Exploration of Genetic Algorithms Through the Iterative Prisoner's Dilemma

Aaron Dufour
TJHSST
Alexandria, Virginia

May 27, 2008

Abstract

Genetic algorithms are used for many optimization problems to find a near-optimal solution when finding the optimal solution would be too time-consuming. Although unable to tell when it has found the optimal solution, a genetic algorithm continues until the probability of having found the optimal solution is sufficiently high. There are many methods used to perform each step of a genetic algorithm, but it is not easy to identify which will work best for a specific problem. The goal of this project is to compare these different methods through the iterative prisoner's dilemma, and to hopefully find which methods work best generically.

1 Introduction

The prisoner's dilemma is a problem involving two players. The players each must decide whether to cooperate with the other or to defect. These decisions must be made without knowledge of the other player's decision. Points are given to each player based on the moves they made. If both players cooperate, they are each given R points. If one cooperates and the other defects, the cooperating player receives S points and the defecting player receives T points. If they both defect, they are each given P points. In order for it to be a prisoner's dilemma, the values must follow the inequality T > R > P > S.

In the iterative prisoner's dilemma, the additional inequality 2R > T + S must be satisfied. In this scenario, the same thing happens, except that the players are against each other many times with memories of the past. In this scenario the best outcome is for both players to cooperate each time, because the total points given when both cooperate (2R) is greater than the number of points given if one defects (S + T) and greater than the number of points if both defect (2P). The problem associated with the iterative prisoner's dilemma is to find the rule that should be followed in order to maximize the number of points received when it participates in this scenario with a variety of other players.

This is a good problem on which to use a genetic algorithm because there is no algorithm faster than brute force that has been proven to find the optimal rule. In my genetic algorithm, I made each solution a collection of bits that represent whether the player should cooperate or defect given a past collection of turns. The fitness value for each possible solution is the number of points it accumulates after going through a set number of turns with each other possible solution in the population. The methods by which the each part of the genetic algorithm is done can be changed easily because each possible solution is a simple string of bits.

2 Background

The iterative prisoner's dilemma has been studied extensively in the past. Because the best rule is agreed upon, it is a good case with which to test genetic algorithms. It has been shown that the best rule is to cooperate on the first turn, and then do the same thing that the opposing player did on the previous turn for the rest of the turns. The only exception is if the opposing player defected the previous turn, there should be a 3% chance for the rule to tell the player to cooperate instead of following the opposing player. This rule was found by Robert Axelrod through a series of tournements in which he invited colleagues to devise rules for the iteratve prisoner's dilemma, and then had them all play against each other. Many people have written a genetic algorithm that solves the iterative prisoner's dilemma, but I have not found a case in which this problem was used to study genetic algorithms.

3 Development Sections

My program can run a genetic algorithm to find a solution to the iterative prisoner's dilemma using many different genetic algorithms methods. It can take user input to tell it which method to use for each part of the algorithm, and can set essential constants such as the mutation rate, the number of generation, the size of the population, etc. As the program runs, it displays a graph showing the fitness of each of the current solutions. It also shows a graph that shows the average fitness of each previous generation. After the genetic algorithm completes its run, another method looks at the average fitness values and determines at which generation the fitness level stabilized. This represents how long it took the genetic algorithm to find the optimal solution. After many genetic algorithm runs with different methods and constant values it should become obvious which is most efficient. Finally, it outputs a file with these average fitness values so that they can again be graphed after the program has completed the run. I will use averages of the number of generations each method takes to get to the optimal solution in order to judge the usefulness of each method. Before I can do this, I will need to write a program that determines when the optimal solution was reached, because the mutations prevent this from being a simple preocedure. My results will be graphs of how different genetic algorithm methods compare against each other. This will hopefully aid in the deciding of which genetic algorithm methods to use by future programmers.

4 Bibliography

Do not Match, Inherit: Fitness Surrogates for Genetics-Based Machine Learning Techniques
The Evolution of Cooperation
Mediation of Prisoner's Dilemma Conflicts and the Importance of the Cooperation Threshold
Genetic Algorithms - A Tool for OR?
n The Genetic Algorithm and the Prisoner's Dilemma