The Iterated Prisoner's Dilemma: An Evolutionary Approach

Joel Rorseth, Michael Melanson, Michael Bianchi, Kolby Sarson

Dr. Robin Gras

03-60-371 Artificial Intelligence Concepts

TABLE OF CONTENTS

ABSTRACT

In this paper, two well known Artificial Intelligence search strategies are used to experimentally develop intelligent prisoners to participate in the Iterated Prisoner's Dilemma. As a form of preliminary investigation of the established opponents to the proposed intelligent strategies, the study first investigates the performance of known optimal solutions to the Iterated Prisoner's Dilemma. To begin the search algorithm experimentation, a hill climbing optimization algorithm was implemented, helping to define a performance baseline and common ground for comparison between other search strategies. Following this, a genetic algorithm is built, accepting a variety of parameters and tuned for optimal performance. After many tests and thorough comparisons, it is determined that the success of the genetic algorithm player is more often superior to that of a hill climbing player. However, little correlation exists directly between either intelligent strategy and its performance, as the score strongly lends itself to the strategy's parameters, in tandem with the opponent's inherent intelligence. A brief explanation of the findings and optimal move sequences is discussed, in an effort to understand the decisions yielded by both intelligent approaches.

INTRODUCTION

The Prisoner's Dilemma is a famous illustration of a game between two rational agents, which demonstrates how two rational players may not cooperate, despite it appearing to be an optimal decision. The Iterated version of the Prisoner's Dilemma establishes a series of dilemmas in the format of a game, in which the dilemma is played repeatedly by both agents. Thus, both players are aware of the last move of their opponent. The game was formalized by Albert W. Tucker, who established the prison analogy as follows:

Two acquaintances, A and B, have been arrested for committing a crime, and have been imprisoned in two confined rooms, with no communication. However, no evidence is found in order to prosecute the two, thus the prosecutors attempt to offer a bargain to each prisoner to confess. A and B are both given the choice to cooperate with their acquaintance by not saying anything, or to defect by blaming the crime on their acquaintance.

- If A and B both stay silent (both *cooperate*), they are both sentenced to 1 year
- If A and B betray each other (both *defect*), they are both sentenced to 2 years
- If A defects while B cooperates, A will be set free and B will be sentenced to 3 years

Given the sentences associated with each action, a rational or self serving agent would choose to defect, the most likely of the actions to produce a favorable outcome. To formalize the game and its actions for this experiment, we first establish a payoff matrix, associating a (proportionally high) score for lesser sentences. The payoff matrix is used to score the performance of the players, herein referred to as *strategies*. The following matrix was formalized by Robert Axelrod, where C and D stand for *cooperate* and *defect*:

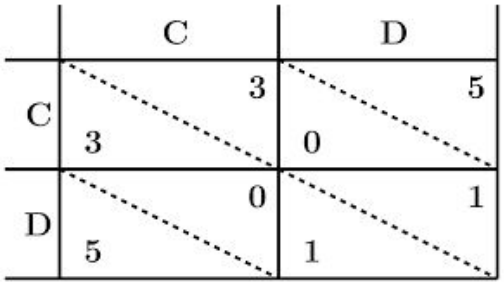

Figure 1 - The payoff matrix first established by R. Axelrod

Goals

The intent of the research conducted was first to compare common known strategies to determine which would be most effective. The three strategies tested and used were Tit-for-Tat (TFT), Tit-for-Two-Tat (TF2T), and Suspicious Tit-for-Tat (STFT). Here after the designing and implementation of artificially intelligent algorithms that could generated an optimal strategy when competing against the common strategies selection would need to be put in place. Lastly, the comparison of the abilities of our artificially intelligent algorithms would need to done in order to produce effective strategies.

Experiment Setup

Many studies have been performed to determine which Iterated Prisoner's Dilemma strategy is the most effective, resulting in many different strategies being developed. The strategy widely considered the best is Tit-for-Tat (TFT). The algorithm for TFT is very simple. It cooperates on the first turn, and mirrors the opponent's moves for the remainder of the game. This allows it to benefit from mutual cooperation while still protecting itself against more untrustworthy strategies. Because of the overall effectiveness of TFT, different variants of the

algorithm have been developed. Two such algorithms are Suspicious Tit-for-Tat (STFT), and Tit-for-Two-Tat (TF2T).

STFT follows the same general pattern as TFT, except instead of cooperating on the first move, it defects. If the opponent's first move is a defection, then STFT usually will outperform TFT. However, there is a significant risk involved with this strategy. When the strategy it is going up against is TFT, one of the variants, or a strategy that behaves very similarly to TFT, this will lead to an infinite loop of defection. It is for this reason that STFT is in general worse than TFT. Cooperating against a defection on the first move is inconsequential compared to being stuck in an infinite defection loop. TF2T varies from the original TFT in that the opposing strategy must defect twice in order for TF2T to return the defection. Similar to STFT, this strategy will outperform TFT if the opposing strategy starts with a defection. However, unlike STFT, the risk of an infinite defection loop against strategies that behave similarly is gone. Because two defections are needed for a defection to be returned by TF2T, the defection loop will be avoided. However, against smarter strategies, requiring two defections in order to return a defection can be taken advantage of, thus lowering the score of TF2T. For this reason, in general TFT is also more effective than TF2T.

In order to acquire a better understanding of their effectiveness, tests using the *Python Axelrod* library were setup to observe how they performed against each other. Due to the similarity of the strategies, many of the match-ups resulted in near ties.

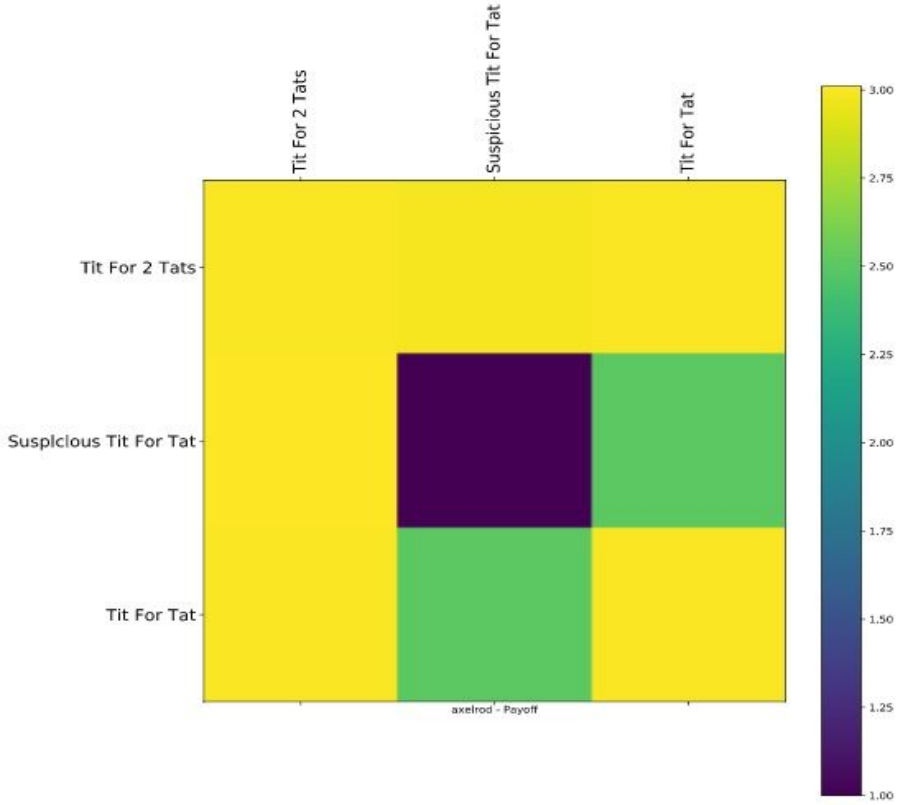

Figure 2 - A colored matrix displaying the score for each strategy when tested against each other

The highest scores obtained were in the matchups between TFT vs TFT, TFT vs TF2T, and TF2T vs TF2T. All of these matchups resulted in a tie. Because TFT and TF2T start with cooperate, in all of these matchups an infinite loop of cooperate is immediately entered, resulting in an equal, but still high, score. Each strategy earns 3 points per turn, resulting in a total of 6 points per turn. TF2T vs STFT resulted in a slightly lower score, with STFT winning the match. STFT earns 5 points off the initial move, but since TF2T requires two defections in order to return a defection, both strategies cooperate for the remainder of the game. Therefore the score is 1 point less than the above mentioned matches, since STFT earns 5 points compared to TF2T's 0 points on the first turn (instead of 3 points each, which is 6 total).

TFT vs STFT resulted in STFT winning, but with a lower score than the matches mentioned thus far. The reason for this is that STFT wins the first turn by defecting against TFT, which cooperates. This results with both strategies flip-flopping their decision in the next turn. This continues for the full game, resulting in 5 points earned per turn between both strategies instead of 6. It is worth noting the winner of this matchup can change, depending on how many rounds are played - an even number will result in a tie, whereas an odd number will give STFT the win. Finally, the last matchup was STFT vs STFT. This resulted in tie, with by far the lowest score. Both strategies are immediately caught in an infinite defection loop, resulting in only 2 points earned per turn between both strategies.

REPRESENTING IPD AS A SEARCH ALGORITHM

To begin the development of intelligent Iterated Prisoner's Dilemma players using optimization and search strategies, the experiment must first define common ground in the form of evaluation of the searches. Moreover, the opponents' strategies and length of the IPD matches must be standardized, in an effort to ensure consistent scoring and fitness evaluation. We first define a *match* as a two player Iterated simulation of the Prisoner's Dilemma. Thus, a match defines two sequences of moves, each representing a distinct player's sequence of turns.

Representing a Game as a Genome

Hereafter, we define a player's participation in the IPD as a *move sequence*, which simply represents the sequence of moves that a player yields for each turn in the match. In a hill climbing algorithm this sequence of moves is the configuration, or state, for which neighbors will be generated and fitness evaluated upon. In the case of a genetic algorithm, move sequences are genomes. We therefore establish that the input and state of both search strategies is *uniform*.

Deterministic Fitness Heuristic for Matches

Due to the fact that both intelligent strategies developed are search / optimization algorithms, and that each defines a *move sequence*, the opportunity is taken to define a common heuristic. This heuristic takes, as input, a single player's *move sequence* (genome) and a given opponent strategy. The opponent’s strategy is specified as a parameter of the simulation, common to both intelligent players. The opponents' (TFT, TF2T and STFT) moves are deterministic, and are thus generated using the player's sequence. Thus, the opponent simply uses its strategy to yield its own *move sequence*. The fitness of the player's score is evaluated using the Axelrod payoff matrix. In general, the heuristic / optimization is designed to maximize score.

HILL CLIMBING

Overview

Being the first attempt in developing an intelligent Iterated Prisoner's Dilemma player, the hill climbing based player implements a simple, local search to greedily mine better scoring sequences of moves. As a hypothesis, it was expected that due to the nature of its greedy search technique, and its likelihood of terminating search upon reaching local maxima, that this intelligent player would yield relatively inferior move sequences. However, tuned with the right parameters, the hill climbing search proves to be a fair player against certain known strategies. The overall structure of the chosen implementation, from a high level, is defined as:

```
Input:  Initial random move sequence
Output:  Move sequence with fitness >= initial move sequence fitness
current_sequence ← initial random move sequence
for i ← 1 to ∞ do
        successors ← Successors ( current_sequence ) ;
        next_sequence ← current_sequence ;
        for  successor in successors  do
                if  Fitness ( successor ) > Fitness ( next_sequence )  then
                        next_sequence ← successor ;
                end
        end
        if  Fitness ( next_sequence ) > Fitness ( current_sequence )  then
                current_sequence ← next_sequence ;
        else
                return current_sequence ;
        end
End
```

Successors Function

Arguably the most important element of the hill climbing search, the Successors function determines the move sequences that are *neighbors* of the current sequence at a given point in the algorithm. As hill climbing does not look ahead of its immediate neighboring states, we define a successor B to a sequence A, as any sequence which is identical to A in every position except for one. Thus, a sequence of length n will yield exactly n successor sequences.

Fitness Function

In an effort to maintain identical parameters for testing other intelligent players, the fitness is standardized across all search algorithms. The fitness of a move sequence is simply the score of the intelligent player when played against the opponent, whom is a predetermined known IPD strategy. Thus, larger player scores yield proportionally higher fitness. The moves that the known strategy will yield in response to the move sequence produced by the search algorithm is deterministic, and is therefore well defined for each of TFT, TF2T and STFT.

GENETIC ALGORITHM

Overview

Hypothesized as being the most promising of the search methods, the genetic algorithm is written to be an elitist and highly parametric implementation. After much testing, it was determined that the following general structure proved to be the highest scoring:

Input: Number of generations *g*, population size *p*, number of survivors *s*
Output: An evolved move sequence, the most fit of the final generation
population ← set of randomly generated move sequences
for *i* ← 1 to *g* **do**
    next_generation ← Survivors (*population*, *s*) ;
    for j ← 1 to (*p* - *s*) **do**
        first_parent ← Successor (*population*) ;
        second_parent ← Successor (*population*) ;
        child ← Crossover (*first_parent, second_parent*) ;
        child ← Mutate (*child*) ;
        next_generation .insert (*child*) ;
    end
    population ← *next_generation* ;
end
return *population* ;

Population

In this implementation, the algorithm begins by generating a random population. This population contains a number of move sequences, all of a fixed length. For each sequence to be

generated, a single bit C (cooperate) or D (defect) has an equal chance of being randomly selected to be placed at a given position in the sequence.

Fitness Function

As previously discussed, the fitness of both intelligent players has been standardized, with fitness being equal to score achieved by a given sequence. The score is determined using the Axelrod scoring matrix, summing the score of each turn in the sequence to determine the total score for the sequence.

Selection

There are several common approaches to the selection process for genetic algorithms. For the experiment, the *Roulette* and *Elitist* selection strategies were both tested. In the initial genetic algorithm implementation, the former was used. The issue we discovered with roulette selection was that there was possibility for future generations to decline in fitness. This was a result of less fit parents having too great a chance of creating children. To verify and combat this setback, the algorithm was modified to perform elitist selection. With this change, a fixed number of the population's fittest move sequences are allowed to persist to the next generation. The average score of the genetic algorithm improved by a large factor after this decision was made, due to the fact that children may very easily score much worse than their parents.

Roulette

In roulette selection, each sequence is given a weighting based on its personal fitness proportionate to the total fitness of all sequences in a population. Next, two parents are randomly selected from the population using these weightings and are used in the child creation.

Elitist

In elitist selection, a given number of sequences, *n*, which are the most fit or elite, are selected from the population and directly added to the following generation. These elite sequences are *not subject* to mutation before joining the next generation. Following this transfer, the remaining spots in the (fixed) population size are filled using the aforementioned Roulette technique, with parents being selected (with weighting) from the *current population*.

Crossover

Our crossover function implements a variation of a single-point crossover. In this method of crossover, a single random index *p* is selected. In common implementations, two children are produced by merging A's sequence before *p* with B's sequence after *p*, then yielding an additional child using the opposite partitions of each. A and B. The implementation decided upon for this implementation produced a single child, by taking only one sequence produced by the split about *p*. Implementing this variation of a single-point crossover allows for more sampling

(for parents) from the population, since the selection process will occur *twice* as many times. This is due to only one child being produced for each two parents, rather than two children.

Mutation

The mutation method we chose to employ is a variation of bit string mutation. In bit string mutation, bits in the sequence are flipped at random locations, with the probability for each bit flipping being 1/l, where l is the length of the sequence. Our variation of this mutation strategy uses a constant mutation probability as a parameter to the genetic algorithm, rather than a probability based on sequence length. This allows for the mutation rate to be changed while keeping the sequence length constant.

General Performance

Overall, our genetic algorithm performed very well against TFT, TF2T and STFT. Of all tested opponent strategies, only one in particular proved difficult to develop a strategy against. This was the case when the genetic algorithm played against STFT, specifically with low mutation probabilities (in the range of 0 to 20%). It was determined that the genetic algorithm consistently got stuck in a loop of defection against STFT, yielding increasingly lower scores. When the mutation probability was increased, this pattern could be broken by more often randomly flipping defects to cooperates, moving the solutions towards the ideal sequence of all cooperates followed by a final defect.

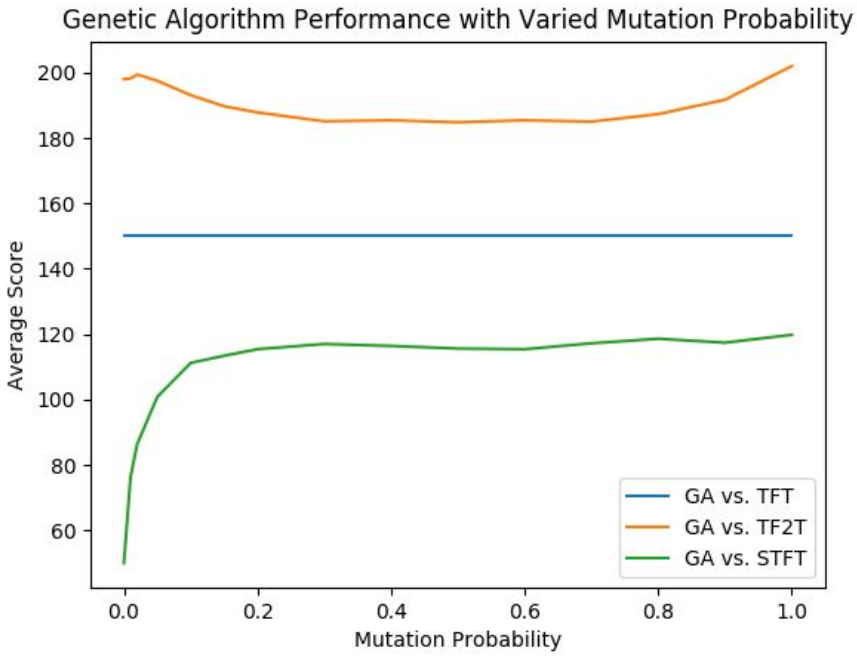

Figure 3.1 - 100 generations, 100 population size, 50 turns in each game, 0.25 survival rate

In Figure 3.1, we see that when competing against TFT or TF2T, the mutation rate has little effect on the average score of the genetic algorithm. However, when pitted against the STFT strategy, its performance displays a visible *positive correlation* with the *mutation rate*.

In comparison, Figure 3.2 shows the effect that modifying the number of generations has. In general, more generations yields more time for the population to evolve potentially better solutions. Against TFT, the average score is largely unaffected by increasing the number of generation, as the score quickly plateaus after introducing only a few more generations. However, against TF2T, the average score slowly but steadily increases as more generations are added. When up against STFT, the number of generations appears to have no effect on the average score, as the results are inconsistent for varying numbers of generations.

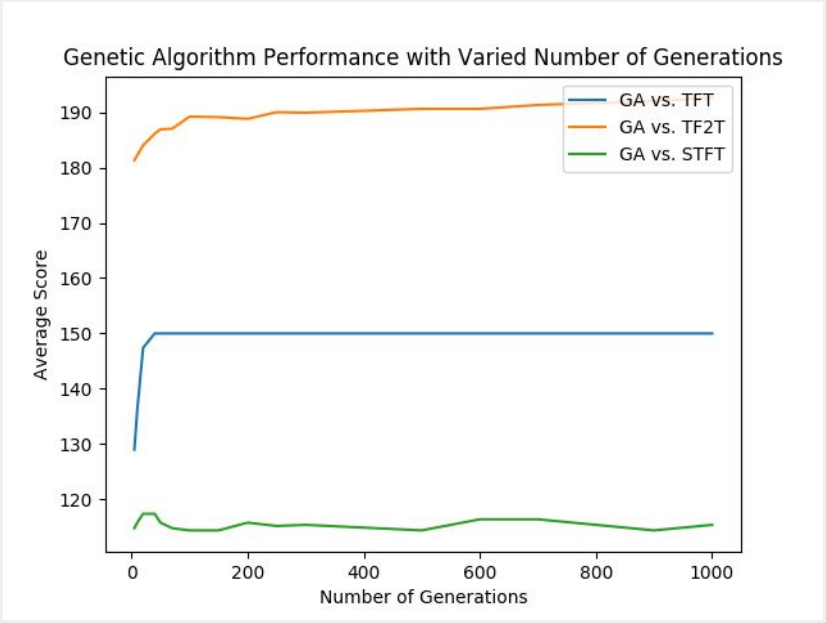

Figure 3.2 - 100 population size, 50 turns in each game, 0.25 survival rate, 0.2 mutation probability

Similarly, Figure 3.3 illustrates how the algorithm generally benefited in performance with increasing population sizes. Undoubtedly, the performance of the genetic algorithm

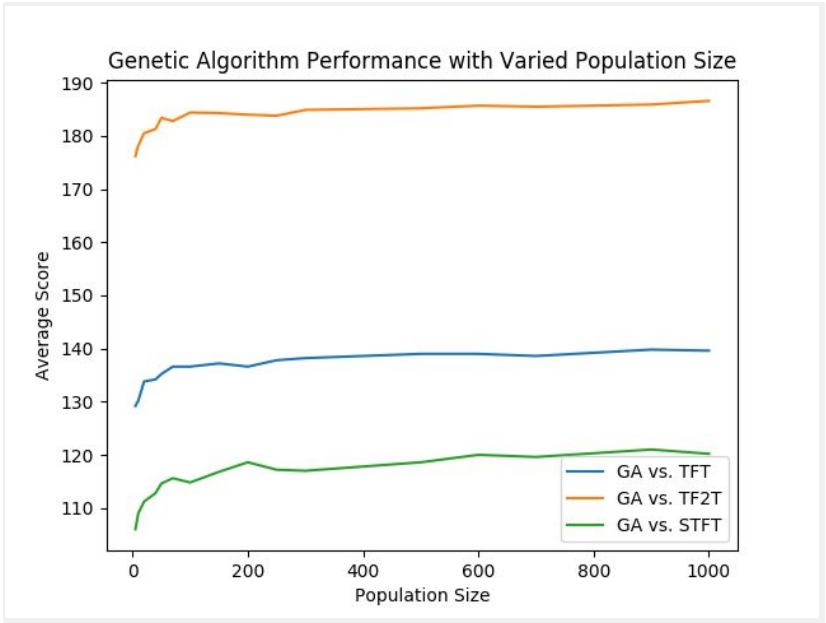

Figure 3.3 - 10 generations, 50 turns in each game, 0.25 survival rate, 0.2 mutation probability

implementation is *positively correlated* with the population size. In particular, across other tests, it becomes apparent that performance against the TF2T and STFT benefits most by this change. This is due to the fact that larger populations will contain more random move sequences, meaning that the probability of randomly generating more fit sequences from the start, is proportionally larger.

Varying the survival rate has an interesting effect on the genetic algorithm, as it has a different effect against each of the strategies. Against TFT, a lower survival rate will result in a sharp increase in average score before leveling off, very similar to how varying the number of generations affects the average score. Against TF2T, a very low survival rate will result in a

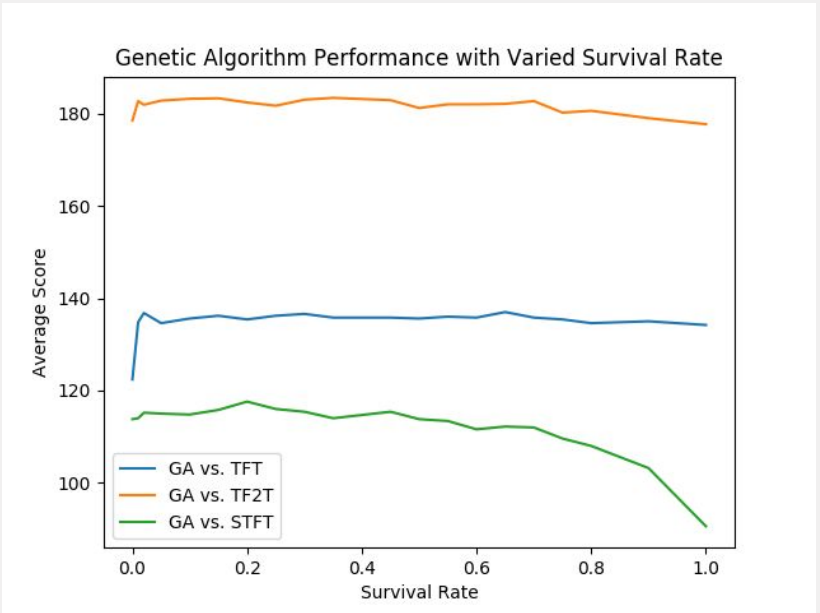

Figure 3.4 - 10 generations, 100 population size, 50 turns in each game, 0.2 mutation probability

slightly higher average score. As the survival rate increases, the average score is not affected very much, until around survival rates of 0.8 and higher, where the average score actually starts to decrease gradually. Against STFT, the average score is barely affected by the survival rate until the survival rate hits 0.5 and higher, where like TF2T it starts to decrease. As the survival rate approaches 1.0, less children are being produced. Thus, a genetic algorithm with 100% survival rate will essentially produce a single random population, which we expect to be poor in score. This is apparent in Figure 3.4, with the true benefit of the Elitist selection being visible for certain survival rates only.

CONCLUSION

In contrast, the genetic algorithm performed when playing against TFT and TF2T. When opposing TFT, over time the genome would evolve to favour mutual cooperation over defection. This is consistent with what was expected, as the max score against TFT is obtained through mutual cooperation with a defection as the last move. For TF2T, the max score is obtained by defecting every other move. Again, the genetic algorithm was consistent with expected results, and generated a higher score than when going against TFT. However, the genetic algorithm was typically unsuccessful when going against STFT. Because STFT defects on the first move, the algorithm tries to compensate for this by keeping genomes that also defect on the first move. This eventually leads to genomes of all defect, producing a score far below the max.

The hill climbing algorithm exceeded expectations when running against TFT, actually outperforming the genetic algorithm. The reason for this is the way the hill climber chose successors. Because there are very few options for a higher or lower score, among all of the successor nodes the hill climber can choose from there will be ties. The hill climber always chooses the first node with the highest increase, as opposed to randomly selecting one. As a result, when running against TFT, a strategy that is harder to take advantage of, hill climber performed better than the genetic algorithm. But when going up against TF2T, a strategy that is easier to take advantage of, hill climber had a hard time adjusting, and wound up with a very similar score to when it went against TFT, ultimately being outperformed by the genetic algorithm.

Overall, the genetic algorithm seems to perform better in most cases, as it explores more of the search space and is not prone to local maxima. However, hill climber still performs relatively well, occasionally beating the performance of genetic algorithms with certain parameters.

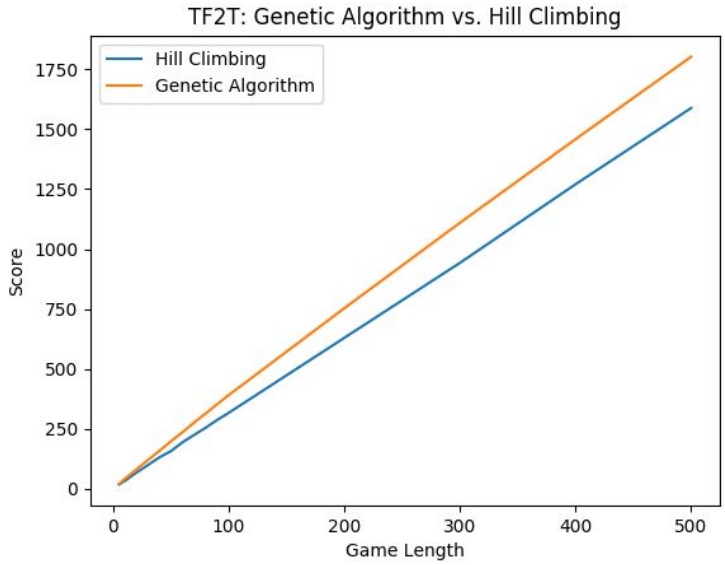

Figure 4.1 - Intelligent strategies face off against TF2T. Here the genetic algorithm is set to use 100 generations, 100 population size, 50 turns in each game, 0.2 mutation prob

In specific, the TFT opponent often yielded quick results with the genetic algorithm, essentially terminating the search early. In this sense, the hill climbing algorithm has a fair probability of outperforming the genetic algorithm for this opponent, as the genetic algorithm will *not* have enough time to develop a population and search the solution space with *larger game sizes*. This effect is demonstrated in Figure 4.2, where with larger game lengths, the successor generation of the hill climbing algorithm is more effective in larger game lengths.

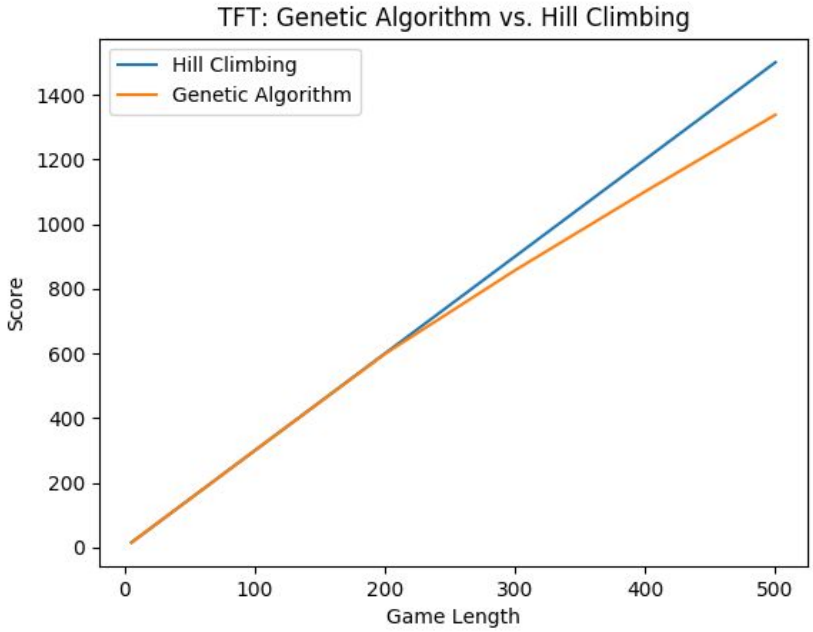

Figure 4.2 - Intelligent strategies face off against TFT. Here the genetic algorithm is set to also use 100 generations, 100 population size, 50 turns in each game, 0.2 mutation prob

WORKS CITED

Axelrod, Robert. “More Effective Choice in the Prisoner's Dilemma.” The Journal of Conflict Resolution, vol. 24, no. 3, 1980, pp. 379–403. JSTOR, JSTOR, www.jstor.org/stable/173638.

Russel, Stuart. Norvig, Peter. “Artificial Intelligence: A Modern Approach.” Pearson, 2018.