Iterative Prisoner's Dilemma Problem with Genetic Algorithms

Grady Bosanko undergraduate student
Union College
Schenectady, NY 12308

Abstract–**In this paper, we attempt to recreate the findings of Robert Axelrod in using genetic algorithms to optimize the iterative prisoner's dilemma problem. The iterative prisoner's dilemma problem is a famous game theory problem that ties into many economic principles regarding how individuals cooperate and compete given their environment. By optimizing a static environment, we may gain insight into the true nature of this problem. Axelrod found that the genetic algorithm outperforms other popular algorithms such as TIT FOR TAT, and our findings agreed, but our findings found a much greater difference between the two algorithms. We explained a few of the reasons as to why this may be, but overall we were able to successfully recreate the experiment performed by Axelrod. Both our findings and the findings of Axelrod show that genetic algorithms are extremely effective in this problem because it is a static environment.**

I. Introduction

In today's craze for artificial intelligence, it's easy to forget the humble beginnings of thinking machines. The Turing test [1] states that a machine is capable of thought if its outputs can mimic those of a human to the point that an observer cannot tell the difference between the man and the machine. The same criteria can be applied to a multitude of different problems, aside from natural language generation.

But why use artificial intelligence in the first place? Computers are capable of running calculations at a scale far greater than a human mind. In the time it takes for a person to come up with a singular solution to a problem, a computer could come up with hundreds, or even thousands. However, a majority of these solutions will most likely not be particularly good solutions.

In order to utilize this computational power, we first looked towards random mutation hillclimbers (RMHCs) [2]. A random mutation hillclimber starts with a random parent solution. Then, this parent is randomly mutated or changed slightly to create a child. Both are evaluated according to a fitness function, and whichever solution scores better becomes the new parent. This can go on indefinitely, but usually goes for a given number of generations.

This term "fitness function" [3] means the function by which a given solution has its viability measured. So for example, if the problem is maximizing a 6-bit binary string, the most viable solution would be "111111". We call the viability of a solution its "fitness" [3], thus the name fitness function.

The RMHC takes only a couple small concepts from biology, that being the notion of mutation, parents, children and generations. To create something more lifelike, perhaps it is necessary to use more biological parallels. A genetic algorithm (GA) [3] [4] does exactly this. GAs work by creating populations of solutions, selecting parents, and creating offspring from those parents using crossover and mutation to create variability in the next generation.

This variability is key for finding the best solutions. Evolutionary algorithms like GAs are able to create diverse and unexpected solutions [5] because of this variability. Even more fascinating is how many parallels can be drawn between this type of evolution and biological evolution.

By adding constraints to the fitness function, solutions begin to fill interesting niches. For example, adding a constraint on reproductive rate could cause solutions to "play dead" [5], or by adding some competitive restraint to the fitness function, organisms could take on parasitic [5][6] tendencies.

In creating artificial intelligence, it is unreasonable to begin with a complex intelligence [7] such as the human intelligence. Rather, something simple like an insect's intelligence is more on par. In this case however, we examine an infinitesimal piece of the human intelligence such that the scale of this feat is within reason.

		Player B: Cooperate	Player B: Defect
Player A	Cooperate	3, 3	0, 5
Player A	Defect	5, 0	1, 1

Fig. 1: The payoff matrix for the Prisoner's Dilemma (adapted from [3]). The scores in each box are for Player A and Player B respectively given each outcome.

The prisoner's dilemma is a famous game theory problem in which two partners in crime are arrested and interrogated separately. If they each stay silent, the cannot be fully convicted, and will serve lesser sentences. However, if one party tells on the other, while the other party stays silent, then

the former party will be set free while the latter party receives a harsher sentence. Finally, if both parties tell on each other, they will both receive moderate sentences. Figure1 shows a visualization of these outcomes, with higher numbers indicating a better outcome for the individual.

In an isolated instance of the prisoner's dilemma, it is on average better to defect. However, the iterative prisoner's dilemma turns this isolated instance into a continuous-time problem. The iterative prisoner's dilemma works by going through the steps of the prisoner's dilemma, except now the parties are awarded points for the outcome. Figure1 shows these exact point values for each outcome. Parties play this prisoner's dilemma problem game, gain points, and repeat for a given number of rounds, hence the name "iterative prisoner's dilemma".

The work in [8] combines the notions of the iterative prisoner's dilemma and GAs. In this experiment, we will attempt to recreate his findings in how GAs can be used to optimize the iterative prisoner's dilemma game, how the GA player compares to other popular strategies, and we will analyze key differences between our results and implementations.

II. Methods

A. Game Logic

One round of the iterative prisoner's dilemma game starts with each player giving an input, either "C" for cooperate or "D" for defect. Then, those inputs are concatenated, and the result is calculated based on Figure1. So if both groups cooperate for example, both players receive three points for that round.

Of course, the iterative prisoner's dilemma is played over more multiple rounds. So, this logic for playing one round is simply repeated for a given number of rounds. The players accumulate points over the course of the game, and at the end, a winner is determined based on who has the most points.

Because this game takes two players, we must then consider which players to use. For our purposes, one player will be the GA player. Then for the other players, [8] used algorithms that competed in a tournament as the other players and used a TIT FOR TAT strategy as a baseline to compare the results to. The TIT FOR TAT strategy looks only at the previous input, and mirrors the move made by the opponent. So if last round the opponent cooperated, TIT FOR TAT will cooperate for its next input.

We will instead use a pattern player and a simple AI player. The pattern player will be given a set pattern, "CCDD" for example, and will repeat these inputs in this order for however many rounds the game is played. The simple AI player will function very similarly to TIT FOR TAT, but will instead look at the previous three inputs. From these inputs, it will look to see if the opponent has cooperated more or defected more. Then, it will mirror whichever input its opponent has done more often.

B. Genetic Algorithm Player

The GA player will have a 70-bit string genome as described in [3][8] which will define its outputs given a set of three previous inputs. We will achieve this by concatenating the previous three inputs, converting them to binary, and decoding the binary to use as an index, as demonstrated by Figure2.

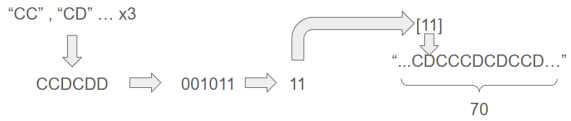

Fig. 2: The flow of how inputs are translated to an output from the GA. Three inputs are concatenated, translated to binary, decoded from binary, and used as an index for the genome.

This accounts for the first 64 bits, and the remaining six bits are used to determine the output if there are not three previous inputs available. This occurs in the first three rounds of the game. The first move is always predetermined as "C", then there are two possible outcomes with one known input, and four possible outcomes with two known inputs, thus giving us the remaining six bits.

C. Genetic Algorithm

The parameters used for the GA are adapted from [8], with the main change being that we used 100 generations instead of 50. Otherwise, we are still using 40 runs, 20 genomes per population, and 70-bit genomes. These individuals undergo mutation at a one in one-hundred chance per locus, where the mutation will randomize the selected locus to either "C" or "D". They will also undergo single-point crossover, where a single locus is selected and the genomes of two parents after this locus are swapped to create two unique children.

1) For a given genome in the population
   a) Play 151 rounds against each pattern player
   b) Play 151 rounds against the simple AI player
   c) Add up GA scores of these games (weighted equally between pattern and AI players)
   d) Average total scores
   e) Return this average score

TABLE I: The pseudocode outline of the fitness function used (adapted from [8]). The patterns used were "DDCC", "DC", and "DCC".

Each individual in each population must be evaluated by the fitness function described in TABLE I. Essentially, a GA player is created using the given genome to be evaluated, and this GA player faces off against three pattern players and the simple AI player. The scores from the match against the simple AI player is weighted three times as heavily so that it has equal weight to playing against the three different patterns. The weighted average of scores is then used as the fitness value for the individual.

The process shown in TABLE II is repeated for all 100 generations. The fitness proportional selection works by

1) Create randomized initial population
2) For 100 generations of population size 20
    a) Evaluate each individual using fitness function
    b) Order population by fitness
    c) Select elites, remove rest of population
    d) Select parents for next generation proportional to relative fitnesses
    e) With two parents, mutate and crossover to form two children
    f) Continue until population size is reached

TABLE II: The pseudocode outline of the GA used. The fitness function used in step 2a) is described in TABLE I.

giving each elite individual a chance of being selected as a parent equal to their own fitness divided by the total fitness of all elites. With this, it is more likely that the most fit individual is selected, but information from less fit individuals in the population is not lost between generations. In other words, variability in genomes is maintained.

III. Results

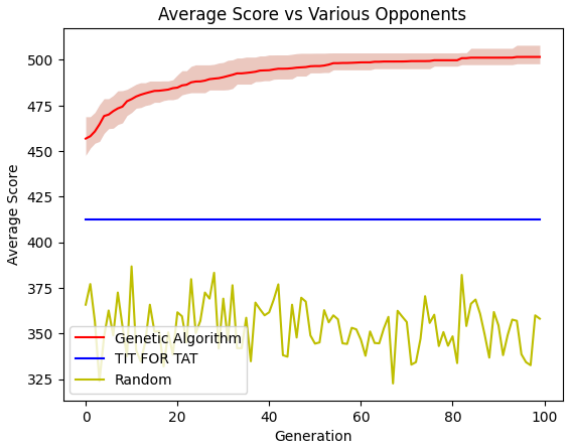

Fig. 3: Results of GA with shaded regions representing 25th and 75th percentiles. TIT FOR TAT and random strategies plotted along same axes for comparison.

Using the procedure described above, we were able to generate data for 40 runs of the GA in just under two minutes. As seen in Figure 3, the random player performed the worst, the TIT FOR TAT player performed moderately, and the GA player performed by far the best. Even the lower 25th percentile of the GA player scored far above the TIT FOR TAT player.

Compared to [8], it was stated that the TIT FOR TAT should score around 428. Our TIT FOR TAT, though operating on the same logic, scores only 412.5 on average. Additionally, [8] states that any score above 450 is a significant improvement to TIT FOR TAT, but even our first generation GA outperforms this score.

IV. Discussion

The results of our experiment, especially when compared to [8], are incredibly interesting because our GA scored significantly better. One difference in terms of the methodology is that [8] uses a different method to select parents for reproduction. This could not have been the cause of such a large discrepancy however since even at generation one there is a considerable gap between the score of our GA and TIT FOR TAT.

Another thought that crossed my mind is that perhaps TIT FOR TAT performs particularly poorly against the pattern players, especially the player that alternates between defecting and cooperation. This would make sense as the GA is given a population size of 20, see TABLE II, so if only a few locuses of the genome are actually used by the pattern players, they will very quickly be able to get the maximum points against these opponents.

To combat this, we changed the weighting of the fitness function. As seen in TABLE I, the pattern players weigh equally into the score with the simple AI player. Making this change did indeed cause the TIT FOR TAT player's average score to increase, but still it falls short of the expected value from [8]. Additionally, after changing the weights the average score of the GA player increased across all generations, essentially counteracting the change.

Another interesting thing to point out is the large discrepancy between the first generation GA score and the random scores. Since the first generation genome is created completely randomly, it would make sense for these to be much more similar. However, the first generation genome represented in Figure 3 is the highest scoring individual out of a population of 20.

For the same reason as discussed earlier, this would make it very likely that the population contains an individual that can maximize its points against the pattern players even in the first generation. Additionally for any given genome of a GA player, the player will always respond the same way to a given previous three inputs. This is not the case for the random player. These two factors combined could have easily caused this large gap between the first generation scores of the random and GA players.

V. Conclusion

The GA completely outperformed TIT FOR TAT, previously thought to be the best strategy for the iterative prisoner's dilemma game [3][8], and demonstrated the GA's affinity for capitalizing on patterns to achieve a higher fitness. In the future, this experiment could be improved by improving the variety and quality of the opponents that make up the fitness function. It would also be interesting to see how the result would change if the GA had access to more previous inputs.

Evolutionary algorithms shine in applications where they can detect some sort of pattern. These sorts of problems are common in fields like mathematics or physics. For example, it is incredibly common for evolutionary algorithms to be used for creating walking gaits [9][10][11][12], which is often a very rhythmic, pattern-induced movement. Though GAs cannot solve every problem, they find great success in problems involving patterns.

REFERENCES

[1] A. M. Turing, *Computing machinery and intelligence*. Springer, 2009.

[2] S. Forrest and M. Mitchell, "Relative building-block fitness and the building-block hypothesis," in *Foundations of genetic algorithms*. Elsevier, 1993, vol. 2, pp. 109–126.

[3] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[4] M. Mitchell, J. H. Holland, and S. Forrest, "The royal road for genetic algorithms: Fitness landscapes and ga performance," Los Alamos National Lab., NM (United States), Tech. Rep., 1991.

[5] J. Lehman, J. Clune, D. Misevic, C. Adami, L. Altenberg, J. Beaulieu, P. J. Bentley, S. Bernard, G. Beslon, D. M. Bryson *et al.*, "The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities," *Artificial life*, vol. 26, no. 2, pp. 274–306, 2020.

[6] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 228–234, 1990.

[7] R. A. Brooks, "Intelligence without representation," *Artificial intelligence*, vol. 47, no. 1-3, pp. 139–159, 1991.

[8] R. Axelrod, "Evolving new strategies," *Genetic Algorithms and Simulated Annealing*, vol. 89, pp. 32–41, 1987.

[9] H. Lipson and J. B. Pollack, "Automatic design and manufacture of robotic lifeforms," *Nature*, vol. 406, no. 6799, pp. 974–978, 2000.

[10] V. Zykov, J. Bongard, and H. Lipson, "Evolving dynamic gaits on a physical robot," in *Proceedings of Genetic and Evolutionary Computation Conference, Late Breaking Paper, GECCO*, vol. 4. Citeseer, 2004, p. 2004.

[11] G. S. Hornby and J. B. Pollack, "Evolving l-systems to generate virtual creatures," *Computers & Graphics*, vol. 25, no. 6, pp. 1041–1048, 2001.

[12] E. J. Earon, T. D. Barfoot, and G. M. D'Eleuterio, "From the sea to the sidewalk: the evolution of hexapod walking gaits by a genetic algorithm," in *International Conference on Evolvable Systems*. Springer, 2000, pp. 51–60.

[13] A. Newell, "Physical symbol systems," *Cognitive science*, vol. 4, no. 2, pp. 135–183, 1980.

[14] A. Thompson, "An evolved circuit, intrinsic in silicon, entwined with physics," in *Evolvable Systems: From Biology to Hardware: First International Conference, ICES96 Tsukuba, Japan, October 7–8, 1996 Proceedings 1*. Springer, 1997, pp. 390–405.

[15] D. Floreano and F. Mondada, "Evolution of homing navigation in a real mobile robot," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 3, pp. 396–407, 1996.

[16] R. A. Watson, S. G. Ficici, and J. B. Pollack, "Embodied evolution: Distributing an evolutionary algorithm in a population of robots," *Robotics and autonomous systems*, vol. 39, no. 1, pp. 1–18, 2002.

[17] D. Cliff, P. Husbands, and I. Harvey, "Evolving visually guided robots," 1993.

[18] P. Funes and J. B. Pollack, "Computer evolution of buildable objects for evolutionary design by computers," *Evolutionary design by computers*, vol. 1999, pp. 387–403, 1999.

[19] J.-B. Mouret and J. Clune, "Illuminating search spaces by mapping elites," *arXiv preprint arXiv:1504.04909*, 2015.

[20] M. Toussaint, "Demonstrating the evolution of complex genetic representations: An evolution of artificial plants," in *Genetic and Evolutionary Computation Conference*. Springer, 2003, pp. 86–97.

[21] C. Igel, N. Hansen, and S. Roth, "Covariance matrix adaptation for multi-objective optimization," *Evolutionary computation*, vol. 15, no. 1, pp. 1–28, 2007.