Evolving Strategies for the Repeated Prisoner's Dilemma Game with Genetic Programming: Studying the Effect of Varying Function Sets

By Daniel J. Scali
2006 Undergraduate Honors Thesis
Advised by Professor Sergio Alvarez
Computer Science Department, Boston College

Abstract: This thesis examines the application of genetic programming to evolving strategies for playing an iterated version of the Prisoner's Dilemma game. The study examines the evolution of strategies for a single population of players pitted against a static environment, as well as the co-evolution of strategies for two distinct subpopulations of players competing against one another. The results indicate that the strategies that can be evolved are strongly influenced by the function set provided during the design process. In co-evolutionary runs in particular, the function set shapes the environment of opponents that an individual strategy is evaluated against. Experimental runs that alter the makeup of the function set facilitate a discussion of how different function sets and environments can lead to diverse strategies with varying levels of performance.
Keywords: genetic programming, prisoner's dilemma, game theory, function set

1. Introduction

The Prisoner's Dilemma has previously been used to show the validity and effectiveness of applying genetic algorithms in a game-theoretic context. [2,5]. The current body of research consists of various methods for representing strategies for the Repeated Prisoner's Dilemma. In each case, the representation scheme that is chosen provides a framework for the evolutionary process and dictates the number of previous moves that a given strategy can consider in the calculation of its next move. For example, Miller [12] modeled strategies as bit-string representations of finite state automata whereas Axelrod's [2] more direct approach used bit strings to reflect the last three moves of the game's results history. Some of these studies evolve strategies based on their performance against a fixed environment [6,12] while others have introduced the idea of co-evolution – the process of assessing a given strategy by its performance against its peers in each evolutionary generation [2,5,9].

Although Fujiki and Dickinson [5] demonstrated that genetic algorithms could be used to evolve Lisp S-expressions, the solution relied heavily upon the construction of a grammar and was contingent upon "a proper set of productions" being used. The goal of the current paper is twofold. First, it aims to take a more structured and extensible approach to evolving strategies for the Repeated Prisoner's Dilemma game by applying the genetic programming paradigm as developed in Koza 1992 [8]. In addition, it attempts to determine the implications of altering the function set that the algorithm draws from when composing new strategies.

2. The Prisoner's Dilemma

The *Prisoner's Dilemma* game has been shown to have a variety of applications in the social sciences and other fields, ranging from trade tariff reduction, to labor arbitration, evolutionary biology, and price matching [1,4].

The Prisoner's Dilemma game is best illustrated anecdotally: Suppose that you are a bank robber. One day, you and your accomplice are both brought to the police station and placed in separate rooms for questioning. Isolated from each other, you are each explained the following: If you both confess, you will receive matching 5 year sentences. On the other hand, without a confession from either of you, the police only have enough evidence to put you both away for a lesser crime which carries a penalty of only 3 years. However, if one robber confesses and the other does not, the recalcitrant party will be sentenced to 10 years in prison while the robber making the confession will receive only 1 year. What should you do?

		Robber 2	
		Deny (Cooperate)	*Confess (Defect)*
Robber 1	*Deny (Cooperate)*	3 , 3	10 , 1
	Confess (Defect)	1 , 10	5 , 5

Table 2.1 *The Prisoner's Dilemma with payoffs as time in jail*

Economic game theory provides tools for analyzing this situation. Table 2.1 models the above scenario as a strategic game, where Robber 1's jail sentence is always listed first. The strategies available for each prisoner boil down to two options: either confess to the crime (Defect from accomplice) or deny the allegations (Cooperate with accomplice).

Each player in the game has one objective: to minimize his time in jail. Robber 1 has no knowledge of what Robber 2's move will be. However, Robber 1 knows that if Robber 2 confesses, his best response is to confess – he receives only 5 years in jail if he confesses as opposed to 10 years if he denies the allegations. He also knows that if Robber 2 chooses to deny, his best response is to confess – he receives only 1 year in jail if he confesses as opposed to 3 years if he denies. No matter what Robber 2 does, it is always in Robber 1's best interests to confess. Thus, confession is a dominant strategy for Robber 1 [4]. Since, a similar analysis holds true for Robber 2, the dominant strategy equilibrium is for both robbers to confess. Curiously, even though `[Confess, Confess]` is a dominant strategy equilibrium, both parties would be better off if the outcome was `[Deny, Deny]`.

		Player 2	
		Cooperate	*Defect*
Player 1	*Cooperate*	R = 3 , R = 3	S = 0, T = 5
	Defect	T = 5 , S = 0	P = 1 , P =1

Table 2.2 *The Prisoner's Dilemma, with payoffs as points*

The Prisoner’s Dilemma can also be described formally. Table 2.2 is a strategic form illustration of another classic example of the Prisoner’s Dilemma game from Axelrod [1]. Note that in this example (and from this point forward), players are trying to maximize their payoff rather than minimize it as in the previous formulation.

Another useful way of viewing the Prisoner’s Dilemma game is as tree, in what is called extensive form. Fig 2.1 is an extensive form representation of the strategic game in Table 2.2. *Information Sets*, shown as dotted lines between two or more nodes of equal depth, indicate the fact that the moves occur simultaneously. Here, because Player 2 finds himself in a two-node information set, he has no knowledge of whether Player 1 has moved C or D. The extensive form game in Figure 2.1 has four possible outcomes which are identical to the outcomes in Table 2.2.

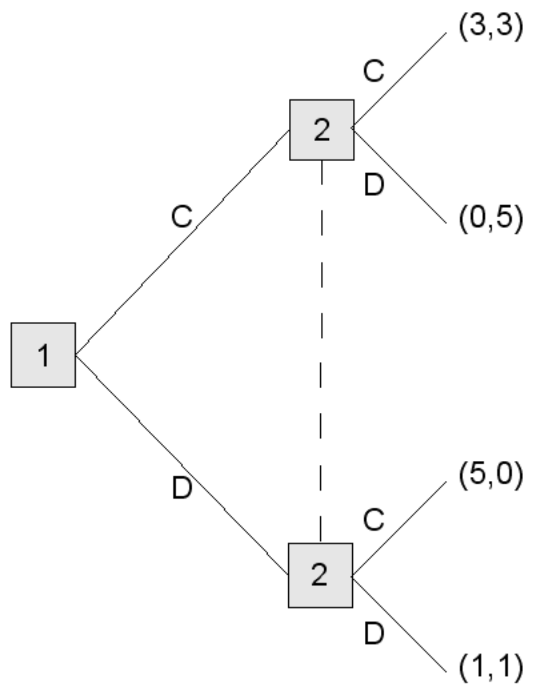


Figure 2.1 *The Prisoner’s Dilemma as an extensive form game*

The game consists of four possible payoffs, which shall be abbreviated *R*, *T*, *S* and *P*. *R* is the reward for mutual cooperation, *T* is the temptation to defect, *S* is the sucker's payoff, and *P* is the punishment for mutual defection. In order for the Prisoner's Dilemma to be present, two relationships must apply. First, it must be true that $T > R > P > S$. This ordering preserves the proper incentive structure, as the temptation to defect must be greater than the reward for cooperation, and so on. Second, the reward for cooperation must be greater than the average of the temptation to defect and the sucker's payoff – i.e. $R > .5(T+S)$. This removes the ability of players to take turns exploiting each other to do better than if they played the game egoistically [1].

A further technical analysis shows how the `[Defect, Defect]` dominant strategy equilibrium is undesirable, but not easily avoidable. `[Defect, Defect]` is not considered an efficient outcome. An outcome in a game is considered *Pareto efficient* if no other outcome exists that makes every player at least as well off and at least one player strictly better off. Therefore, mutual cooperation is the only Pareto efficient solution in the Prisoner's dilemma [6]. However even though mutual cooperation is Pareto efficient, the `[Defect, Defect]` outcome is not easily avoidable because it is a Nash equilibrium. A *Nash equilibrium* is an outcome of a game where neither player can unilaterally change his move in order to improve his own payout [4]. It follows that `[Defect, Defect]` is a Nash equilibrium since a player's best choice is to defect when he knows that his opponent will defect. Since `[Defect, Defect]` is the only Nash equilibrium, it is impossible for two rational egoists who play the game only once to end up in any other state.

Although the possibility of mutual cooperation emerging in such a situation seems bleak, cooperation can in fact be sustained when the game is played multiple times. The resulting game is called the *Repeated Prisoner's Dilemma* (RPD). Research shows that the best strategy for playing the repeated game is much different from the best strategy for playing just one round [1]. Repeating the game introduces the possibility for reciprocity that exists in many real-life interactions– players are now able to reward and punish each other based on past interactions. For example, in a repeated setting a rational player might try to achieve the gains from cooperation, but would be able to retreat to defection if an opponent was uncooperative.

The success of the `Tit-for-Tat` strategy in the Repeated Prisoner's Dilemma has been well-documented. When using the `Tit-for-Tat` strategy, a player cooperates on the first move and then mimics the opponent's move from the last round for the remainder of the game. `Tit-for-Tat` scores well in tournaments because of its ability to reward its opponent for cooperation while also punishing it for defection [1]. Other strategies that have scored well are `Tit-for-Two-Tats`, `Grim Trigger`, and `Pavlov` [5,6]. `Tit-for-Two-Tats` is a variation of the `Tit-for-Tat` strategy which cooperates unless defected on for two consecutive moves. The `Grim Trigger` strategy simulates a player who is provocable and unforgiving: it cooperates until defected on, at which time it permanently defects. Last, the `Pavlov` strategy is an intuitive "win-stay, lose-switch" strategy. If it receives a desirable payoff, it repeats its move in the next round. However, if it receives an undesirable payoff, it will try a different move in the next round. Hence, `Pavlov` cooperates if the last move reflects a mutual cooperation or defection (i.e. `[Cooperate, Cooperate]` or `[Defect, Defect]`) and defects otherwise. The `Pavlov` strategy can either defect or cooperate on the first move[1].

3. Genetic Programming

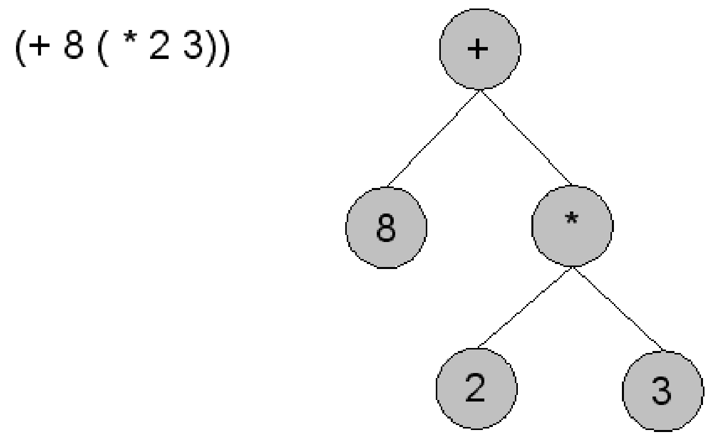
Genetic programming (GP) is an evolutionary computation technique which is based upon the genetic algorithm (GA) developed by Holland [7]. The genetic programming paradigm, as it will be used in the scope of this paper, was popularized by John Koza [8].

Genetic algorithms take advantage of the Darwinian concept of natural selection to evolve suitable solutions to complex problems. In the typical genetic algorithm, a possible solution to a problem – called an individual – is represented as a bit string. Each individual is assigned a fitness, which is simply a determination of how effective a given individual is at solving the problem. For example, when evolving a game-playing strategy, a fitness measure might be the number of points the individual scored in the game. To kickoff the evolutionary process, an initial population of individuals is generated through a random process as detailed by Koza [8]. Next, the fitness of each individual in the population is evaluated. Favoring individuals with the highest fitness, new generations are repeatedly created via genetic techniques such as mutation, crossover, and reproduction. During the process, progressively more fit generations are created until either a satisfactory best-of-generation individual is produced or a pre-determined number of generations have been created.

The difference between the genetic algorithm and genetic programming lies primarily in representation. Genetic programming is distinct because it represents

[1] The `Pavlov` implementation in this paper cooperated on the first move.

solutions as actual computer programs. Programming languages such as Lisp are well-suited for GP because they are structured as symbolic expressions (S-expressions) which can be represented as trees. The nodes of the trees are members of a set of functions and a set of terminals which are defined by the user in the GP representation. Some examples of functions given in Koza [8] are arithmetic operations (+,-,*, etc), mathematical functions (sin, cos, log, etc.), Boolean operations (AND, OR, NOT, etc.), conditional operators (IF-THEN-ELSE), as well as any other domain-specific functions that can be defined. The terminal set typically consists of constant values or state variables. In order to ensure that the trees generated by GP are valid, the function set and terminal set must both be closed. The closure property, stated more formally, ensures that any function in the function set must be able to accept as an argument any value that could be derived from another function in the function set or any terminal in the terminal set. Figure 3.1 shows an example of a simple mathematical Lisp S-expression and a corresponding tree.



```
(+ 8 ( * 2 3))
```

Figure 3.1 *A simple Lisp expression and tree*

There are three operations that are typically carried out during the evolutionary process: reproduction, crossover, and mutation. These primary operations are intended to simulate the process of Darwinian evolution and natural selection thus building stronger populations from generation to generation. In reproduction, an individual is selected from the population and then simply copied into the new generation. The crossover operation, illustrated in Figure 3.2, selects two different individuals from the population, randomly selects one node from each to be the crossover point, and then swaps the subtrees found at the crossover nodes to create two new individuals for the new generation. Lastly, mutation introduces random changes into the new generation. When mutation (Figure 3.3) is applied, an individual is selected and a mutation point is selected at random. A randomly generated subtree is inserted at the mutation point and the mutated individual is added to the new population.

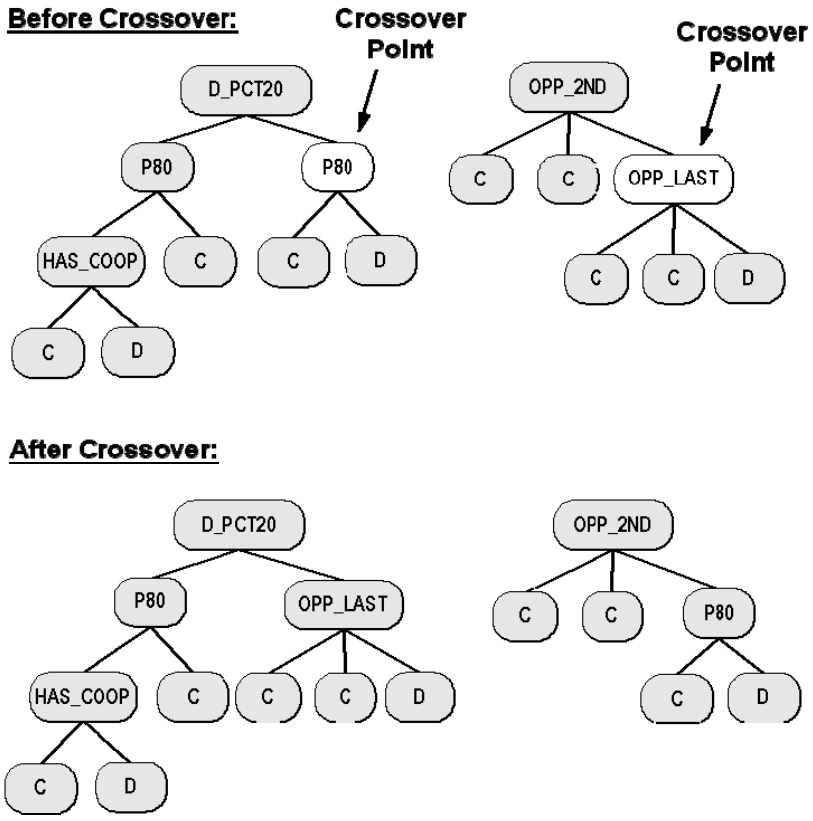


Figure 3.2 *An example of the crossover operation*

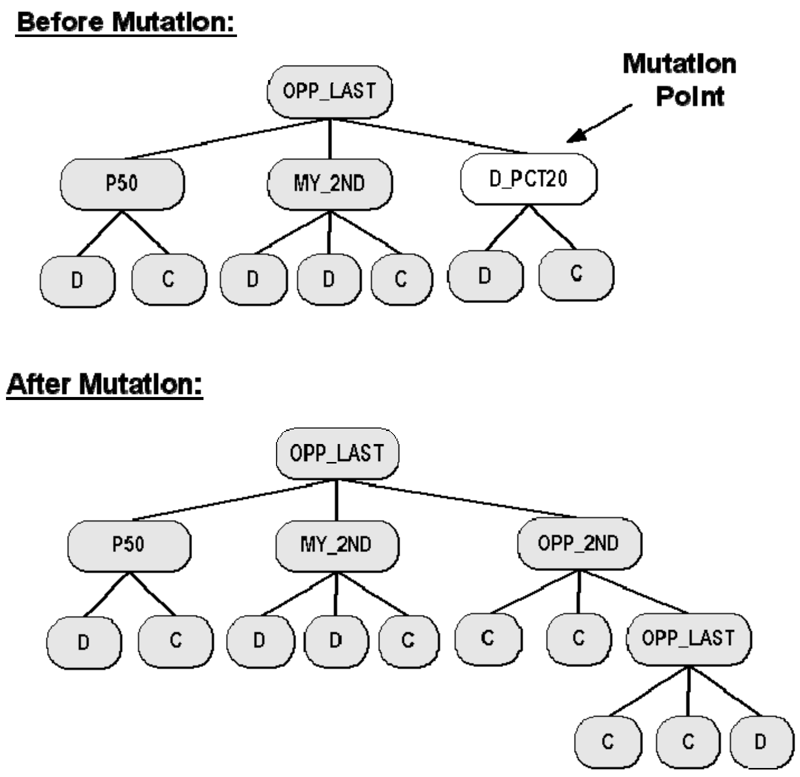


Figure 3.3 *An example of the mutation operation*

The method used to select individuals for the genetic operations can vary, but is typically based on fitness. Two popular selection methods are fitness-proportionate selection and tournament selection. For *fitness-proportionate selection*, the probability that a given individual is selected increases in proportion to an individual's fitness with respect to the rest of the population. In *tournament selection*, a certain number of individuals are selected at random from the group and then the one with the best fitness is selected [8].

There are also some variations of tournament selection that examine tree size as a secondary objective behind fitness. During the evolutionary process, individuals' tree sizes can become unnecessarily and unmanageably large – a phenomenon known as GP bloat. Applying lexicographic parsimony pressure helps to solve this problem. In *lexicographic parsimony pressure*, *n* individuals are chosen at random. As in regular tournament selection, the individual with the best fitness is selected. However, in the event of tie, lexicographic parsimony pressure selects the individual with a smaller tree size. This multiobjective technique helps limit the size of the evolved trees and therefore reigns in bloat [10].

Selection methods can be supplemented by a technique known as *elitism*. The reader may have observed that the selection methods described above are probabilistic in nature – although they will select relatively more fit individuals, they do not necessarily ensure that the most fit individuals pass on to the next generation. If an individual is not selected at random for consideration by the selection method, then it can not pass on to the next generation. Adding elitism to the evolutionary process guarantees that the top *n* individuals (the "elites") move on to the next generation. After positions for the elites are secured in the next generation, the remainder of the population is selected via a standard selection method.

4. ECJ

The genetic programming implementation used for this paper is a Java-based evolutionary computation toolkit called ECJ [11]. ECJ provides a set of classes which can be used to customize a Genetic programming problem domain such as the Repeated Prisoner's Dilemma. The configurable parameter files used by the system allow for a considerable amount of tweaking by the user. The methods for initial population generation, selection, fitness, and genetic operators can all be specified at run time. In addition, ECJ supports co-evolution, which is particularly useful for the evolving game-playing strategies. Co-evolution allows an individual's fitness to be calculated based on its performance playing a game against another evolving subpopulation.

5. Methodology

The Problem

In this version of the Repeated Prisoner's Dilemma, the payoff structure will be defined as [R, T, S, P] = [3, 5, 0, 1] as in Figure 1.2. Each individual plays the Prisoner's Dilemma game 100 times against the *n* opponents in its environment. For static environments, this means playing any number of pre-determined, well-known strategies. In co-evolutionary environments, this means playing 100 randomly selected opponents from the opposing subpopulation. The objective of each GP run will be to evolve the highest-scoring strategy possible.

The Function and Terminal Sets

Applying genetic programming to the Repeated Prisoner's Dilemma requires a conscious determination of how to represent the problem. Although the function set and the terminal set that are created do not directly guide the evolutionary process, the union of these two sets provides the "tools" with which the evolutionary process is allowed to work.

Previous work on evolving a strategy for a simple, 32-outcome extensive form game proves extensible to the Repeated Prisoner's Dilemma [8]. The 32-outcome game consists of only five total moves. Therefore, the function set can provide information about the game's entire history through only four functions which return the value of moves one through four. The Repeated Prisoner's Dilemma is different because of its infinite nature. To understand this, picture what Figure 2.1 would look like if the game was played more than once – the four terminal nodes would be replaced by four Prisoner's Dilemma games in extensive form, which would recursively contain more Prisoner's Dilemma games, and so on. Because the Repeated Prisoner's Dilemma can consist of any number of iterations, it would be impossible to keep the size of both the function and terminal sets bounded while also taking into account the history of the whole game. Therefore, when using this methodology to model the Repeated Prisoner's Dilemma, the number of historical moves taken into consideration must be finite.

The terminal set in this problem consists of the two moves available to the players. In the case of the Prisoner's Dilemma this means Cooperate (C) and Defect (D). In addition, it is important to provide for a move being Undefined (U) to ensure that closure is established. This is required because the functions will not necessarily be able to provide data during the first four moves. For example, if in the first round of a game, a call is made to a function that checks the opponent's second-to-last move, there is no second-to-last move to report for either player, so the function should return U. Therefore, the terminal set T = {C, D, U}.

All of the functions in the function set are based on the CASE statement in Lisp. The first category of functions includes functions that are based on recent move history. These functions can evaluate to any of three arguments in order to provide information about the most recent four moves of both players. To illustrate with an example, the function `OPP_LAST` will evaluate to its first argument if the opposing player's last move is undefined, to its second argument if the move was `C`, and to its third argument if the move was `D`. The functions `MY_LAST`, `MY_2ND`, `MY_3RD`, and `MY_4TH` consult the move history variables to provide information about the individuals own moves (from most recent to least recent, respectively) while `OPP_LAST`, `OPP_2ND`, `OPP_3RD`, and `OPP_4TH` provide the same information for the opponent's moves.

The next category included the functions `HAS_COOP` and `HAS_DEFECTED`. These functions check to see if the opponent has ever, within the current history, cooperated or defected, respectively. Both functions are of arity two in order to handle the "true" and "false" cases. For example, if the opponent has ever cooperated (according to the current move history) `HAS_COOP` will return its first argument, but will return its second argument in all other cases.

Fitness

Only rudimentary fitness calculations are required for the repeated Prisoner's Dilemma. An individual's raw fitness is the sum of its scores from the *n* 100-round contests that it plays against its *n* opponents. Programmatically, it is useful to standardize fitness. A standardized fitness measure, shown in Figure 5.1, is constructed by subtracting the raw fitness from the total number of points available. In simple terms, standardized fitness denotes the number of points that a particular strategy *failed* to earn over the course of its interactions. Therefore, the lower a strategy's standardized fitness is, the better the strategy is. Although a perfect standardized fitness of 0 can only be achieved in an environment filled with opponents who unconditionally cooperate, the standardized fitness measure provides context when comparing two strategies or graphically analyzing data.

```
Standardized Fitness = T * 100 * n - s
```

where: T = the temptation to defect = 5
n = the number of opponents
s = the total number of points scored = raw fitness

Figure 5.1 *An explanation of Standardized fitness*

Evolutionary Operators and Parameters

Unless otherwise noted, runs performed during the study used the following evolutionary operators and parameters: The evolutionary operators were crossover and reproduction, used at the probability of .7 and .3 respectively. Tournament selection with lexicographic parsimony pressure and a tournament size of 7 was used to select the individuals to be operated upon.

In runs against a static environment, a population of 500 individuals was evolved for 100 generations. For co-evolutionary environments, both subpopulations consisted of 500 individuals and were run for up to 500 generations.

The Experiments

Environment Name	**Opponents in Environment**
Environment 1	`All C`
Environment 2	`All D`
Environment 3	`All C, All D, Tit-for-Tat`
Environment 4	`All D, Grim Trigger`
Environment 5	`Tit-for-Tat, Grim Trigger`
Environment 6	`Tit-for-Tat, Grim Trigger, Pavlov`
Environment 7	`All C, Grim Trigger`
Environment 8	`DBTFT, DPC, All D, DTFT, CBTFT, All C, Grim Trigger, Tit-for-Tat`

Table 5.1 *Summary of the environments of pre-defined strategies used in genetic programming runs*

Procedures were designed to examine the effect of the function set's composition on the fitness of the evolved strategies. A standard evolutionary process was used to evolve strategies against different fixed environments and one co-evolutionary environment. In a fixed environment, any number of predetermined strategies made up the environment of opponents. These environments are summarized in Table 5.1 and Java source code for the strategies can be found in Appendix A. In the co-evolutionary environment, the individual being evaluated played against a selection of its counterparts in the current evolutionary generation. Figure 5.2 shows a high level view of the experimental procedure. In Step 1, the function set was sequentially set equal to three predefined function sets: FS1, FS2, and FS3. The function set FS1 consisted of eight functions (`MY_LAST`, `MY_2ND`, `MY_3RD`, `MY_4TH`, `OPP_LAST`, `OPP_2ND`, `OPP_3RD`, and `OPP_4TH`) that could check the last four moves of each player in the history. After this, FS2 was constructed by adding the `HAS_COOP` and `HAS_DEFECTED` functions to FS1. The evolutionary process was again set in motion in all environments. Finally, the function set was reduced to FS3, a function set consisting of just two functions: `MY_LAST` and `OPP_LAST`.

```
Procedure:
   1. Set F = FS1 = {MY_LAST, MY_2ND, MY_3RD, MY_4TH, OPP_LAST,
      OPP_2ND, OPP_3RD, OPP_4TH}
   2. Run against all environments
   3. Set F = FS2 = {MY_LAST, MY_2ND, MY_3RD, MY_4TH, OPP_LAST,
      OPP_2ND, OPP_3RD, OPP_4TH, HAS_COOP, HAS_DEFECTED }
   4. Run against all environments
   5. Set F = FS3 = {MY_LAST, OPP_LAST }
   6. Run against all environments
```

Figure 5.2 *The experimental procedure and various function sets*

6. Results and Discussion

Static Environments

A summary of fixed (i.e. static) environment results can be found in Table 6.1. One of the most interesting static environments was Environment 3, which consisted of the opponents `Tit-for-Tat`, `All D`, and `All C`. The strategy evolved with FS1 and a description of how it performed against the opponents in this environment is shown in Figure 6.1.

Evolved Strategy:

`(OPP_LAST D (MY_LAST D C D) (OPP_2ND D C (MY_LAST C C D)))`

Evolved Strategy:	**DD**DDDDDD…
`All D`:	**DD**DDDDDD…
Evolved Strategy:	**DD**DDDDDD…
`All C`:	**CC**CCCCCC…
Evolved Strategy:	**DD**CCCCCC…
`Tit-for-Tat`:	**CD**DCCCCC…

Figure 6.1 *A strategy evolved in Environment 3 with Function Set 1*

The success of the evolved strategy was predicated on its ability to recognize the three different opponents in the environment and react accordingly. By opening each 100-game interaction with two defections, the GP-evolved strategy was able to elicit a unique response from each of its three opponents. After that, it implemented the optimal strategy against all three: it exploited `All C` by defecting, protected against `All D` by defecting, and cooperated with the less-exploitable `Tit-for-Tat` for the remainder of the interaction.

Further investigation reveals that the opening sequence of `DD` was not trivial. Consider the alternatives: If the strategy were to open with `CC`, it would not be able to tell the difference between `Tit-for-Tat` and `All C`, hence forfeiting the opportunity to exploit `All C` for two extra points per game. Likewise, an opening sequence of `CD` would have elicited identical responses from `Tit-for-Tat` and `All C`. The last possibility is `DC`, which removes the ambiguity between opponents but scores fewer points than `DD` – it fails to exploit `All C` and gets exploited by `All D` on the second move. The diagnostic strategy of opening each interaction with two defections was integral to the evolved strategy's success.

	Function Set 1 (FS1)		Function Set 2 (FS2)		Function Set 3 (FS3)	
Environment Number	**Evolved Strategy**	**Std. Fitness**	**Evolved Strategy**	**Std. Fitness**	**Evolved Strategy**	**Std. Fitness**
1	`D`	0	`D`	0	`D`	0
2	`D`	400	`D`	400	`D`	400
3	`(OPP_LAST D (MY_LAST D C D) (OPP_2ND D C (MY_LAST C C D))) *`	604*	`(HAS_DEFECTED (HAS_COOP C D) D)`	604	`(MY_LAST D C (OPP_LAST C D C))`	703
4	`(OPP_LAST C C D)`	601	`(HAS_DEFECTED D C)`	601	`(OPP_LAST C C D)`	601
5	`C`	400	`C`	400	`C`	400
6	`C`	600	`C`	600	`C`	600
7	`D`	396	`D`	396	`D`	396
8	`(OPP_3RD (MY_LAST D D (OPP_LAST D D C)) (MY_LAST D (OPP_2ND D C D) (OPP_LAST D D C)) (MY_LAST D C D))`	1246	`(MY_3RD (MY_LAST C D D) (MY_LAST C C D) (OPP_LAST C D (OPP_2ND C C (MY_LAST C C D))))`	1236	`(OPP_LAST C (MY_LAST C C D) D)`	1433
* result based on a run where mutation was used at a probability of .3 and crossover at a probability of .7						

Table 6.1 *Summary of results for genetic programming runs against environments of pre-defined opponents*

Additionally, the results show that changes to the function set can limit this diagnostic ability and thus produce less-effective strategies. This became apparent when the function set FS3 was used to evolve strategies in the same environment (Environment 3). The results of this run are summarized in Figure 6.2.

Evolved Strategy:

`(MY_LAST D C (OPP_LAST C D C))`

Evolved Strategy:	**D**CCCCCCC…
`All D`:	DDDDDDDD…
Evolved Strategy:	**D**DDDDDDD…
`All C`:	CCCCCCCC…
Evolved Strategy:	**D**CCCCCCC…
`Tit-for-Tat`:	CDCCCCCC…

Figure 6.2 *A strategy evolved in Environment 3 with Function Set 3*

Strategies developed using the functions in FS3 can only examine the history from the previous move. Therefore, the use of a diagnostic approach was not possible. Because the function set is limited, the strategy had to make do with what it can ascertain from analyzing the previous interaction. As the move history indicates, the strategy's more limited diagnostic approach differentiated between the opponents that are capable of defecting (`Tit-for-Tat` and `All D`) and the opponent that unconditionally cooperates (`All C`). Once it grouped the opponents, it implemented the optimal strategy against each group. Again, the best way to play `All C` is to exploit it via defection. Against the `Tit-for-Tat` and `All D` grouping, it chose to defect on the first move and then cooperate thereafter. This was a calculated choice which sacrificed the one point per game that would have been received by defecting against `All D` in exchange for the two points per game that it gained by cooperating with (rather than defecting against) Tit-for-Tat. The results from Environment 3 show that reducing the function set from FS1 to FS3 can have an effect on the diagnostic capability of the evolved strategies and consequently result in a degradation of performance.

Environment 4, which consisted of Grim Trigger and All D, also exhibited interesting behavior across function sets. When the function set was changed, the performance of the evolved strategy stayed constant, but its representation changed. With the function set equal to FS1, the Tit-for-Tat strategy (OPP_LAST C C D) was evolved. With the function set equal to FS2, the Grim Trigger strategy (HAS_DEFECTED D C) was evolved. As Figure 6.3 shows, both strategies used the same sequence of moves in Environment 4. In this case, varying the function set caused aesthetic rather than functional changes.

Evolved Strategies:
(HAS_DEFECTED D C) and (OPP_LAST C C D)

Evolved Strategy:	CDDDDDDD…
All D:	DDDDDDDD…
Evolved Strategy:	CCCCCCCC…
Grim Trigger:	CCCCCCCC…

Figure 6.3 *Two strategies evolved in Environment 4 with Function Sets 1 and 2*

The results described here from Environments 3 and 4 suggest that genetic programming is effective at evolving capable strategies in specific environments. The evolutionary process was able to adapt an optimal or near-optimal strategy for playing the Repeated Prisoner's Dilemma game based on the function set that it had to work with and the environment of opponents that it was placed in. The results reinforce an important point about competitive domains such as the Repeated Prisoner's Dilemma – a strategy's performance must be understood within the context of the specific environment in which it was evolved. Although the genetic programming paradigm is working properly in these cases, it would be incorrect to interpret the evolved strategies as strategies that generalize well across a variety of environments. Therefore, the next natural step is to evolve strategies based on a richer environment than can be constructed manually. This was achieved through the co-evolutionary experiments which are described in the next section.

Co-evolutionary Environments

Co-evolution evaluates a given individual based on its performance against its peers in the evolutionary population. In the co-evolutionary runs in this study, the population of 1000 consisted of two subpopulations of size 500. An individual's fitness was based on the number of points that it scored when it played the Repeated Prisoner's Dilemma game against 100 randomly-selected members of the opposing subpopulation. During co-evolution, the environment of opponents is constantly changing, making it more difficult for the evolutionary process to adapt individuals for a specific environment. In this predator-prey environment, if one subpopulation develops a particularly useful trait, the opposing subpopulation is forced to adapt, and vice-versa. The intuition is that once the subpopulations stabilize or converge, a reasonably generalized strategy will have emerged.

Early co-evolutionary runs suggested that the chaotic nature of co-evolution can force populations towards defection. Figure 6.4 shows the mean adjusted fitness[2] of Subpopulation 1 for the first 100 generations of a co-evolutionary run using the function set FS1. The strategies that fared well in the early going were those that reduced to `All`

[2] Adjusted Fitness = 1 / (1 + standardized fitness)

D. For example, the best individual in Generation 5 was (MY_LAST D D D). As the graph reveals, no significant increase in fitness was observed during the run. The population eventually converged to the All D strategy.

Obviously, the All D strategy is not a good general strategy for playing the Repeated Prisoner's Dilemma game since it leaves out even the possibility of sustaining cooperation, limiting itself to a payout of *P* for the majority of its interactions. However there are at least two settings in which the All D strategy might be a good idea. The most obvious is in an environment made up of many naïvely cooperating strategies like All C. The second case is a very random or noisy environment. When opponents are unpredictable, the All D strategy hedges by defecting in order to guarantee itself at least one point per round. It is likely that the strategies evolved by this run were forced into permanent defection because of the noisy and unpredictable nature of co-evolution.

Co-evolving solutions with the FS2 function set resulted in a sharp improvement in performance. Figure 6.5 shows how the average fitness of the subpopulation rebounded as the best strategies in each generation were transformed from strategies that tended to unconditionally defect to ones that sustained cooperation. In the early going, the strategies that performed the best were equivalent to All D. For example, the best strategies in the initial generation were (MY_LAST D D D) and (HAS_DEFECTED D D). However, within the first ten generations the HAS_DEFECTED function emerged, producing strategies like (HAS_DEFECTED D (OPP_4TH C C D)) which had the best score in Generation 10. By Generation 12, the Grim Trigger strategy (HAS_DEFECTED D C) took hold, resulting in the large upward spike in mean fitness observed in Figure 6.5. Eventually the population converges to the Grim Trigger strategy. In fact at one point (Generation 48), once the population was dominated by `Grim Trigger` strategies, the strategy `All C` emerged as the best in Subpopulation 2 since it was a smaller individual than `Grim Trigger` and performed just as well as `Grim Trigger` would have performed against itself.

Although the `Grim Trigger` strategy has been argued to be a good general strategy in its own right by Dacey and Pendegraft, the widely-cited success of `Tit-for-Tat` in Axelrod's [1] computerized tournaments necessitates a comparison between `Grim Trigger` and `Tit-for-Tat`. How would `Tit-for-Tat` have fared in the same environment that evolved `Grim Trigger`? To answer this question, the co-evolutionary run that evolved `Grim Trigger` was run a second time. During this additional run, data was collected so that `Tit-for-Tat` could serve as a benchmark strategy without altering the evolutionary process. For each generation in the run, both `Tit-for-Tat` and `Grim Trigger` played the repeated Prisoner's Dilemma against the same 100 randomly selected opponents from Subpopulation 2.

Figure 6.6 plots the performance (in terms of points scored against 100 randomly-selected opponents played) of `Tit-for-Tat`, `Grim Trigger`, and the best strategy from Subpopulation 1. The results show very little variation in performance between `Tit-for-Tat` and `Grim Trigger`. In fact, the two series of data points representing them are virtually identical. In general, each strategy's performance was comparable to the best-of-generation strategy. The interesting downward spikes on the graph are the product of adapting populations and the luck of the draw. At these points, the majority of the 100 opponents selected at random from the population were biased toward defection, forcing both `Tit-for-Tat` and `Grim Trigger` to score poorly. In general,

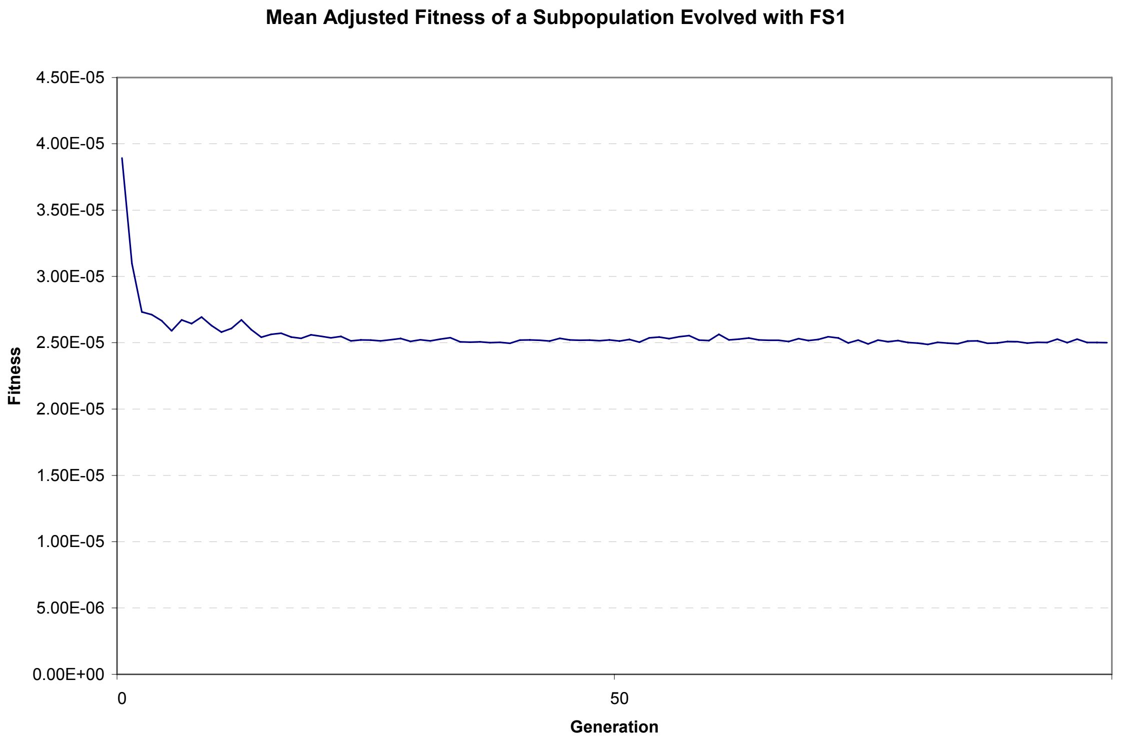


Figure 6.4 *Mean adjusted fitness of a subpopulation during co-evolution with FS1*

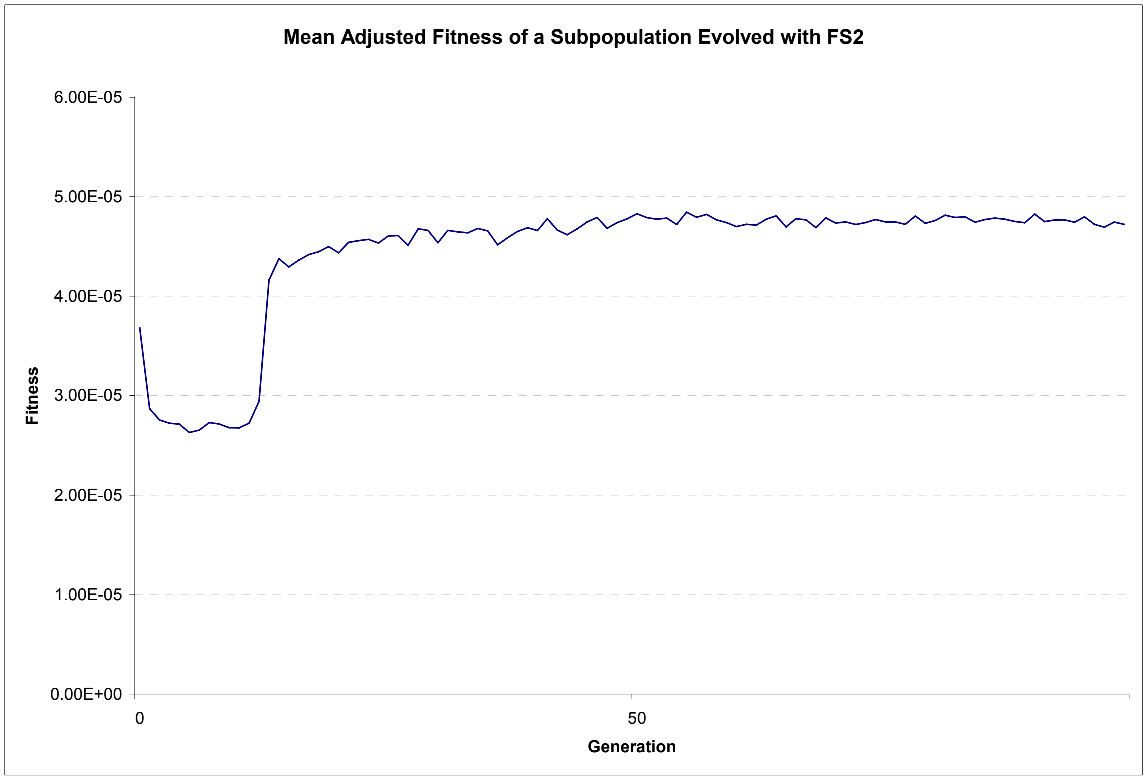


Figure 6.5 *Mean adjusted fitness of a subpopulation during co-evolution with FS2*

however, the data shows that the `Grim Trigger` strategy performed comparably to `Tit-for-Tat` and the best-of-generation strategy.

The fact that `Grim Trigger` emerged from co-evolution speaks to the ability of genetic programming to evolve human-competitive solutions to game theoretic problems like the Repeated Prisoner's Dilemma. By using the population as an evolving environment of opponents, the co-evolutionary process did promote a strategy which was less dependent on a specific environment for its success. However, these results should be interpreted in light of a few important factors.

First, a given strategy's performance is ultimately dependent upon the nature of the opponents that it plays against. For example, no matter how effective `Grim Trigger` is in general, if it were to play in an environment full of `All D` strategies, it would actually do worse than `All D`. In a noisy environment, one misperceived defection from an opponent could cause `Grim Trigger` to throw away chances to improve its score with an otherwise cooperative opponent. These examples do not suggest that `Grim Trigger` is a bad strategy, they just point out that every strategy needs to be understood in the context of its environment.

Second, although co-evolution promoted a changing and diverse environment for strategies to play in, it is important to note that the environment was a product of the function set. In the fixed environment examples, the environment was limited to the pre-defined opponent strategies that were selected. Analogously, in a co-evolutionary environment, the environment is constrained by the strategies that can be produced with the given function set. Since the function set is used to build the environment of opponents during co-evolution, changes to the function set are amplified in a co-evolutionary setting.

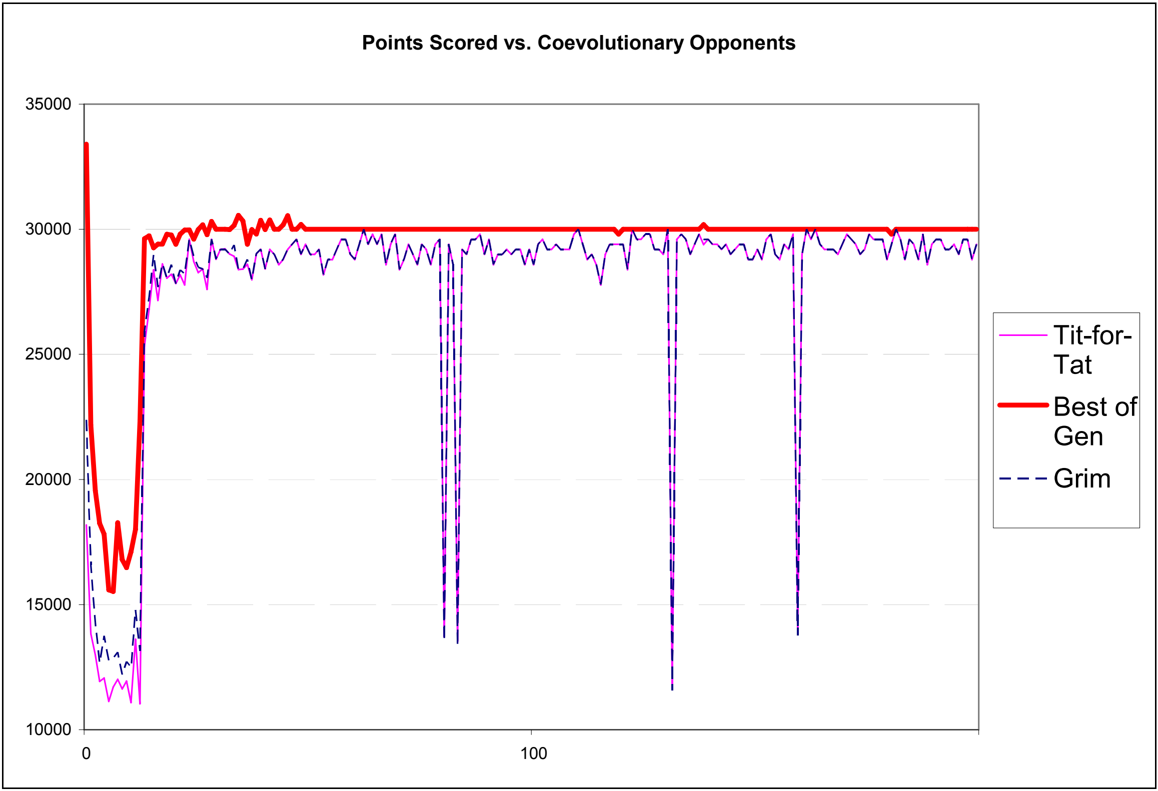


Figure 6.6 *Comparing the performance of Grim Trigger and Tit-for-Tat*

7. Conclusion

Genetic programming was used to successfully generate strategies for the Repeated Prisoner's Dilemma game. First, genetic programming was used to generate strategies in manually-constructed environments of pre-defined opponents. The strategies that were evolved were effective against the specific set of opponents in the static environment, but could not be regarded as good general strategies. Where the effectiveness of these strategies depended upon their diagnostic nature, changes to the function set resulted in non-trivial variation in the performance of the evolved strategies.

In co-evolution, the pre-defined opponents were replaced by a strategy's peers in the evolving population. `Grim Trigger`, a strategy regarded as a good general strategy in previous work, emerged as the most effective strategy. `Grim Trigger` was shown to perform similarly to `Tit-for-Tat` in this environment. The results suggest that changing the function set has an increased effect in co-evolution since the changes alter the environment of opponents.

The application of genetic programming to evolutionary game theory discussed in this paper suggests some interesting directions for future research. From a game-theoretic perspective, the results of the study might be analyzed further. Are strategies that emerge during evolution well-generalized strategies, evolutionary stable, or both? This study also points out that the results of genetic programming runs are sensitive to changes made to the function set. Additional research might build on these findings to provide a more complete understanding of the general characteristics of functions that, when added, tend to improve results.

8. Acknowledgements

Thank you to my advisor Professor Sergio Alvarez for his wealth of insightful comments. His time, support, and expertise were integral to the success of this project. Thank you to my parents and friends for supporting me over the course of the year.
I would also like to thank Liviu Panait and Sean Luke for their technical assistance with the ECJ software. Lastly, thank you to Mark Irvine for introducing me to the Prisoner's Dilemma and convincing me to sign up for the masochistic undertaking that is the Senior Honors Thesis – I enjoyed the journey.

Appendix A: Java Source Code for Opponent Strategies

```
abstract class Strategy
{
    char[] myHistory;
    char[] oppHistory;
    int moveNumber;
    int gameLength;
    boolean hasDefected;
    boolean hasCooperated;

    public Strategy()
    {
       this.reset();
    }

    public void updateHistory(char myMove, char oppMove, int num)
    {
       moveNumber = num;
       myHistory[moveNumber]  = myMove;
       oppHistory[moveNumber] = oppMove;
       if (oppMove == 'D'){
           hasDefected = true;
       }else{
           hasCooperated = true;
       }
    }

    public void reset(){
       hasDefected = false;
       hasCooperated = false;
       moveNumber = 0;
       gameLength = 100;
       myHistory  = new char[gameLength];
       oppHistory = new char[gameLength];

       for(int i=0; i < myHistory.length; i++)
       {
           myHistory[i]  = 'U';
           oppHistory[i] = 'U';
       }
    }

    abstract char getMove();
}
```

```
class AllC extends Strategy
{
    public char getMove()
    {
       return 'C';
    }
}
```

```
class AllD extends Strategy
{
    public char getMove()
    {
       return 'D';
    }
}
```

```
class GrimTrigger extends Strategy
{
    public char getMove()
    {
       if(hasDefected){
              return 'D';
       }else{
              return 'C';
       }
    }
}
```

```
class Pavlov extends Strategy
{
    public char getMove()
    {
       if(moveNumber == 0) return 'C';

       if(oppHistory[moveNumber-1] == myHistory[moveNumber-1]){
              return 'C';
       }else{
              return 'D';
       }
    }
}
```

```
class TitForTat extends Strategy
{
    public char getMove()
    {
       if(moveNumber == 0) return 'C';

       if(oppHistory[moveNumber-1] == 'C'){
           return 'C';
       }else{
           return 'D';
       }
    }
}
```

```
class CBTFT extends Strategy
{
    public char getMove()
    {
       if(moveNumber == 0) return 'C';

       if(oppHistory[moveNumber-1] == 'C'){
           return 'D';
       }else{
           return 'C';
       }
    }
}
```

```
class DBTFT extends Strategy
{
    public char getMove()
    {
       if(moveNumber == 0) return 'D';

       if(oppHistory[moveNumber-1] == 'C'){
           return 'D';
```

```
        }else{
            return 'C';
        }
    }
}
```

```
class DPC extends Strategy
{
    public char getMove()
    {
        if(hasCooperated){
            return 'C';
        }else{
            return 'D';
        }
    }
}
```

```
class DTFT extends Strategy
{
    public char getMove()
    {
        if(moveNumber == 0) return 'D';

        if(oppHistory[moveNumber-1] == 'C'){
            return 'C';
        }else{
            return 'D';
        }
    }
}
```

Appendix B: Java Source Code for Functions and Terminals

For brevity, the source code for the functions and terminals used in static environment experiments is not included. The source code shown here was used for co-evolution. It contains modifications to the classes used in the static environments which facilitate co-evolution and add the ability to specify a pre-defined strategy for use as a benchmark in co-evolutionary environments.

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class C extends GPNode{

    public String toString() { return "C"; }

    public void checkConstraints(final EvolutionState state,
                                final int tree,
                                final GPIndividual typicalIndividual,
                                final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=0)
           state.output.error("Incorrect number of children for node " +
                             toStringForError() + " at " +
                             individualBase);
    }

    public void eval(final EvolutionState state,
                    final int thread,
                    final GPData input,
                    final ADFStack stack,
                    final GPIndividual individual,
                    final Problem problem)
    {
       PDdata data = ((PDdata)(input));
       data.x = 'C';
    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class D extends GPNode{

    public String toString() { return "D"; }

    public void checkConstraints(final EvolutionState state,
                           final int tree,
                           final GPIndividual typicalIndividual,
                           final Parameter individualBase)
    {
      super.checkConstraints(state,tree,typicalIndividual,individualBase);
      if (children.length!=0)
          state.output.error("Incorrect number of children for node " +
                          toStringForError() + " at " +
                          individualBase);
    }

    public void eval(final EvolutionState state,
                  final int thread,
                  final GPData input,
                  final ADFStack stack,
                  final GPIndividual individual,
                  final Problem problem)
    {
      PDdata data = ((PDdata)(input));
      data.x = 'D';
    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;

import ec.app.pd_compete.*;

public class HAS_COOP extends GPNode {
    public String toString() { return "HAS_COOP"; }

    public void checkConstraints(final EvolutionState state,
                           final int tree,
                           final GPIndividual typicalIndividual,
                           final Parameter individualBase)
    {
      super.checkConstraints(state,tree,typicalIndividual,individualBase);
      if (children.length!=2)
          state.output.error("Incorrect number of children for node " +
                          toStringForError() + " at " +
                          individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
       PD pd = (PD)problem;

      boolean oppHasCooperated;
      if (pd.whosTurn == pd.P1){
          oppHasCooperated = pd.p2HasCooperated;
      }else if(pd.whosTurn == pd.P2){
          oppHasCooperated = pd.p1HasCooperated;
      }else{
          oppHasCooperated = pd.p3HasCooperated;
      }
```

```
       if(oppHasCooperated){
           children[0].eval(state,thread,input,stack,individual,problem);
       }else{
           children[1].eval(state,thread,input,stack,individual,problem);
       }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;

import ec.app.pd_compete.*;

public class HAS_DEFECTED extends GPNode {
    public String toString() { return "HAS_DEFECTED"; }

    public void checkConstraints(final EvolutionState state,
                                final int tree,
                                final GPIndividual typicalIndividual,
                                final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=2)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
        PD pd = (PD)problem;

       boolean oppHasDefected;
       if (pd.whosTurn == pd.P1){
           oppHasDefected = pd.p2HasDefected;
       }else if (pd.whosTurn == pd.P2){
           oppHasDefected = pd.p1HasDefected;
       }else{
           oppHasDefected = pd.p3HasDefected;
       }

       if(oppHasDefected){
           children[0].eval(state,thread,input,stack,individual,problem);
       }else{
           children[1].eval(state,thread,input,stack,individual,problem);
       }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class MY_2ND extends GPNode {
    public String toString() { return "MY_2ND"; }

    public void checkConstraints(final EvolutionState state,
                                 final int tree,
                                 final GPIndividual typicalIndividual,
                                 final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=3)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
        PD pd = (PD)problem;

       char temp;
       if (pd.whosTurn == pd.P1){
           temp = (pd.p1Moves.get(1)).charValue();
       }else if (pd.whosTurn == pd.P2){
           temp = (pd.p2Moves.get(1)).charValue();
       }else {
           temp = (pd.p2Moves2.get(1)).charValue();
       }

       switch(temp){
       case 'C':
           children[1].eval(state,thread,input,stack,individual,problem);
           break;

       case 'D':
           children[2].eval(state,thread,input,stack,individual,problem);
           break;

       default:  // assume undefined
           children[0].eval(state,thread,input,stack,individual,problem);
           break;
       }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;
```

```
public class MY_3RD extends GPNode {
    public String toString() { return "MY_3RD"; }

    public void checkConstraints(final EvolutionState state,
                                 final int tree,
                                 final GPIndividual typicalIndividual,
                                 final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=3)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
         PD pd = (PD)problem;

       char temp;
       if (pd.whosTurn == pd.P1){
           temp = (pd.p1Moves.get(2)).charValue();
       }else if (pd.whosTurn == pd.P2){
           temp = (pd.p2Moves.get(2)).charValue();
       }else {
           temp = (pd.p2Moves2.get(2)).charValue();
       }

       switch(temp){
       case 'C':
           children[1].eval(state,thread,input,stack,individual,problem);
           break;

       case 'D':
           children[2].eval(state,thread,input,stack,individual,problem);
           break;

       default:  // assume undefined
           children[0].eval(state,thread,input,stack,individual,problem);
           break;
       }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class MY_4TH extends GPNode {
    public String toString() { return "MY_4TH"; }

    public void checkConstraints(final EvolutionState state,
                              final int tree,
                              final GPIndividual typicalIndividual,
                              final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=3)
           state.output.error("Incorrect number of children for node " +
                             toStringForError() + " at " +
                             individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
        PD pd = (PD)problem;

       char temp;
       if (pd.whosTurn == pd.P1){
           temp = (pd.p1Moves.get(3)).charValue();
       }else if(pd.whosTurn == pd.P2){
           temp = (pd.p2Moves.get(3)).charValue();
       }else{
           temp = (pd.p2Moves2.get(3)).charValue();
       }

       switch(temp){
       case 'C':
           children[1].eval(state,thread,input,stack,individual,problem);
           break;

       case 'D':
           children[2].eval(state,thread,input,stack,individual,problem);
           break;

       default:  // assume undefined
           children[0].eval(state,thread,input,stack,individual,problem);
           break;
       }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class MY_LAST extends GPNode {
    public String toString() { return "MY_LAST"; }

    public void checkConstraints(final EvolutionState state,
                              final int tree,
                              final GPIndividual typicalIndividual,
                              final Parameter individualBase)
    {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=3)
            state.output.error("Incorrect number of children for node " +
                            toStringForError() + " at " +
                            individualBase);
    }

    public void eval(final EvolutionState state,
                    final int thread,
                    final GPData input,
                    final ADFStack stack,
                    final GPIndividual individual,
                    final Problem problem)
    {
         PD pd = (PD)problem;

        char temp;
        if (pd.whosTurn == pd.P1){
            temp = (pd.p1Moves.get(0)).charValue();
        }else if(pd.whosTurn == pd.P2){
            temp = (pd.p2Moves.get(0)).charValue();
        }else{
            temp = (pd.p2Moves2.get(0)).charValue();
        }

        switch(temp){
        case 'C':
            children[1].eval(state,thread,input,stack,individual,problem);
            break;

        case 'D':
            children[2].eval(state,thread,input,stack,individual,problem);
            break;

        default:  // assume undefined
            children[0].eval(state,thread,input,stack,individual,problem);
            break;
        }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class OPP_2ND extends GPNode {
    public String toString() { return "OPP_2ND"; }

    public void checkConstraints(final EvolutionState state,
                                final int tree,
                                final GPIndividual typicalIndividual,
                                final Parameter individualBase)
    {
        super.checkConstraints(state,tree,typicalIndividual,individualBase);
        if (children.length!=3)
            state.output.error("Incorrect number of children for node " +
                            toStringForError() + " at " +
                        individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
         PD pd = (PD)problem;

       char temp;
       if (pd.whosTurn == pd.P1){
           temp = (pd.p2Moves.get(1)).charValue();
       }else if(pd.whosTurn == pd.P2){
           temp = (pd.p1Moves.get(1)).charValue();
       }else{
           temp = (pd.p3Moves.get(1)).charValue();
       }

       switch(temp){
       case 'C':
           children[1].eval(state,thread,input,stack,individual,problem);
           break;

       case 'D':
           children[2].eval(state,thread,input,stack,individual,problem);
           break;

       default:  // assume undefined
           children[0].eval(state,thread,input,stack,individual,problem);
           break;
       }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class OPP_3RD extends GPNode {
    public String toString() { return "OPP_3RD"; }

    public void checkConstraints(final EvolutionState state,
                                 final int tree,
                                 final GPIndividual typicalIndividual,
                                 final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=3)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
    }

    public void eval(final EvolutionState state,
                    final int thread,
                    final GPData input,
                    final ADFStack stack,
                    final GPIndividual individual,
                    final Problem problem)
    {
        PD pd = (PD)problem;

       char temp;
       if (pd.whosTurn == pd.P1){
           temp = (pd.p2Moves.get(2)).charValue();
       }else if (pd.whosTurn == pd.P2){
           temp = (pd.p1Moves.get(2)).charValue();
       }else {
           temp = (pd.p3Moves.get(3)).charValue();
       }

       switch(temp){
       case 'C':
           children[1].eval(state,thread,input,stack,individual,problem);
           break;

       case 'D':
           children[2].eval(state,thread,input,stack,individual,problem);
           break;

       default:  // assume undefined
           children[0].eval(state,thread,input,stack,individual,problem);
           break;
       }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class OPP_4TH extends GPNode {
    public String toString() { return "OPP_4TH"; }

    public void checkConstraints(final EvolutionState state,
                            final int tree,
                            final GPIndividual typicalIndividual,
                            final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=3)
           state.output.error("Incorrect number of children for node " +
                            toStringForError() + " at " +
                            individualBase);
    }

    public void eval(final EvolutionState state,
                    final int thread,
                    final GPData input,
                    final ADFStack stack,
                    final GPIndividual individual,
                    final Problem problem)
    {
        PD pd = (PD)problem;

       char temp;
       if (pd.whosTurn == pd.P1){
           temp = (pd.p2Moves.get(3)).charValue();
       }else if (pd.whosTurn == pd.P2){
           temp = (pd.p1Moves.get(3)).charValue();
       }else {
           temp = (pd.p3Moves.get(3)).charValue();
       }

       switch(temp){
       case 'C':
           children[1].eval(state,thread,input,stack,individual,problem);
           break;

       case 'D':
           children[2].eval(state,thread,input,stack,individual,problem);
           break;

       default:  // assume undefined
           children[0].eval(state,thread,input,stack,individual,problem);
           break;
       }

    }
}
```

```
package ec.app.pd_compete.func;
import ec.*;
import ec.gp.*;
import ec.util.*;
import ec.coevolve.*;

import ec.app.pd_compete.*;

public class OPP_LAST extends GPNode {
    public String toString() { return "OPP_LAST"; }

    public void checkConstraints(final EvolutionState state,
                                 final int tree,
                                 final GPIndividual typicalIndividual,
                                 final Parameter individualBase)
    {
       super.checkConstraints(state,tree,typicalIndividual,individualBase);
       if (children.length!=3)
           state.output.error("Incorrect number of children for node " +
                              toStringForError() + " at " +
                              individualBase);
    }

    public void eval(final EvolutionState state,
                     final int thread,
                     final GPData input,
                     final ADFStack stack,
                     final GPIndividual individual,
                     final Problem problem)
    {
        PD pd = (PD)problem;
       char temp;
       if (pd.whosTurn == pd.P1){
           temp = (pd.p2Moves.get(0)).charValue();
       }else if(pd.whosTurn == pd.P2){
           temp = (pd.p1Moves.get(0)).charValue();
       }else {
           temp = (pd.p3Moves.get(0)).charValue();
       }

       switch(temp){
       case 'C':
           children[1].eval(state,thread,input,stack,individual,problem);
           break;

       case 'D':
           children[2].eval(state,thread,input,stack,individual,problem);
           break;

       default:  // assume undefined
           children[0].eval(state,thread,input,stack,individual,problem);
           break;
       }

    }
}
```

Appendix C: Java Source Code for the Prisoner's Dilemma Problem

The following Java source code is the formulation of the Prisoner's Dilemma problem that was used to co-evolve solutions in ECJ.

```
package ec.app.pd_compete;

import ec.util.*;
import ec.*;
import ec.gp.*;
import ec.gp.koza.*;
import ec.simple.*;
import ec.coevolve.*;

import java.util.*;

public class PD extends GPProblem implements GroupedProblemForm{

    public static final String P_DATA = "data";

    public PDdata input;

    //values of "My" moves and Opponent's moves
    public LinkedList<Character> p1Moves;
    public LinkedList<Character> p2Moves;

    public LinkedList<Character> p3Moves;
    public LinkedList<Character> p2Moves2;

    //temp variables for current moves
    public char p1Move;
    public char p2Move;
    public char p3Move;
    public char p2Move2;

    //cooperation and defection information
    public boolean p1HasDefected;
    public boolean p1HasCooperated;
    public boolean p2HasDefected;
    public boolean p2HasCooperated;

    public boolean p3HasDefected;
    public boolean p3HasCooperated;
    public boolean p2HasDefected2;
    public boolean p2HasCooperated2;

    public int p1defectCount;
    public int p2defectCount;
    public int p2defectCount2;
    public int p3defectCount;

    public int moveNum;

    //random number generator
    public MersenneTwisterFast rand;
```

```
//Define payoff values here to make it easier to adjust them if needed
public static final int MUTUAL_COOP = 3;
public static final int MUTUAL_DEFECT = 1;
public static final int TEMPTATION = 5;
public static final int SUCKER = 0;

//A variable used in the functions from the function set to determine who
//is P1 and who is P2
public int whosTurn;
public static final int P1 = 1;
public static final int P2 = 2;
public static final int P3 = 3;

//comparison strategies
private Strategy p3;

//for printing p3 info
private int p3score = 0;
private int oppCount = 0;
private int maxScore = 0;

public Object protoClone() throws CloneNotSupportedException
{
    PD newobj = (PD) (super.protoClone());
    newobj.input = (PDdata)(input.protoClone());
    return newobj;
}
public void setup(final EvolutionState state,final Parameter base)
{
    // very important, remember this
    super.setup(state,base);

    input =(PDdata)state.parameters.getInstanceForParameterEq(
            base.push(P_DATA), null, PDdata.class);
    input.setup(state,base.push(P_DATA));

     p3 = new GrimTrigger();

     //set up random number generator
     rand = new MersenneTwisterFast(3252354);
}


public void preprocessPopulation( final EvolutionState state,
                                  Population pop )
{
   int opps = (state.parameters).getInt(
                 new Parameter("eval.subpop.0.num-rand-ind"));
   maxScore = opps*TEMPTATION*100;

    for( int i = 0 ; i < pop.subpops.length ; i++ )
        for( int j = 0 ; j < pop.subpops[i].individuals.length ; j++ )
             ((KozaFitness)(pop.subpops[i].individuals[j].fitness)).
                 setStandardizedFitness(state, (float)opps*TEMPTATION*100 );

}

public void postprocessPopulation( final EvolutionState state,
                                   Population pop )
{
    for( int i = 0 ; i < pop.subpops.length ; i++ )
        for( int j = 0 ; j < pop.subpops[i].individuals.length ; j++ )
        {
            pop.subpops[i].individuals[j].evaluated = true;
           }

      //report and reset scoring for p3
      int standardized = maxScore-p3score;
      System.out.print("Generation " + state.generation + ",");
      System.out.println(p3score + "," + standardized);
      p3score = 0;
      oppCount = 0;
   }

    public void evaluate(final EvolutionState state,
                         final Individual[] ind,
                         final boolean[] updateFitness,
                         final boolean countVictoriesOnly,
                         final int threadnum)
   {

      int sum1    = 0;
      int result1 = 0;

      int sum2    = 0;
      int result2 = 0;

      int sum3    = 0;
      int result3 = 0;

      //reset move history
      p1Moves = new LinkedList<Character>();
      p2Moves = new LinkedList<Character>();
      p3Moves = new LinkedList<Character>();
      p2Moves2 = new LinkedList<Character>();

      p1HasDefected   = false;
      p1HasCooperated = false;
      p2HasDefected   = false;
      p2HasCooperated = false;

      p3HasDefected   = false;
      p3HasCooperated = false;
      p2HasDefected2   = false;
      p2HasCooperated2 = false;

      p1defectCount  = 0;
      p2defectCount  = 0;
      p2defectCount2 = 0;
      p3defectCount  = 0;

      p3.reset();

      for(int x=0; x < 4; x++){
          p1Moves.addFirst(new Character('U'));
          p2Moves.addFirst(new Character('U'));
          p3Moves.addFirst(new Character('U'));
          p2Moves2.addFirst(new Character('U'));
      }

      for(int j=0;j<100;j++){//play the game 100 times
          moveNum = j;

          whosTurn = P1;
          //Evaluate the individual to get Player 1's move
          ((GPIndividual)ind[0]).trees[0].child.eval(
                   state,threadnum,input,stack,((GPIndividual)ind[0]),this);
          p1Move = input.x;
          if(p1Move == 'C'){
             p1HasCooperated = true;
          }else{
             p1HasDefected = true;
             p1defectCount++;
          }
          whosTurn = P2;
          //Evaluate the individual to get Player 2's move
          ((GPIndividual)ind[1]).trees[0].child.eval(
                   state,threadnum,input,stack,((GPIndividual)ind[1]),this);
          p2Move = input.x;
          if(p2Move == 'C'){
             p2HasCooperated = true;
          }else{
             p2HasDefected = true;
             p2defectCount++;
          }

          //calculate each individual's payout based on given moves
          result1 = getPayout(p1Move, p2Move);
          result2 = getPayout(p2Move, p1Move);

          //keep a tally for how each player is doing
          sum1 += result1;
          sum2 += result2;

          //Update both players' move history
          p1Moves.addFirst(new Character(p1Move));
          p2Moves.addFirst(new Character(p2Move));

          //--- Play TFT (or another Strategy) against subpopulation 2 ---//

          whosTurn = P3;
          //Get Strategy's move
          p3Move = p3.getMove();
          if(p3Move == 'C'){
             p3HasCooperated = true;
          }else{
             p3HasDefected = true;
             p3defectCount++;
          }

          //how would p2 have played against p3?
          //Evaluate the individual to get Player 2's move
          ((GPIndividual)ind[1]).trees[0].child.eval(
                   state,threadnum,input,stack,((GPIndividual)ind[1]),this);
          p2Move2 = input.x;
          if(p2Move2 == 'C'){
             p2HasCooperated2 = true;
          }else{
             p2HasDefected2 = true;
             p2defectCount2++;
          }

          //calculate P3's payout based on given moves
          result3 = getPayout(p3Move, p2Move2);

          //keep a tally for how each player is doing
          sum3 += result3;

          //Update both players' move history
```

```
            p3Moves.addFirst(new Character(p3Move));
            p2Moves2.addFirst(new Character(p2Move2));

            p3.updateHistory(p3Move, p2Move, j);

            //-------- Done playing TFT against subpopulation 2 --------//
        }

        if (sum1 >= 100*MUTUAL_DEFECT)
               ((KozaFitness)(ind[0].fitness)).hits=
                      ((KozaFitness)(ind[0].fitness)).hits + 1;
        if (sum2 >= 100*MUTUAL_DEFECT)
               ((KozaFitness)(ind[1].fitness)).hits=
                      ((KozaFitness)(ind[1].fitness)).hits + 1;

         float prevFit1 = ((KozaFitness)(ind[0].fitness)).rawFitness();
         float prevFit2 = ((KozaFitness)(ind[1].fitness)).rawFitness();

        if( updateFitness[0] )
        {
            ((KozaFitness)(ind[0].fitness)).setStandardizedFitness(
                      state,(float)(prevFit1-sum1));
        }

        if( updateFitness[1] )
        {
            ((KozaFitness)(ind[1].fitness)).setStandardizedFitness(
                      state,(float)(prevFit2-sum2));
        }

        //update p3's score
        if(oppCount < 100)
            p3score += sum3;
        oppCount++;
    }

    //user-defined function that calculates the outcome value
    //of the game
    private int getPayout(char myMove, char oppMove)
    {
        if(myMove == oppMove){
            if(myMove == 'C'){
               return MUTUAL_COOP;
            }else{
               return MUTUAL_DEFECT;
            }
        }else{
            if(myMove == 'C'){
               return SUCKER;
            }else{
               return TEMPTATION;
            }
        }
    }
}
```

Appendix D: Sample ECJ parameter file

This is the ECJ parameter file that was used for co-evolutionary runs with FS2

```
parent.0 = ../../gp/koza/koza.params

generations = 500
evalthreads = 1
breedthreads = 1
seed.0 = 55678

checkpoint = false
checkpoint-modulo = 10

# set up statistics
stat.num-children      = 2
stat.child.0           = ec.gp.koza.KozaStatistics
stat.child.1        = ec.gp.koza.KozaShortStatistics
stat.child.0.file      = $out.stat
stat.child.1.file      = $out.short

# set up population
pop.subpops =                   2
pop.subpop.0 =                        ec.Subpopulation
pop.subpop.1 =                        ec.Subpopulation

# we are using competitive coevolution
eval =
ec.coevolve.MultiPopCoevolutionaryEvaluator

#the num-rand-ind set here is used in fitness calculation -- make sure subpop 0
#and 1 are the same number
eval.subpop.0.num-rand-ind =           100
eval.subpop.0.num-elites =       0
eval.subpop.0.num-ind =                0
eval.subpop.0.select =                 ec.select.TournamentSelection
eval.subpop.0.select.size =            1

eval.subpop.1.num-rand-ind =           100
eval.subpop.1.num-elites =       0
eval.subpop.1.num-ind =                0
eval.subpop.1.select =                 ec.select.TournamentSelection
eval.subpop.1.select.size =            1

# set up subpopulations
pop.subpop.0.size = 500
pop.subpop.1.size = 500

pop.subpop.0.fitness =                 ec.gp.koza.KozaFitness
pop.subpop.1.fitness =                 ec.gp.koza.KozaFitness

pop.subpop.0.duplicate-retries =  100
pop.subpop.1.duplicate-retries =  100



pop.subpop.0.species = ec.gp.GPSpecies
pop.subpop.0.species.ind = ec.gp.GPIndividual
pop.subpop.1.species = ec.gp.GPSpecies
pop.subpop.1.species.ind = ec.gp.GPIndividual

pop.subpop.0.species.ind.numtrees = 1
pop.subpop.0.species.ind.tree.0 = ec.gp.GPTree
pop.subpop.0.species.ind.tree.0.tc = tc0
pop.subpop.1.species.ind.numtrees = 1
pop.subpop.1.species.ind.tree.0 = ec.gp.GPTree
pop.subpop.1.species.ind.tree.0.tc = tc0

pop.subpop.0.species.pipe = ec.breed.MultiBreedingPipeline
pop.subpop.0.species.pipe.generate-max = false
pop.subpop.0.species.pipe.num-sources = 3
pop.subpop.0.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline
pop.subpop.0.species.pipe.source.0.prob = 0.7
pop.subpop.0.species.pipe.source.1 = ec.breed.ReproductionPipeline
pop.subpop.0.species.pipe.source.1.prob = 0.3
pop.subpop.0.species.pipe.source.2 = ec.gp.koza.MutationPipeline
pop.subpop.0.species.pipe.source.2.prob = 0.0

pop.subpop.1.species.pipe = ec.breed.MultiBreedingPipeline
pop.subpop.1.species.pipe.generate-max = false
pop.subpop.1.species.pipe.num-sources = 3
pop.subpop.1.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline
pop.subpop.1.species.pipe.source.0.prob = 0.7
pop.subpop.1.species.pipe.source.1 = ec.breed.ReproductionPipeline
pop.subpop.1.species.pipe.source.1.prob = 0.3
pop.subpop.1.species.pipe.source.2 = ec.gp.koza.MutationPipeline
pop.subpop.1.species.pipe.source.2.prob = 0.0

breed.reproduce.source.0 = ec.parsimony.LexicographicTournamentSelection
gp.koza.mutate.source.0 = ec.parsimony.LexicographicTournamentSelection
gp.koza.xover.source.0 = ec.parsimony.LexicographicTournamentSelection
gp.koza.xover.source.1 = ec.parsimony.LexicographicTournamentSelection
select.lexicographic-tournament.size = 7

#add elitism
breed.elite.0 = 10
breed.elite.1 = 10

# We have one function set, of class GPFunctionSet
gp.fs.size = 1
gp.fs.0 = ec.gp.GPFunctionSet
# We'll call the function set "f0".  It uses the default GPFuncInfo class
gp.fs.0.name = f0
gp.fs.0.info = ec.gp.GPFuncInfo

# We have functions in the function set.  They are:
gp.fs.0.size = 12
gp.fs.0.func.0 = ec.app.pd_compete.func.C
gp.fs.0.func.0.nc = nc0
gp.fs.0.func.1 = ec.app.pd_compete.func.D
gp.fs.0.func.1.nc = nc0
gp.fs.0.func.2 = ec.app.pd_compete.func.MY_LAST
gp.fs.0.func.2.nc = nc3
gp.fs.0.func.3 = ec.app.pd_compete.func.MY_2ND
gp.fs.0.func.3.nc = nc3
gp.fs.0.func.4 = ec.app.pd_compete.func.MY_3RD
gp.fs.0.func.4.nc = nc3
gp.fs.0.func.5 = ec.app.pd_compete.func.MY_4TH
gp.fs.0.func.5.nc = nc3
gp.fs.0.func.6 = ec.app.pd_compete.func.OPP_LAST
gp.fs.0.func.6.nc = nc3
gp.fs.0.func.7 = ec.app.pd_compete.func.OPP_2ND
gp.fs.0.func.7.nc = nc3
gp.fs.0.func.8 = ec.app.pd_compete.func.OPP_3RD
gp.fs.0.func.8.nc = nc3
gp.fs.0.func.9 = ec.app.pd_compete.func.OPP_4TH
gp.fs.0.func.9.nc = nc3
gp.fs.0.func.10 = ec.app.pd_compete.func.HAS_DEFECTED
gp.fs.0.func.10.nc = nc2
gp.fs.0.func.11 = ec.app.pd_compete.func.HAS_COOP
gp.fs.0.func.11.nc = nc2

eval.problem = ec.app.pd_compete.PD
eval.problem.data = ec.app.pd_compete.PDdata

# The following should almost *always* be the same as eval.problem.data
# For those who are interested, it defines the data object used internally
# inside ADF stack contexts
eval.problem.stack.context.data = ec.app.pd_compete.PDdata
```

References

[1] Axelrod, Robert. *The Evolution of Cooperation*. New York: Basic Books, 1984.

[2] Axelrod, Robert. *The Complexity of Cooperation*. Princeton: Princeton University Press, 1997.

[3] Dacey, Raymond and Norman Pendegraft. "The Optimality of Tit-For-Tat." *International Interactions* 15, no. 1 (1988): 45-64.

[4] Dixit, Avinash and Susan Skeath. *Games of Strategy*. 2nd ed. New York: W.W. Norton, 2004.

[5] Fujiki, Cory and John Dickinson. "Using the Genetic Algorithm to Generate Lisp Source Code to Solve the Prisoner's Dilemma." *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. Edited by John J. Grefenstette, 236-40. Hillsdale: Erlbaum, 1987.

[6] Golbeck, Jennifer. "Evolving Strategies for the Prisoner's Dilemma." *Advances in Intelligent Systems, Fuzzy Systems, and Evolutionary Computation*. February 2002, p 299-306.

[7] Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor: University of Michigan Press, 1975.

[8] Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge: MIT Press, 1992.

[9] Lowsky, David. "Using a Cooperative Fitness Function to Coevolve Optimal Strategies in the Iterated Prisoner's Dilemma Game." *Genetic Algorithms and Genetic Programming at Stanford*. Edited by John R. Koza, 131-39. Stanford, 1999.

[10] Luke, Sean and Liviu Panait. "Lexicographic Parsimony Pressure." *GECCO-2002: Proceedings of the Genetic and Evolutionary Computation Conference.* Edited by W. B. Langdon et al, 829-36. Morgan Kauffman, 2002.

[11] Luke, Sean et al. *ECJ: A Java-based Evolutionary Computation Research System*. Version 13. Available at http://cs.gmu.edu/~eclab/projects/ecj/

[12] Miller, John Howard. "Two Essays on the Economics of Imperfect Information." PhD diss., University of Michigan, 1988.