR TAT" strategy, in which cooperation is reciprocated and defection is punished. That is, the "TIT FOR TAT" strategies essentially copy the previous decision of their opponent.

In Axelrod's experiment, the environment of the genetic algorithm was someone static in the sense that the candidate solutions only played against the same four strategies. In the next section, we implement a more dynamic GA for searching for strategies in which each candidate solution plays against all of the other candidate solutions.

4 Implementing a Genetic Algorithm in Python

We have already discussed the structure of a genetic algorithm, and the GA that we implement here will follow that structure for the most part. We know that we need to generate a population and implement some kind of fitness function, but we have not yet discussed how the candidate solutions will *play* against one another. That is, how will the the chromosomes know when to cooperate or defect? We need to give the candidate solutions a *memory*, so they can decide what move to make based on their opponents previous decisions. In our example, we will allow the chromosomes to remember the past three games. Specifically, when creating the Python function that simulates a single game, we will store the decisions of the past three games in a list, which we will constantly update as games are played. Since a game consists of each player either defecting or cooperating, there are $(2 \cdot 2)^3 = 4^3 = 64$ possible memories.

Each chromosome will have to know what move to make for each of those 64 memories. Furthermore, a chromosome needs to make a first move, before there is a memory to work with.

Hence, we will encode the chromosomes as a string of 67 C's and D's in Python. The first 64 loci will each correspond to a specific memory, and the D or C at each locus will determine whether the chromosome defects or cooperates based on the corresponding memory. The last three loci will consist of the first three decisions that the chromosome will make when matched up with a new opponent (before a memory exists).

Imagine a tree of all of the possible 6 bit memories. The first locus of a chromosome (which has index 0 since indexing begins with 0 in Python) will correspond to the first branch of the memory tree CCCCCCC. The second locus of a chromosome (which has index 1) would correspond to the second branch CCCCCD. The sixty third locus of a chromosome (with index 62) would correspond to the sixty third branch DDDDDC. In general, a quick way to determine which memory a certain locus of a chromosome corresponds to, calculate the binary representation of the index of the locus. Then the corresponding memory is the result of assigning C to every 0 and D to every 1 in the binary representation of the index. For example, bin(62) = 111110, so the sixty third locus of a chromosome corresponds to DDDDDC, as we said before. As one final example, the index of the twenty eighth locus is 27, which has the binary representation bin(27) = 011011. Hence, the twenty eighth locus of a chromosome corresponds to the memory CDDCDD.

Note 2. *In a six bit memory, say DDCDDC, the first two letters are the decisions made by player A and player B three games ago. The second two letters are the decisions from two games ago, and the last two letters are the decisions made in the last game.*

4.1 Code

With the above considerations in mind, let's look at the code for our GA. First we will need to import the `math` and `random` libraries, and we will need a key of possible memories for the each chromosome to refer to.

```
import math
import random

memory_key = ['CCCCCC', 'CCCCCD', 'CCCCDC', 'CCCCDD', 'CCCDCC', 'CCCDCD',
              'CCCDDC', 'CCCDDD', 'CCDCCC', 'CCDCCD', 'CCDCDC', 'CCDCDD',
              'CCDDCC', 'CCDDCD', 'CCDDDC', 'CCDDDD', 'CDCCCC', 'CDCCCD',
              'CDCCDC', 'CDCCDD', 'CDCDCC', 'CDCDCD', 'CDCDDC', 'CDCDDD',
              'CDDCCC', 'CDDCCD', 'CDDCDC', 'CDDCDD', 'CDDDCC', 'CDDDCD',
              'CDDDDC', 'CDDDDD', 'DCCCCC', 'DCCCCD', 'DCCCDC', 'DCCCDD',
              'DCCDCC', 'DCCDCD', 'DCCDDC', 'DCCDDD', 'DCDCCC', 'DCDCCD',
              'DCDCDC', 'DCDCDD', 'DCDDCC', 'DCDDCD', 'DCDDDC', 'DCDDDD',
```

```
            'DDCCCC', 'DDCCCD', 'DDCCDC', 'DDCCDD', 'DDCDCC', 'DDCDCD',
            'DDCDDC', 'DDCDDD', 'DDDCCC', 'DDDCCD', 'DDDCDC', 'DDDCDD',
            'DDDDCC', 'DDDDCD', 'DDDDDC', 'DDDDDD']
```

Essentially, before each game is played, each chromosome will find the index of the current memory in `memory_key`, and then it will make a decision based on the letter (either C or D) at the locus in its string representation with the same index as that of the memory. Additionally, we will need a key that represents the payoff matrix, so we are able to assign the right scores after each game.

```
points_key = {'CC':(3, 3), 'CD':(0, 5), 'DC':(5, 0), 'DD':(1, 1)}
```

We want each chromosome of a given population to play multiple games against each of the other chromosomes. Thus, we need a Python function that simulates something similar to a round-robin tournament. The following code creates such a function.

```
def play_tournament(population, pop_size):

    points = [0]*pop_size
    memory = []

    for s in range(pop_size):

        # arrange the initial memories of t and s
        for t in range(s, pop_size):
            memory_s = population[s][64] + population[t][64] + population[s][65] +
                       population[t][65] + population[s][66] + population[t][66]
            memory_t = population[t][64] + population[s][64] + population[t][65] +
                       population[s][65] + population[t][66] + population[s][66]

            # play the game 100 times
            for i in range(100):
                memory_index_s = memory_key.index(memory_s)
                memory_index_t = memory_key.index(memory_t)
                move_s = population[s][memory_index_s]
                move_t = population[t][memory_index_t]
                outcome = points_key[move_s + move_t]
                points[s] += outcome[0]
                points[t] += outcome[1]
                memory_s = memory_s[2:] + move_s + move_t
                memory_t = memory_t[2:] + move_t + move_s
    return points
```

In the above code, `memory_s` and `memory_t` are the memories that `s` and `t` use to make their decisions, respectively. The reason that the memory strings are not the same for both chromosomes is that when a chromosome looks up

a memory in the `memory_index`, it understands the entries to be of the form (MyScore)(OpponentsScore)(MyScore)(OpponentsScore)(MyScore)(OpponentsScore). Thus, we need to arrange the initial memories of `s` and `t`, in such a way as to account for this fact. The rest of the code is simply successive iterations of the players looking up their memory's index in the `memory_key` and retrieving their points from `points_key`. In our experiment, we will have each chromosome play one hundred games against each chromosome, including itself.

Now that we have described how the actual prisoners' dilemma game is going to be played between chromosomes in a population, we can discuss the rest of the code for our GA. As noted in section 2.3, every genetic algorithm begins with a randomly generated initial population. We can accomplish this in Python wight the following function.

```
def generate_population(n, l):

    population = []
    chromosome = ''
    genes = ['C', 'D']

    # create a new chromosome x
    for x in range(n):
        # place C or D at index l of chromosome x
        for y in range(l):
            chromosome += random.choice(genes)
        population.append(chromosome)
        chromosome = ''

    return population
```

In the function heading `generate_population(n, l)` the the parameter `n` is the desired size of the population, and the parameter `l` is the desired length of each chromosome in the population. A key part of this function is the `random.choice(genes)` command, which randomly chooses between C or D as the letter to place in each locus of each chromosome.

Once our randomly generated population has played a round of the prisoners' dilemma tournament, we are able to determine the fitness of each chromosome as a function of the total points earned in the tournament (where total points earned is the sum of the payoffs earned in each game).

```
def get_fitness(points, pop_size):

    fitness = [0]*pop_size
    average_fitness = 0

    for x in range(pop_size):
        fitness[x] = (points[x] / (100.0 * pop_size))**2
```

```
        average_fitness += fitness[x]

    average_fitness = average_fitness / pop_size
    fitness.append(average_fitness)

    return fitness
```

In the above code, the fitness of a given chromosome is calculated as the square of its average score over all games played in the tournament. The reason we square the average score is so that it is easier to see variation in the distribution of fitness in the final population. The function `get_fitness` also calculates the average fitness of the entire population. It returns a list of all the fitnesses, with the average fitness of the population as the last entry.

At this point, we have a function which generates a random population, and a function that assigns a fitness to each chromosome in the population. Recall that the other two essential features of a genetic algorithm (from section 2.3) are crossover between parents and random mutation of offspring. A Python function that performs these two operations on a population of chromosomes is given below.

```
def get_new_population(population, fitness, crossover_rate, mutation_rate):

    fitness_range = [0]*(len(population) + 1)
    new_population = []
    parent_1 = ''
    parent_2 = ''
    offspring_1 = ''
    offspring_2 = ''
    mutated_1 = ''
    mutated_2 = ''

    # create a partitioned interval
    for x in range(len(population)):
        fitness_range[x + 1] = fitness_range[x] + fitness[x]

    # perform roulette-wheel sampling
    for y in range(int(math.ceil(float(len(population))/2))):

        spin_1 = random.uniform(0, fitness_range[len(population)])
        for z in range(len(population)):
            if fitness_range[z] <= spin_1 <= fitness_range[z + 1]:
                parent_1, parent_2 = population[z], population[z]
                break

        while parent_1 == parent_2:
            spin_2 = random.uniform(0, fitness_range[len(population)])
```

```
            for z in range(len(population)):
                if fitness_range[z] <= spin_2 <= fitness_range[z + 1]:
                    parent_2 = population[z]
                    break

        # determine if crossover takes place
        if random.uniform(0, 1) < crossover_rate:
            crossover_position = random.randrange(67)
            offspring_1 = parent_1[:crossover_position] + parent_2[crossover_position:]
            offspring_2 = parent_2[:crossover_position] + parent_1[crossover_position:]
        else:
            offspring_1 = parent_1
            offspring_2 = parent_2

        # determine if mutation takes place at each locus of each offspring
        for x, y in zip(offspring_1, offspring_2):

            if random.uniform(0, 1) < mutation_rate:
                if x == 'C':
                    mutated_1 += 'D'
                else:
                    mutated_1 += 'C'
            else:
                mutated_1 += x

            if random.uniform(0, 1) < mutation_rate:
                if y == 'C':
                    mutated_2 += 'D'
                else:
                    mutated_2 += 'C'
            else:
                mutated_2 += y

        new_population.append(mutated_1)
        new_population.append(mutated_2)

        parent_1, parent_2 = '', ''
        mutated_1, mutated_2 = '', ''

    if len(population) % 2 != 0:
        junk = new_population.pop(random.randrange(len(new_population)))

    return new_population
```

Notice that there are four parameters in the function heading of `get_new_population`. The `population` parameter takes a list of the chromosomes in the current pop-

ulation, `fitness` is the list containing the fitness of each chromosome (which results from calling the `get_fitness` function); `crossover_rate` is the probability that crossover will occur between any two parents, and `mutation_rate` is the probability that mutation will occur at each locus of a given offspring. The first half of the above code performs roulette-wheel sampling by creating a partitioned interval such that the length of each subinterval is equal to the fitness of a specific chromosome in the population. A random number from the interval is then generated, and the chromosome corresponding to the subinterval in which the number falls is chosen for recombination. This ensures that fitness-proportionate selection (as described in section 2.2) takes place. The second half of the code performs crossover and mutation, and then the function returns a list containing the offspring (i.e. the new population).

The last thing that we need is a callable Python function that brings all of these other functions together, so that a user can actually run experiments.

```
def prisoners_dilemma(pop_size, num_generations, num_runs, crossover_rate, mutation_rate):

    fittest_by_gen = []
    fittest_by_run = []
    fittest_current_gen = ''
    fittest_current_run = ''
    points = [0]*pop_size
    fitness = [0]*pop_size
    population = []
    offspring = []
    memory = []
    runs = []
    generations = []

    for r in range(num_runs):
        # do everything for first random generation
        population = generate_population(pop_size, 67)
        points = play_tournament(population, pop_size)
        fitness = get_fitness(points, pop_size)
        generations.append(zip(population, fitness))
        generations[0].append(fitness[-1])

        for g in range(1, num_generations):
            population = get_new_population(population, fitness,
                                            crossover_rate, mutation_rate)
            points = play_tournament(population, pop_size)
            fitness = get_fitness(points, pop_size)
            generations.append(zip(population, fitness))
            generations[g].append(fitness[-1])

        runs.append(generations)
```

```
        generations = []

    return runs
```

The parameter pop_size is the desired population size for each generation of chromosomes, num_generations is the desired number of generations per run, num_runs is the desired number of runs, and crossover_rate and mutation_rate are the same as described above (recall that 'generation' and 'run' were defined at the end of section 2.3). The prisoners_dilemma function returns a list containing the specified number of runs. In turn, each run is a list containing the specified number of generations.

Now, we can finally search for solutions to the prisoners' dilemma. Let us run an experiment of 10 runs with 50 generations each; each generation consisting of 30 candidate solutions with a crossover rate of 0.7 and a mutation rate of 0.001.

```
python: A = prisoners_dilemma(30, 50, 10, 0.7, 0.001)
```

Now, let's take a look at what happened in the first run. In particular, we will compare the initial of the first run to the final population of the first run.

```
python: for x in range(31):
>>> >>>     print A[0][0][x]
```

```
('CDDCDCCCDDDDCCDCCDCCDDDDDCCDCCCDCDDCDCDCDDDDCCCCDDCDDDCCCDCCDCCDDCC', 7.99758400000000)
('CCCDDDCCCCCDCCCCCDCCDCDCCDDCCCCCDCDDDDDCCCDDDDCDCCDDDDDCCCDCDCCDDDC', 5.55544900000000)
('CCDCCCDCDCDDDCCCCCCDCCDCCDCDCCDDDCCDDCCDCCDCDDDCCCCCCCDDDDCDCCDCDDD', 4.25321877777778)
('CCDCDCCDDCCCCDCCCCCCDCCCDDCDDDDCCDCCDCDCCDDDDDDCDDDDDCCCDCDDCDDCCDC', 5.45222500000000)
('DDDDDCCDDCCCDCDCCCDDDDCDDDDCCDCDCDDCDCCCDCCDCDCDDCCCDCCCDDCCDCDDDCD', 4.90032011111111)
('DDCCDDCDDCDDCDDDCCDDDDDCCDCDCCDCCDCCDDCDDCDCDDDDCCDCDDCCDCDDDCCDCDD', 6.90112900000000)
('DCDDDCDDDCDDDDDCCDCDDCCDCDDDDDDDCCCCDCCDDDDCCCCDCDCCCDDDDDDDCDCCCDC', 6.94322500000000)
('CCDDDCCCCDDDDDDDCDCCCDDDCDCCDCCDCDCDDCCCDDDCDDDCDCDDCCCDCCDDCCDDCCC', 7.65260011111111)
('DDCDDDDDDDDDCCCDCCDDDCDDDCCDDDCDDDDCCDCCCDCDDCCCDCDDCCCDDCCDCDCDDCD', 5.23341877777778)
('DCCCDDDCDDCCDCCDCDDDCDCDDCDDCDCDCDDDDDCCCDDCDCDCCDCCDDDDCCCDCDDDCCD', 6.39921344444444)
('CCDCCCCCDCDCDCCDDDCCCDCCDDCDCDDCCDCDCCDDDCCCDCDDDDCCCCCCDCDCDDDCDCC', 4.70600711111111)
('DDCDDCDDDCCDCCDDDCDCCDDCCCDDCCCDCDDCCDDCDCDDDCCDCCDCDDCCDDDDCCCCCCD', 6.27001600000000)
('DCCDCCDCDDCDCCDCCCDDCDCDDDDDCCDCDCDDCDCDCDCDDCDCCCCDDDDDCDDDCCDCDDD', 7.56616711111111)
('DDCCDCCCDDCCCDCDDCDCDDDCCDCCCCCCCCCCDDCDCDDDCCDCDDCDDDDDDCCCDDCDCCD', 4.82973877777778)
('CCDCCCDDCCDDCCDDDCCCDCCCDDCCDDDDCDDDCDCCCDDCDCDDDDDDDDCCCCDDDCCCDCD', 5.63745877777778)
('CDCCCDDCDDDCCCDDCCCDCCCDCDDDCCCDCDCCCDCCDCDCCCDCCDDCDDCCCCCDDDCCDCC', 7.36579600000000)
('CDCDDCDDCDCDDCDCCCCCCDDCCDCDDDCCDCCCDCCCDCCDDDDDDCDCDCCCDCCCCCCCDCD', 3.27851377777778)
('DCDDDCDDCDDCCCDCDCDCDCDCCCDDCCCCDCCDCCCDDDCCCCDDCCDCCCDCCDDCCCDCDDD', 3.94816900000000)
('CDCDDCCDCDCCDCCDCCDDCCDDCDCCCCCCCDCCDCDDDCDCCCCDDDDCCCCCDCDCDCCCCDD', 4.06963377777778)
('CCCCCCCCCCDDCCCCDDDCDDCDDCCCCDCDDDCDDDCCDCCCCDCCCDCCDCCCCDCCDDDCCDD', 4.15344400000000)
('DCDCDDDCCDCDDCCDDCDCCCDCCDCCDCDDDCCCCDDCCCDCCCCDDCCCDCDCDCCDDCCDDCD', 3.06483377777778)
('CCDDDDDDCDDDCDCDCCDCDDCCCDDDDCCDDCCCDCCCCCCCCDDDDDCCCCCCCCCDCCDDCDD', 5.46156900000000)
('DDCDCCDCDCCCDCCCDCDDDDDDCDCDDDDCCDDCCCDDCDCDDDCCCCCDDCCDDCCDCCDCDCC', 5.78883600000000)
('CDCDCCDDDDCDDCCCCDDDDCDCCCCDCDCCDCCCCDDCDCDDCDDDCDDCCDCCCCDCCCCCCDD', 6.95728544444444)
('CCDCCCCCCDCCDDCDCDDDCDDCDCCDDDDCCDDDDCCDCDCDCCDCCDDDDDCCCDDDCCDCDCC', 4.71324100000000)
('DDDCCDDDCDDDCDDCDCCDCCCCDDCDDDDCCCDDDDDCCDCCCCCDCDDDCCCDCDCDCCCCDDC', 5.95360000000000)
('DCDCDDDDDDCCCDDDDDCDDCCCCCCDDCDDDCDCDDCCDCCDCDDDCDCDDCCDCCCCDDDDCDD', 5.67392400000000)
('CDDDCCDDDDDCDCDCDDCCCDDCDDDCCCCDCCDCCDCDDCDCDDCCCCCDCDCDCDCCCDCCDDD', 6.66844544444444)
('CCDCDCCDCCDDDCDCDDCDDCCDCCDCDDCCDCDDDDCCCDDCDDDDCDDCCCCCCDCDDCCDDCD', 4.15344400000000)
('DCDDDDDCCCCDCCCDCDCDCCDDDDDCCDDCCCCCDCCDDCCCCDCCCDDCCDCDCDDDCDDDCCD', 4.67136177777778)
5.54066226666667
```

```
python: for x in range(31):
>>> >>>    print A[0][49][x]
```

```
('CDDCDCCCDDDDCCDCCCDDDDDDDDCDCCDCCDCCDCCCDDDDCCDCDDCDCDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDDCCDDDDDDDCCDCCCDDCCCDCDCDDDDCCCCDDDDDDDDCDDDCCDCDDD', 5.84994844444445)
('CDDCDDCCDDDDCDCDCCCDDDDDDDDCDCCCDDCCCDCCCDDDDCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDCCDDDDCCDCCCDDDDDDDDCDCDDCCDCCDCDCDDDDCCDCDDCDDCDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDCCCDDDDDDDDCDCCCDCDCDDCCCDDDDCCDCDDCDDCDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDDCCDDDDDDDDCDCDDCCDCCDCDCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDCCDDDDCCDDCCDDDDDDDDCDCCDCCDCCDCCCDDDDCCDCDDCDCDDDCDDDCCDCDDD', 5.84994844444445)
('CDDCDCCCDDDDCCDCCCCDDDDDDCCDCCCDDCCCDCCCDDDDCCDCDDCDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDDCCDDDDDDDDCDCCCDDDCDDCCCDDDDCCDCDDCDDDDDCDDDCDDDD', 5.84994844444445)
('CCDCDDCCDDDDCCDDCCCDDDDDDDCCDCDDCCDCCDCDCDDDDCCDCDDCDDCDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDCCDDDDCCDCCCDDDDDDDDCDCDDCCDCCDCDCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CDDCDCCCDDDDCCDCCCDDDDDDDDCDCCCDDCDCDDCCCDDDDCCDCDDCDDDDCDDDCCDCDDD', 5.84994844444445)
('CDDCDCCCDDDDCCDCCCDDDDDDDDCDCCCDCDCDDCDCDDDDCCCCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDDCCDDDDDDDCCDCCCDDCCCDCCCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDCCCDDDDDDDCCDCDDCCDCCDCDCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CDDCDCCCDDDDCCDCCCCDDDDDDDCCDCCCDDCCCDCDCDDDDCCDCDDCDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDCCCDDDDDDDDCDCCCDCDCDDCCCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDDCCDDDDDDDCCDCCCDDCCCDCCCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDCCCDDDDDDDDCDCCCDCDCDDCCCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDCCDDDDCCDCCCDDDDDDDDCDCCCDCDCDDCCCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CDDCDDDDDDDDCCDCCCCDDDDDDDCCDCCCDDCCCDCDCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDDCCDDDDDDDCCDCCCDDCCCDCDCDDDDCCDCDDCDDDDDCDDDCCDCDDC', 5.47872044444444)
('CCDCDDCCDDDDCCDCCCDDDDDDDDCDCDDCCDCCDCDCDDDDCCDCDDCDDCDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCCDCCCDDDDDDDDCDCCCDCDCDDCCCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CDDCDDCDDDDDDCCDDCCDDDDDDDCCDCCCDCCDCCDCDCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDDDDDDDCDDCCCDDDDDDDDCDCCCDDCCCDCCCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
('CCDCDDCCDDDDCCDDCCCDDDDDDDCCDCCCDDCCCDCCCDDDDCCDCDDCDDDDDCDDDCCDCDDD', 5.84994844444445)
5.83757417777778
```

In the above output, each line consists of a single chromosome on the left along with its fitness on the right. The number at the bottom-left is the average fitness of the entire population. Comparing the first and last generations of the first run, we notice a few interesting things. For one, the average fitness of the population does not change that much, with an increase of only 5.83747 - 5.54066 = 0.29681. Second, we see that there is variation in the fitness of the chromosomes in the first generation, but by the fiftieth generation all of the chromosomes have the same fitness. Another observation is that the first generation contains chromosomes that are actually fitter than any of the chromosomes in the last generation. Let's do this same type of comparison with a few more runs. In particular, we will look at the second run, and then we will jump to the eighth and ninth runs.

```
python: for x in range(31):
>>> >>>    print A[1][0][x]
```

```
('DDDDDDDDDDCDCDCDDDCDDDDCCDDDDCCCDDDDDDDCCDDCDDCDDDCDDCCDDDCDCCCDCDD', 7.09156900000000)
('DDCDDDCCDDCCDCDCDDDCDCCCCDDCCCDCDDDCCCDCDCDDCCDCDDCDCCDDCDCDCCDCCCC', 4.56392011111111)
('CCCCDCCDDDCDDDDCDCCCCDDDDCDCDCDCDCCCDCDDCDCCDCCCDDDDDCCCCCCCCDCDDCD', 5.51153877777778)
('CCDCDCCDCDDDCDCCCDDDDDDDDDCCDCDCDCDCDDDCDDCDCCCDDDDDDDCDDDCDCDCCDCC', 5.08502500000000)
('CCDDDDDDDDCCCDDDCDCDDCCCCDCDCCDDCDCCCDDCCDDDDCDCDDCCDDDDDDDDCDDDCDC', 9.28014677777778)
('DDCDCCDCDCDDCCDCCDCDCDCDDDDDDCCDCCDDCDDDCCCCCDDDDDDDCCDDCDCCCCDCDDC', 4.65984177777778)
('CCDDCDCDCDDDCCCCDDDCCDDDCCCCCCDDDDDCDDDDCCCCDCDDCDDDDDDCDCDCCDCDDCD', 6.35376044444444)
('CDDCCCCDDDCDDDCDCCDDCCCCCCCCDCDCDDDDCCCCDCDCDCCCCDCDCDDCDCCCCDDDDDD', 4.05619600000000)
('DCDDCCCCDDCCCDCCDCCDCDCDDDDDCCCDDDDCDCDDDDDDDCCCDCDCCDCDDDDDDCCDCCC', 6.22668844444445)
('DDDDDDDCDCCCDDCDCCCDDDCCDCDDDCCDCCDDDDDDCDDCCDDCDCDDDDDCDDCCCDCDCCC', 5.62322177777778)
('DDDDCCCCCDCCDDCCDDDCCCCDDCDCDDDCCCDCCDDDDCCDDCCCCCDDCCDCDDDCDDDDDCD', 5.32532544444444)
('DCCCCCCDDCDDCCCDDCCDDCDCCCCDDDCDDDDCCCCDDDCCDDCDDDCCCDCCCCDDDDCCCDD', 3.62902500000000)
('CCCCCCDCCDCDCDCDDCCDCCCCDDDCCCCCDCCCCCDDCDCDCDDCCCDDCDCCDCCCDCDDDDD', 3.64810000000000)
('CCCCCCCDDDDCCCCCCDDDCDDDCCDDDDDCCDCCCDCCCDCDCDCCDCCCCCDDDCCDDCDDDD', 4.35278677777778)
('DDDDCCDDCDCDDDDDCDCDDDCDCDDDDDDCDDDCCCCDDDCCCCCDCCDCCCCDDCDDCDDCCDC', 6.07786844444444)
('CDCDCDCDDDDCDCCDDCDDDCDCCCCDDCDCDCCCDDCCDDDCDCCCCCCCDDCDDDCCCDDCDCD', 6.33025600000000)
('CCCCDCCDDCDCCCCDDDCCDCDDCDCCDDCCCDDCDCCDDCCDCDCCDCDCCCCDDDDCCDDCCCD', 4.31254444444444)
('DCDDCDCCCCCCDCDDDCDDDCCCDCDDCCDCDCDCDCDCCDCDDDCDCCDDCCDCCCDDCCCCCCC', 2.60392011111111)
('CDDDCDDDCCCDDCCDCDCCDDDDCCDCDDDDCCDCCCDCCDDDDDCCCCCCCDCDDDDDDCCCCCC', 8.91420544444444)
('DDDDCDCCDCCCCDCDCDDDCDDDDCDDCDCDDDCDDDCCCCCDDCCCCDDDCDCCCDCDCDCCCCD', 6.04996011111111)
('CCCDCCCDDCDDDDDCCCDDDCCCDCCCCCCDCCCCDCCDDCDDCCCDDCDCDDDDDCDDCDDCDCD', 7.78782044444444)
('CDDDDCDCCDDCCCDDCDCCCDDDDDCDDCDDDDCCCCDCDCCCDDDDCCDCDDDDDCCDDDCCDDC', 4.77422500000000)
('CDCDDDDCCDCCDCCDCDDCCCDCDCCCDDDDCCCDDCDCCCCDCDCCCCCDDDDDDCCCDDDDDDC', 4.54968900000000)
('DCCDDCCDDCCCCCDCCCDCDCDCDDCCCDCCDDDDCCCCDCDCDCDCDDCDDDDCDCDCCDDCCDC', 5.86446944444445)
('CCDDCCDCDDCDDDCCDCCDCDDDCCCCCCDDDCDDCCCCCDCDDCCDCCDCCDCDDDCDDDDDCCC', 4.16432044444444)
('CCCDDDCDCDCCCCDDCCDDCDCCCCCCDCDCDCCCDCCDDDDDCDCDDDCCCCDDDDDDCCCDDCD', 5.93409600000000)
('CDCCCDCCCCDCCCCCCCDDCDDCCCCDDCDCDCDDDDCCDCDDDCDCDCCDDDCCDCCCCDCDCDD', 6.25333377777778)
('CDCCCDDDDCCDCDDDDCDCDDCCDCCDDCDDCCCDCCDDDDDDDDDCDCCCDDDCCCCDCCDCDDD', 7.74508900000000)
('CDCDCCCCDCDDCCDCDCDDDCCDDCCCDDCDCDDCCCCCCCCCCDCDDDDDDCCDCDDCDDCCDCC', 3.84552100000000)
('DCDDCDDDDDDCDDCDCCCCDCDCCCCCDDDDCCDCCDDDCDCDCDDDDDCCDCCCCDCDDDCDCCC', 3.53941511111111)
5.47179597037037
```

python: for x in range(31):
>>> >>>   print A[1][49][x]

```
('CDDDCCDDCDCDDDDCCDCDDCCDDDDDDDCCCDDCCCDCDCDCCDDCDDCCDDDDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDCDDCCDDDDDDDCCCDCCCCDCDCDDDCDCDDCCDCDDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDDDDCDDCDDDDCCDDDDDCCDDDDCCCDCDCDDDCDCDDCCDCDDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDDDDCCDDDDDDCDDDDDCCCDCDCDDDCDDDDCCDDDDDDDDCDDDCCC', 9.61000000000000)
('CDDDCDDDCDCDDDDDCDDDDCCDDDDDDDDCDDDCCCDCDCDCCDDCDDCCDDCDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDDDDCCDDDDDCCDDDDCCCCDCDCDDDCDCDDCCDDDDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDDCDDDDCDDCDDDDCCDDDDDCCDDDDCCCCDCDCDDDCDCDDCCDDCDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDCCDCDDCCDDDDDDCDDDDDCCCDCDCDCCDDCCDCCDDCDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDCCDCDDCCDDDDDDDCCCDCCCCDCDCDDDCDCDDCCDCDDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDDDDCDDCDDDDCCDDDDDCCDDDDCCCDCDCDDDCDCDDCCDCDDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDCDDCDDDDCDDDDDDCCDDDDCCCDCDCDCCDDCDDCCDDCDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDCCDCDDDDDCDDDDCCDDDDDDDDCDDDCCCDCDCDDDCDCDDCCDDCDDCDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDCDDCCDDDDDDDCCCDDCCCDCDCDCCDDCDDCCDCCDDDDDCDDDCCC', 9.61000000000000)
('CDDDDCDDCDCDDDDCCDCDDCCDDDDDDDDDDDDCCCDCDCDDDCDDDDCCDDDDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDCCDCDDCCDDDDDDCDDDDDCCCDCDCDCCDDCCDCCDDCDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDCDDCDDDDCDDDDDDCCDDDDDCCCDCDCDCCDDCDDCCDCCDDDDDCDDDCCC', 9.61000000000000)
('CDDDDCDDCDCDDDDCCDCDDCCDDDDDDDDDDCCCCDCDCDDDCDCDDCCDCDDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDDDDCDDCDDDDCCDDDDDCCDDDDDCCCDCDCDDDCDDDDCCDDDDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDCDDCDDDDCDDDDDDCCDDDDCCCDCDCDDDCDCDDCCDDCDDCDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDCDDCCDDDDDDDCCCDDCCCDCDCDCCDDCDCCCDDDDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDCCDCDDCCDDDDDDCDDDDDCCCDCDCDDDCDCDDCCDCCDDDDDCDDDCCC', 9.61000000000000)
('CDDDDCDDCDCDDDDDCDDDDCCDDDDDCCDDDDCCCCDCDCDCCDDCCDCCDDCDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDDDDCDDCDDDDCCDDDDDCCDDDDCCCCDCDCDDDCDCDDCCDDCDDDDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDCDDCDDDDCDDDDDDCCDDDDCCCCDCDCDDDCDCDDCCDDCDDCDDCDDDCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDDDDCCDDDDDCCDDDDCCCCDCDCDDDCDCDDCCDDDDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDCDDCCDDDDDDCDCDDDCCCDCDCDCCDDCCDCCDDCDDDDDCDDDCCC', 9.61000000000000)
('CDDDDCDDCDCDDDDDCDDDDCCDDDDDCCDCDDDCCCDCDCDDDCDCDDCCDDCDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDDDDCCDDDDDDDDCDDDCCCDCDCDCCDDCDDCCDDCDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDDCDDDDCDDCDDDDCCDDDDDCCDDDDCCCCDCDCDDDCDCDDCCDDDDDCDDCDDCCCC', 9.61000000000000)
('CDDDCCDDCDCDDDDDCDCDDCCDDDDDDDCCCDDCCCDCDCDCCDDCDDCCDCDDDDDDCDDDCCC', 9.61000000000000)
9.61000000000001
```

```
python: for x in range(31):
>>> >>>    print A[8][0][x]

   ('DCDDDCDDCCCDDCDDDDDCCDDDCDDDDCCDDCDDCDDCDCCCCCCCDDCCDDCDDCCDCDCCCCD', 6.22336177777778)
   ('CCDDDCDCCDDCDDCCDCDDDCDCCDCCDCCDCDDDDCDDCCDCDCDDDDDCCDCDDDCDCCCDDCDCD', 5.87416011111111)
   ('DCDDCCDCDCCCDCDCCDCCCCDCDDCCCCDDCCDCDDCCCDDCCCDDDDCDDDCDCDDDDDCDDCC', 4.05888177777778)
   ('CDCCDCDDDCDDCDDDCCDDCDCCCCDDCCCCDDDCCDDDCCDCCCDDDDDCDDDCDDDDCDCCDDD', 7.22892844444444)
   ('CCDCDCDDCCCDCCCCDCCCDCDCDCDCDCDCCDDCCDCDDDDDCDCDDDDCDDCDCCDDDCDCCDD', 5.56488100000000)
   ('DCCCCCDCCCCCDCDDDDDCDCDDDDDDDDDCCCCCDDCDCDCDCCDDCDDDCDCDCCDDCDDCCCD', 5.76960400000000)
   ('CCCCCDDDCDCCDCCCDDCDDDCDCCDDCDDCCCCCDDDDCDCDCDCDDCCDDCCDCCCDCDCDCDC', 3.72361344444444)
   ('CDDDDDCCCCCCCCCCDDDCDDDCDDCDDCCCDDCCDDDCCCDCDDDDDDCCCDDCCDCDCDCDDCC', 4.01868844444444)
   ('CCCDDCDDCCCDDDDCCDDCCCCCCDCDCDCCDCCCDCDDDDCCDCCDCDDDCCDDDCDCDDCDDDC', 4.11008044444444)
   ('DDDCDDCCCDCCCCCDCCCCCDDDDDDCDCDDCDDCCDDCDCDCDCCDDDCCCCCCCCCDCDCCCDC', 5.53033611111111)
   ('DDDCCCDCCDDDDCCCCCCDCCCDDCDDCCCDCDCCDCDCDCDCDDCCDDCDDDCDCDDDDCDDDDD', 6.23001600000000)
   ('CCCDDDCCCCCCDDCDDCDCCDCDCCDDDCDDCCCDCDCDDCDCCDCDDDDCCCDDDDCDCDDDCDC', 5.36076844444444)
   ('DDCCDDCDCDDCCCCCCCCCCCDDCDDDCCDDDCCCDCCCCCCCCCCDCCDDDDDDDDDDDCDDCDD', 5.11212100000000)
   ('DCDCCCDDCDCDDDDCCDCDCCCCDCCDDCDDDCCCCDCCDDCCDCCDCCCCCCDDCDDCCDDCDCD', 3.93096711111111)
   ('DCDDDCCCDCDDCCCCCDDCCCCDDDCCDDDDDDCCDCCDCDCCCCDDDCCDDDCDDCCDDDCDCDD', 4.37367511111111)
   ('CDDCDCCCCCDDCDDDDCDCCCDDDCDCDDCDCCCDCCCCCDDDDDDCDDCDDDDCDDCDDCCDCD', 5.96824900000000)
   ('CCCCDCDCDCDDDCDDCCCDCCDCDDDCCDCCCCDCCCCCCCDCDDCCDDDCDCDDDCDCCDDCDDC', 4.34305600000000)
   ('CCDDDDDDDDDCDCCCDDDCCCCDDDDDCCDDCCDCCCDCCCDDDDCDDCCDDCDCCDCCCCDDDCC', 4.79172100000000)
   ('CCCCDCDCCCDDDDDCDDCCDCDCCDDCDCCCCCCDCCDDDCDCCCCDDDDDDDDCDCDCCDCCDDD', 3.63156544444444)
   ('DDCCCDCCCCDDCDDDCDDDCCDCCDCCCDCCCCCDCCCDDDCDDCCCCDDDCCCDDCCDCCDCC', 3.69280277777778)
   ('CDDDCCDDCCDDCDDDCCDDDDCDDDCDDDCDCDCDCDCCCCCCDCCDCDCDCCDDDCDDDCCCCCD', 6.39921344444444)
   ('DDDCCDDCCDDDDDDDDDCDDCCDDCCDCCCCDCDDCDDDDCDCCDCCCCCCDCDDCCCCDDCDCCD', 5.72485377777778)
   ('CDDDCCCCDCCCDDDDDCCCCDDDCCCCDCCCDDDDDDCDCDCDDDDDCCCDDDDCDCDDCDDCCDC', 6.02866177777778)
   ('CDCDCDCDCCDDCCDCDDCDCDDCCDCCDCCDCCDCCDDCDCDDCDDCCCCCCCCCDCDDDDCDCCDC', 4.97884844444444)
   ('DCCCCCDDCDCDDCDDDDCDDCDCDCDCCDCDCDCDCCDDCCDDDCDDDDDDDDDCCDDDDDCDCDDC', 6.94673877777778)
   ('DDCCCCCCCCDDCDDDDCDCCCDCDCCCCCDCCDCDDCDCCCCDDDCDCCDCDCCCCDDDDCCDDDCC', 3.55699600000000)
   ('CCDDCDCDDDCDDCDDDCCDDDCDCCCCCCCCDCCDDDDDDCCDCDDCCDCCDCDCDCDCCDDCCDD', 5.24562677777778)
   ('DDDCCDCDDDCCDCDDCDDCDCDDDDCCCDDCDDDCDDCDCDCCCDDDDCDDCDDDCCDDDCDDCDC', 5.46624400000000)
   ('DDCDDCCDDDCCCDCDDCCDCDDCDDCCDCDDCCDDDCCCDCCCDDDCCCCCCCDCCCDCDCCDCDC', 4.93728400000000)
   ('DDCCDCCCDDDCCCDCDDDDCDDCDDDDCCCCDCDDDCCCDCCDDCCDCDCCDCCDCCDCCCCD', 6.06636900000000)
   5.16294378148148

python: for x in range(31):
>>> >>>    print A[8][49][x]
```

```
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDCDDCCCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDDDCCCDDDCCCDCDDDDCCCDDDCDDDCDCDDCDDCDCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDCDDCDDCDCDCDDCCDCDCCCCCDDCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDDDCCCDCCDCCCDCDCDCCDDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCCDCCDCDDCDDCDCDCDDCCDCDCCCCCDDCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDDDDCDDCDCCDDCDCDCDCDCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDDDDCDDCDCCDDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDCDDCDCDCDCDCDCDCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDDDDCDDCDCCDDDDCDCDCDCCCDDCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDCDDCDDCDCDCDDCCDCDCCCCCDDCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCDDDCDCCCDCDCDDCCCDCCDCCCDCDCCCCCDCCCCCCCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDDDDCDDCDCDCDDCCDCDCDCCDDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDDDDCDDCCDCCDCCCDCDCDCCDDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCDDDCDCCCDCDCDDCCDCDCDDCCDCDCCCCCDCCCCCCCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCDDDCDCCDCDDDCDDCDCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDDDCCCDDDCCCDCDDDDCCCDDDCDDDCDCDDCDDCDCDCDDCCDCDCCCCCDDCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDDDCCCDCCDCCCDCDCDCCDDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDCDCDDCCCDCCDCCCDCDCDCCDDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDCDDCCCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDCDCDDCCCDCCDCCCDCDCDCCDDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCDCDDDCDDCDCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDCDDCCCCDDDDCDCDCDCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDCDDCDCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDDDCCCDCCDCCCDCDCDCCDDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDCDDCDCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDDDDCDDCDCCDDCDCDCDCDCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDDDDCDDCDCDCDDCCDCDCCCCCDCCCCCCCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDCDDCDCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCDCDDCDDCCDDDDCDDCDCDCDDCCDCDCCCCCDCCCCCCCCDDD', 9.61000000000000)
('CCDCDCCCDDDCCCDCDDDDCCCCCDDDCCDCDDDCDDCDCDCDDCCDCDCCCCCDCCCCCDCCDDD', 9.61000000000000)
9.61000000000001
```

```
python: for x in range(31):
>>> >>>     print A[9][0][x]
```

```
('DDCCCDDCCCCDDCCDCDCCDCCDCCDDDCCDCDDCDDDDDCDCCDDCCDCCDCDCDDCCDCDDCDC', 3.56580277777778)
('CDCCDDDDCCDCDDDCDCDDCCCCDCDCDCDCDCCDDDDCDCDCDCCDDCDDDCCCCDCCCDCDDCD', 7.80643600000000)
('CDCDDDDCCCDDDCDDCCDDCCDCCDCCCCCDCDDDCDDCDDCCDCDCCDDDCDCDCCDDCCDCDDC', 6.26500900000000)
('CDDCDDCCDCCCCDDDCCCDDDCCDCDDCCDCCDDDCCCCCCDCDDCCCDCDCDCCDCCDCCDCDD', 4.80778711111111)
('CDDCCCDCDDCDCCDCCCDDCCCCCCCCCCCDCCDCCCDCDDCDCCCCDCCDCCCCDCCDDCDCDCC', 3.74809600000000)
('DCCDDDCCDDDDDCDDDCDDCCCCDCDDDCCDCCDCDCDCCDCCCCDCCCCDDCCCDDDCDDDCCDC', 5.37466944444444)
('DCDCDCCDDCCCCDDDDCDDDDCDDDCCCDDCDCDCDDCDCCDCDCDCDDDDCCDDDCDDDCDCDCC', 8.61031211111111)
('DCCDCCCCCDCDDCDCCCCCDCDDCDCCCDCDDCDDCCDDCDCDDDCCCCDCCCCCDCCCDCDDCCC', 3.59734444444444)
('DCCCCDCCCDDDDCCDDCDCCDDDDDCDCCCDDCCCDDDDCDCDDCCCDCCCDCCCCDCDCCDDDCC', 5.63904177777778)
('CCDCDDCCDCCDCDDDDDCDCDDDDDDCDDCDDDCDDDCCCDDDDCDDCDCCDDCCCDCDDDCDDCC', 7.68398400000000)
('DCCDDDDDCDDDCCCDDDDCDCCCDCDDDCCCDDCDDDDDCCCDCDCCCDDCDCCDDDDCDDCCDDC', 6.02211600000000)
('CCDDDDCCDCDDCCCCCCCCDCCDDDDDCDCCCDCCCDDDDDCDCDCCCDDCDCDCCCDCCDDCCDD', 4.55537877777778)
('CCDCDCDDDDCDCDDDDDDCDDDCCDCDDCDCCCCDDDDDDDCCDCCDCCDDCCDCCCDDDCCDDDC', 5.68186677777778)
('CDDCCDDDDCDDDCDDDDCCCCCDCCCCDCCCCCDCCCDCDCDCDCDCCDDDDDDDCCDCCDCDCDC', 4.92692011111111)
('DCCDCCDDDDCCCCCCDDCCDDCCCDDCDDDDCCDCDCDCCCCCDCCDDCDCDDCCCDDCCDDCDCD', 4.22987777777778)
('CCDCCDCDCDCCCDCDDCCCDDCDCDCCDDCDDCDDDDCCCCDDDDCCDCDDCCCCCCDCCDCDDDC', 5.11966044444444)
('DDDCDDDDCCDCDCCDDDDCCDDDCCDCDDCCDDCCDDDCDCCCCCCDDCDDCDCDCCDCDCCCDDD', 6.01557377777778)
('DCDCCDCCCDDCDDDDCCDCDDCDDCCCCCCCCCCDDDDCCCDDDCDCDCDCDDDCCDDCCCDDCDD', 8.05424400000000)
('DDDCDCDCDDCDDDDCCCDDCCCDCCCDCCDDCDDCCCDCDCDDDCDCCCDDDCCCDCCDDCDCCCC', 4.26422500000000)
('DDDDCDCDCCDDDCCCDCCDDCCDCCDCCDCDDDDDCCCDCCDCCCCDDCCDCDDCCCCDCDCCDDC', 7.27560711111111)
('CCDCCCDCCDCDCDCDDCCDCCDCCDCDDDCDDCDCCCDDDDDDCCCCCCDDDDDDDDDCCCCDCDD', 5.24562677777778)
('CCDDDCDDDDDCCDCCDDCCDCDCDDCCDDCCCCDCCCDCDCCCCDDCDCCDCDCCDCCDDDCCCCD', 5.44600011111111)
('DDCCDDDDDDCCDCCDCDCCCCCCDCDCCDCCDDCCDDCCCCCCDDCCDDCCCCCCDDCCDCCCDCC', 5.21361111111111)
('CDCCCDCCCDCDDDDCDDDCCCCDCCDCCDCCCDDCDDCDCCCCDCCDCCCDDCDCCDCCDCCDCDC', 4.66704011111111)
('DDCDDCCCDCDDCCCCDDCDDDDDCCDDDCDDDCCCDCDDDDCDCDDCCCDCCCCCCCDDDCDDCCD', 7.46200277777778)
('CCDCCDDCCDCCDDDCCCCDCDCDCDCDCCDDCCDCDCDDDCDDCCCCCDDCDDCDCDCDCDCDDDC', 5.77601111111111)
('CDCCCDDDDDCDCDDCDCDDCDCCDDCCCCCDCCCDDCCDCDCDCDDCDCDCCCCDDDDDDCDCCDD', 6.62376011111111)
('DDCDDCDDCDCDDCDCCCCDCCDCCDCDDCDCCCDDDCDDCDCDCDCCCCDDCDCCCCDCDCDDDCC', 4.92396100000000)
('DCDDCCDDCDDCCDDDCDDCDCDCDDCCCCDDCDCCCDCDDDCDDCDDCDDDCDDDDCCDDCCDCCC', 4.81948844444444)
('CCCCCCCCDCCCDDCCDCCDDCDDCDCCDDDDDDDDDDCCDDDCDDCCCCDCDCDCCCDDCDCDDDC', 5.69140544444445)
5.63709531481481
```

```
python: for x in range(31):
>>> >>>    print A[9][49][x]

    ('CCCDDCDDCDDDCCCCDDCDCDDDDDCCCCCCCCCCDCDDCCDDDDCDCCCDDCCCCCCDDDCDDCCC', 9.61000000000000)
    ('CDCCDDCCCCDDDCDDDCDCCDDDDCCCCCCCCCCDDDDCCDCDCDCDCCCDDCDCCCCDDDCDDCCC', 9.61000000000000)
    ('CDCDDCCDCDDDCCCCDDCDCDDDDDCCCCCCCCCCDCDDCCDDDDCDCCCDDCCCCCCDDDCDDCCC', 9.61000000000000)
    ('CCCDDCDDCDDDDCCDDCDCCDDDDDCCCCCCCCCCDCDDCCCCDDCDCCCDDCCCCCCDDDCDCCCC', 9.61000000000000)
    ('CDCDDCDDCDDDCCCCDDCDCDDDDCCCCCCCCCCDCDDDDDCCCCDDCDCCCDDCCCCCDDDCDDCCC', 9.61000000000000)
    ('CDCCDCCCCCDDDCCDDCDCCDDDDDCCCCCCCCCCDCDDCCCDDDCDCCCDDCCCCCDDDCDDCCC', 9.61000000000000)
    ('CDCCDCDDCDDDCCCCDDCDCDDDDCCCCCCCCCCDCDDCCDDDDCDCCCDDCCCCCDDDCDDCCC', 9.61000000000000)
    ('CCCDDCCCCCDDDCCDDCDCDDDDDCCCCCCCCCCDCDDCCCDDCDCDCDDCCCCCCDDDCDDCCC', 9.61000000000000)
    ('CDCDDCCDCDDDCCCCDDCDCDDDDDCCCCCCCCCCDCDDCCCDDCDCDCDDCCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCCDCCCCCDDDCDDDCDCCDDDDDCCCCCCCCCCDCDDCCCDDDCDCCCDDCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCDDCDDDCCCCDDCDCDDDDDCCCCCCCCCCDDDDCCDCDCCCDDCCCCCDDDCDDCCC', 9.61000000000000)
    ('CDCCDCCCCCDDDCDDDCDCCDDDDDCCCCCCCCCCDCDDCCCDDCDCCCDDCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCCDCDDDCCCCDDCDCDDDDDCCCCCCCCCCDCDDCCCDDCDCDCDDCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCDDCDDDDCCDDCDCCDDDDDCCCCCCCCCCDCDDCCDDDDCDCCCDDCCDCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCCDCDDDCCCCDDCDCDDDDDCCCDCCCCDCDDCCCCDDCDCDCDDCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCDDCDDDDCCDDCDCCDDDDCCCCDCDDDCDCCCDDDCDCCCDDCCCCCDDDCDCCC', 9.61000000000000)
    ('CDCDDCDDCDDDDCCDDCDCCDDDDCCCCDCDDDCDCCCDDDCDCCCDDCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCCDCDDDCCCCDDCDCDDDDDCCCCCCCCCCDCDDCCCCDCCDCCCDDCCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCCDCDDDCCCCDDCDCDDDDDCCCCCCCCCCDCDDCCCDDDCDCCCDDCCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCCDCCCCCDDDCCDDCDCCDDDDDCCCCCCCCCCDCDDCCCCDDCDCCCDDCCCCCCDDDCDCCC', 9.61000000000000)
    ('CDCCDCCCCCDDDDCCDDCDCCDDDDCCCCCCCCCCDCDDCCDDDDCDCCCDDCCCCCCDDDCDDCC', 9.61000000000000)
    ('CCCDDCDDCDDDCCCCDDCDCCDDDDCCCCCCCCCCDDDDCCDCDCDCCCDDCCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCDDCDDDCCCCDCDCCDDDDCCCCCCCCCCDCCCDDCDCCCDDDCDCCCDDCCCCCCDDDCDDCC', 9.61000000000000)
    ('CCCDDCDDCDDDCCCCDDDCCDDDDCCCCCCCCCCDCCCDDCDCCCDDDCDCCCDDCCCCCCDDDCDDCC', 9.61000000000000)
    ('CCCDDCDDCDDDCCCCDDCDCDDDDDCCCCCCCCCCDCDDCCDDDDCDCCCDDCCCCCCDDDCDDCC', 9.61000000000000)
    ('CCCDDCDDCDDDDCCDDCDCCDDDDCCCCCCCCCCDCDDCCDDDDCDCCCDDCCDCCCDDDCDDCC', 9.61000000000000)
    ('CDCDDCDDCDDDDCCDDCDCCDDDDCCCCCCCCCCDDDDCCDCDCDCDCDCCCCCCCDDDCDDCC', 9.61000000000000)
    ('CDCCDDCCCCDDDCDCDCDCCDDDDCCCCCCDCCDDDCDCCCDDDCDCDCDCCCCCCDDDCDDCCC', 9.61000000000000)
    9.61000000000001
```

In run two, we see that the average fitness of the population increases by more than 4 points, and that the final generation includes chromosomes with a higher fitness than any chromosome in the original population. In runs eight and nine, we see essentially this same behavior with respect to the average fitness and the maximum fitness in the different generations. Thus, we see that most of the time the population gets better at playing the prisoners dilemma, as a whole.

The one thing that all four runs have in common is that, in the final generation, all of the chromosomes have the same exact fitness. What makes this phenomenon a bit puzzling is that even though all of the chromosomes in the final generations have the same fitness, they are not all using the same exact strategies (as can be verified by noting that the chromosomes do not all have the same string representations). Thus, it seems that with the GA that we are using, populations tend to evolve into communities of equally skilled individuals rather than populations with *extremely* skilled individuals.

This diverges from what Axelrod found in his experiments, where certain strategies clearly stood out as scoring higher than others. I think this disparity between our experiment and Axelrod's is mainly due to the fact that our environment is dynamic, while his was largely static. That is, the chromosomes in each generation of our experiment are competing directly with one another, whereas the individuals in Axelrod's experiments were only competing directly with four strategies that remained unchanged from generation to generation.

5 Further Research

The next step in this research would be to implement an experiment with a more static environment, where the individuals are not playing the prisoners' dilemma against one another directly. It would be interesting to compare the outcomes of a GA with that kind of environment to the outcomes seen in this paper.

Also, it would be interesting to see what happens if we change some aspects of our genetic algorithm. For instance, what would happen if we did not require each chromosome to play against itself in the round-robin tournament? Perhaps this would allow some individuals to break away from the pack and earn a higher fitness. I would also like to try restricting the memory of the candidate solutions to just a single game. This would make the chromosomes' string representations much shorter, as the strategies would not be as complicated. If nothing else, shorter bit strings would make it easier to analyze the strategies themselves. Since we used a memory of three games in this paper, it was not really possible to visualize if a strategy was mainly "TIT FOR TAT", for instance.

References

Mitchelle, Melanie. *An Introduction to Genetic Algorithms.* Cambridge, MA: MIT, 1996. Print.