Chapter 10

Is it Accidental or Intentional? A Symbolic Approach to Noisy Iterated Prisoner's Dilemma

Tsz-Chiu Au and Dana Nau
Department of Computer Science and Institute for Systems Research
University of Maryland, College Park

10.1 Introduction

The Iterated Prisoner's Dilemma (IPD) has become well known as an abstract model of a class of multi-agent environments in which agents accumulate payoffs that depend on how successful they are in their repeated interactions with other agents. An important variant of the IPD is the *Noisy* IPD, in which there is a small probability, called the *noise level*, that accidents will occur. In other words, the noise level is the probability of executing "cooperate" when "defect" was the intended move, or vice versa.

Accidents can cause difficulty in cooperations with others in real-life situations, and the same is true in the Noisy IPD. Strategies that do quite well in the ordinary (non-noisy) IPD may do quite badly in the Noisy IPD [Axelrod and Dion, 1988; Bendor, 1987; Bendor et al., 1991; Molander, 1985; Mueller, 1987; Nowak and Sigmund, 1990]. For example, if two players both use the well-known Tit-For-Tat (TFT) strategy, then an accidental defection may cause a long series of defections by both players as each of them punishes the other for defecting.

This chapter reports on a strategy called the Derived Belief Strategy (DBS), which was the best-performing non-master-slave strategy in Category 2 (noisy environments) of the 2005 Iterated Prisoner's Dilemma com-

Table 10.1 Scores of the best programs in Competition 2 (IPD with Noise). The table shows each program's average score for each run and its overall average over all five runs. The competition included 165 programs, but we have listed only the top 25.

			Score					
Rank	Program	Author	Run1	Run2	Run3	Run4	Run5	Avg.
1	BWIN	P. Vytelingum	441.7	431.7	427.1	434.8	433.5	433.8
2	IMM01	J.W. Li	424.7	414.6	414.7	409.1	407.5	414.1
3	DBSz	T.C. Au	411.7	405.0	406.5	407.7	409.2	408.0
4	DBSy	T.C. Au	411.9	407.5	407.9	407.0	405.5	408.0
5	DBSpl	T.C. Au	409.5	403.8	411.4	403.9	409.1	407.5
6	DBSx	T.C. Au	401.9	410.5	407.7	408.4	404.4	406.6
7	DBSf	T.C. Au	399.2	402.2	405.2	398.9	404.4	402.0
8	DBStft	T.C. Au	398.4	394.3	402.1	406.7	407.3	401.8
9	DBSd	T.C. Au	406.0	396.0	399.1	401.8	401.5	400.9
10	lowES-TFT_classic	M. Filzmoser	391.6	395.8	405.9	393.2	399.4	397.2
11	TFTIm	T.C. Au	399.0	398.8	395.0	396.7	395.3	397.0
12	Mod	P. Hingston	394.8	394.2	407.8	394.1	393.7	396.9
13	TFTIz	T.C. Au	397.7	396.1	390.7	392.1	400.6	395.5
14	TFTIc	T.C. Au	400.1	401.0	389.5	388.9	389.2	393.7
15	DBSe	T.C. Au	396.9	386.8	396.7	394.5	393.7	393.7
16	TTFT	L. Clement	389.1	395.8	394.1	393.4	394.7	393.4
17	TFTIa	T.C. Au	389.5	394.4	395.1	389.6	397.7	393.3
18	TFTIb	T.C. Au	391.7	390.0	390.5	401.0	392.4	393.1
19	TFTIx	T.C. Au	398.3	391.3	390.8	391.0	393.7	393.0
20	mediumES-TFT_classic	M. Filzmoser	396.7	392.6	398.3	390.8	386.0	392.9
21	TFTIy	T.C. Au	391.7	394.6	390.8	392.1	394.9	392.8
22	TFTId	T.C. Au	395.6	393.1	388.8	385.7	391.3	390.9
23	TFTIe	T.C. Au	396.7	391.1	385.2	388.2	393.5	390.9
24	DBSb	T.C. Au	393.2	386.1	392.6	391.1	391.0	390.8
25	T4T	D. Fogel	391.5	387.6	400.4	387.3	383.5	390.0

petition (see Table 10.1).

Like most opponent-modeling techniques, DBS attempts to learn a model of the other player's strategy (i.e., the *opponent model*[1]) during the games. Our main innovation involves how to reason about noise using the opponent model.

The key idea used in DBS is something that we call *symbolic noise detection*—the use of the other player's deterministic behavior to tell whether an action has been affected by noise. More precisely, DBS builds a symbolic model of how the other player behaves, and watches for any

[1]The term "opponent model" appears to be the most common term for a model of the other player, even though this player is not necessarily an "opponent" (since the IPD is not zero-sum).

deviation from this model. If the other player's next move is inconsistent with its past behavior, this inconsistency can be due either to noise or to a genuine change in its behavior; and DBS can often distinguish between these two cases by waiting to see whether this inconsistency persists in the next few iterations of the game.[2]

Of the nine different version of DBS that we entered into the competition, all of them placed in the top 25, and seven of them placed among top ten (see Table 10.1). Our best version, DBSz, placed third; and the two players that placed higher were both masters of master-and-slave teams.

DBS operates in a distinctly different way from the master-and-slaves strategy used by several other entrants in the competition. Each participant in the competition was allowed to submit up to 20 programs as contestants. Some participants took advantage of this to submit collections of programs that worked together in a conspiracy in which 19 of their 20 programs (the "slaves") worked to give as many points as possible to the 20th program (the "master"). DBS does not use a master-and-slaves strategy, nor does it conspire with other programs in any other way. Nonetheless, DBS remained competitive with the master-and-slaves strategies in the competition, and performed much better than the master-and-slaves strategies if the score of each master is averaged with the scores of its slaves. Furthermore, a more extensive analysis [Au and Nau, 2005] shows that if each master-and-slaves team had been limited to 10 programs or less, DBS would have placed first in the competition.

10.2 Motivation and Approach

The techniques used in DBS are motivated by a British army officer's story that was quoted in [Axelrod, 1997, page 40]:

> I was having tea with A Company when we heard a lot of shouting and went out to investigate. We found our men and the Germans standing on their respective parapets. Suddenly a salvo arrived but did no damage. Naturally both sides got down and our men started swearing at the Germans, when all at once a brave German got onto his parapet and shouted out: "We are very sorry about that; we hope no one was hurt. It is not our fault. It is that

[2] An iteration has also been called a period or a round by some authors.

damned Prussian artillery." (Rutter 1934, 29)

Such an apology was an effective way of resolving the conflict and preventing a retaliation because it told the British that the salvo was not the intention of the German infantry, but instead was an unfortunate accident that the German infantry did not expect nor desire. The reason why the apology was convincing was because it was consistent with the German infantry's past behavior. The British had was ample evidence to believe that the German infantry wanted to keep the peace just as much as the British infantry did.

More generally, an important question for conflict prevention in noisy environments is *whether a misconduct is intentional or accidental.* A deviation from the usual course of action in a noisy environment can be explained in either way. If we form the wrong belief about which explanation is correct, our response may potentially destroy our long-term relationship with the other player. If we ground our belief on evidence accumulated before and after the incident, we should be in a better position to identify the true cause and prescribe an appropriate solution. To accomplish this, DBS uses the following key techniques:

(1) **Learning about the other player's strategy.** DBS uses an induction technique to identify a set of rules that model the other player's recent behavior. The rules give the probability that the player will cooperate under different situations. As DBS learns these probabilities during the game, it identifies a set of *deterministic* rules that have either 0 or 1 as the probability of cooperation.
(2) **Detecting noise**. DBS uses the above rules to detect anomalies that may be due either to noise or a genuine change in the other player's behavior. If a move is different from what the deterministic rules predict, this inconsistency triggers an *evidence collection process* that will monitor the persistence of the inconsistency in the next few iterations of the game. The purpose of the evidence-collection process is to determine whether the violation is likely to be due to noise or to a change in the other player's policy. If the inconsistency does not persist, DBS asserts that the derivation is due to noise; if the inconsistency persists, DBS assumes there is a change in the other player's behavior.
(3) **Temporarily tolerating possible misbehaviors by the other player.** Until the evidence-collection process finishes, DBS assumes that the other player's behavior is still as described by the deterministic rules. Once the evidence collection process has finished, DBS decides whether to believe the other player's behavior has changed, and

updates the deterministic rules accordingly.

Since DBS emphasizes the use of deterministic behaviors to distinguish noise from the change of the other player's behavior, it works well when the other player uses a pure (i.e., deterministic) strategy or a strategy that makes decisions deterministically most of the time. Fortunately, deterministic behaviors are abundant in the Iterated Prisoner's Dilemma. Many well-known strategies, such as TFT and GRIM, are pure strategies. Some strategies such as Pavlov or Win-Stay, Lose-Shift strategy (WSLS) [Kraines and Kraines, 1989; Kraines and Kraines, 1993; Kraines and Kraines, 1995; Nowak and Sigmund, 1993] are not pure strategies, but a large part of their behavior is still deterministic. The reason for the prevalence of determinism is discussed by Axelrod in [Axelrod, 1984]: clarity of behavior is an important ingredient of long-term cooperation. A strategy such as TFT benefits from its clarity of behavior, because it allows other players to make credible predictions of TFT's responses to their actions. We believe the success of our strategy in the competition is because this clarity of behavior also helps us to fend off noise.

The results of the competition show that the techniques used in DBS are indeed an effective way to fend off noise and maintain cooperation in noisy environments. When DBS defers judgment about whether the other player's behavior has changed, the potential cost is that DBS may not be able to respond to a genuine change of the other player's behavior as quickly as possible, thus losing a few points by not retaliating immediately. But this delay is only temporary, and after it DBS will adapt to the new behavior. More importantly, the techniques used in DBS greatly reduce the probability that noise will cause it to end a cooperation and fall into a mutual-defect situation. Our experience has been that it is hard to re-establish cooperation from a mutual-defection situation, so it is better avoid getting into mutual defection situations in the first place. When compared with the potential cost of ending an cooperation, the cost of temporarily tolerating some defections is worthwhile.

Temporary tolerance also benefits us in another way. In the noisy Iterated Prisoner's Dilemma, there are two types of noise: one that affects the other player's move, and the other affects our move. While our method effectively handles the first type of noise, it is the other player's job to deal with the second type of noise. Some players such as TFT are easily provoked by the second type of noise and retaliate immediately. Fortunately, if the retaliation is not a permanent one, our method will treat the retaliation

in the same way as the first type of noise, thus minimizing its effect.

10.3 Iterated Prisoner's Dilemma with Noise

In the Iterated Prisoner's Dilemma, two players play a finite sequence of classical prisoner's dilemma games, whose payoff matrix is:

		Player 2	
		Cooperate	Defect
Player 1	Cooperate	(u_{CC}, u_{CC})	(u_{CD}, u_{DC})
	Defect	(u_{DC}, u_{CD})	(u_{DD}, u_{DD})

where $u_{DC} > u_{CC} > u_{DD} > u_{CD}$ and $2u_{CC} > u_{DC} + u_{CD}$. In the competition, u_{DC}, u_{CC}, u_{DD} and u_{CD} are 5, 3, 1 and 0, respectively.

At the beginning of the game, each player knows nothing about the other player and does not know how many iterations it will play. In each iteration, each player chooses either to cooperate (C) or defect (D), and their payoffs in that iteration are as shown in the payoff matrix. We call this decision a *move* or an *action*. After both players choose a move, they will each be informed of the other player's move before the next iteration begins.

If $a_k, b_k \in \{C, D\}$ are the moves of Player 1 and Player 2 in iteration k, then we say that (a_k, b_k) is the *outcome* of iteration k. If there are N iterations in a game, then the total scores for Player 1 and Player 2 are $\sum_{1 \le k \le N} u_{a_k b_k}$ and $\sum_{1 \le k \le N} u_{b_k a_k}$, respectively.

The *Noisy Iterated Prisoner's Dilemma* is a variant of the Iterated Prisoner's Dilemma in which there is a small probability that a player's moves will be mis-implemented. The probability is called the *noise level*.[3] In other words, the noise level is the probability of executing C when D was the intended move, or vice versa. The incorrect move is recorded as the player's move, and determines the outcome of the iteration.[4] Furthermore, neither player has any way of knowing whether the other player's move was executed correctly or incorrectly.[5]

[3] The noise level in the competition was 0.1.

[4] Hence, a mis-implementation is different from a misperception, which would not change the outcome of the iteration. The competition included mis-implementations but no misperceptions.

[5] As far as we know, the definitions of "mis-implementation" used in the existing literature are ambiguous about whether either of the players should know that an action has been mis-executed.

For example, suppose Player 1 chooses C and Player 2 chooses D in iteration k, and noise occurs and affects the Player 1's move. Then the outcome of iteration k is (D, D). However, since both players do not know that the Player 1's move has been changed by noise, Player 1 and Player 2 perceive the outcome differently: for Player 1, the outcome is (C, D), but for Player 2, the outcome is (D, D). As in real life, this misunderstanding would become an obstacle in establishing and maintaining cooperation between the players.

10.4 Strategies, Policies, and Hypothesized Policies

A *history* H of length k is the sequence of outcomes of all iterations up to and including iteration k. We write $H = \langle (a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k) \rangle$. Let $\mathcal{H} = \langle (C, C), (C, D), (D, C), (D, D) \rangle^*$ be the set of all possible histories. A *strategy* $M : \mathcal{H} \to [0, 1]$ associates with each history H a real number called the *degree of cooperation*. $M(H)$ is the probability that M chooses to cooperate at iteration $k + 1$, where $k = |H|$ is H's length.

For examples, TFT can be considered as a function M_{TFT}, such that (1) $M_{TFT}(H) = 1.0$ if $k = 0$ or $a_k = C$ (where $k = |H|$), and (2) $M_{TFT}(H) = 0.0$ otherwise; Tit-for-Two-Tats (TFTT), which is like TFT except it defects only after it receives two consecutive defections, can be considered as a function M_{TFTT}, such that (1) $M_{TFTT}(H) = 0.0$ if $k \geq 2$ and $a_{k-1} = a_k = D$, and (2) $M_{TFTT}(H) = 1.0$ otherwise.

We can model a strategy as a *policy*. A *condition* $Cond : \mathcal{H} \to \{\mathsf{True}, \mathsf{False}\}$ is a mapping from histories to boolean values. A history H *satisfies* a condition $Cond$ if and only if $Cond(H) = \mathsf{True}$. A *policy schema* Ω is a set of conditions such that each history in $\mathcal{H}$ satisfies exactly one of the conditions in Ω. A *rule* is a pair $(Cond, p)$, which we will write as $Cond \to p$, where $Cond$ is a condition and p is a degree of cooperation (a real number in $[0, 1]$). A rule is *deterministic* if p is either 0.0 or 1.0; otherwise, the rule is *probabilistic*. In this paper, we define a policy to be a set of rules whose conditions constitute a policy schema.

M_{TFT} can be modeled as a policy as follows: we define $Cond_{a,b}$ to be a condition about the outcomes of the last iteration of a history, such that $Cond_{a,b}(H) = \mathsf{True}$ if and only if (1) $k \geq 1$, $a_k = a$ and $b_k = b$, (where $k = |H|$), or (2) $k = 0$ and $a = b = C$. For simplicity, we also write $Cond_{a,b}$ as (a, b). The policy for M_{TFT} is $\pi_{TFT} = \{(C, C) \to 1.0, (C, D) \to 1.0, (D, C) \to 0.0, (D, D) \to 0.0\}$. Notice that the policy schema for π_{TFT}

is $\Omega = \{(C, C), (C, D), (D, C), (D, D)\}$.

Given a policy π and a history H, there is one and only one rule $Cond \rightarrow p$ in π such that $Cond(H) = \mathsf{True}$. We write p as $\pi(H)$. A policy π is *complete* for a strategy M if and only if $\pi(H) = M(H)$ for any $H \in \mathcal{H}$. In other words, a complete policy for a strategy is one that completely models the strategy. For instance, π_{TFT} is a complete policy for M_{TFT}.

Some strategies are much more complicated than TFT—we need a large number of rules in order to completely model these strategies. If the number of iterations is small and the strategy is complicated enough, it is difficult or impossible for DBS to obtain a complete model of the other player's strategy. Therefore, DBS does not aim at obtaining a complete policy of the other player's strategy; instead, DBS leans an approximation of the other player's strategy during a game, using a small number of rules. In order to distinguish this approximation from the complete policies for a strategy, we call this approximation a *hypothesized policy*.

Given a policy schema Ω, DBS constructs a hypothesized policy π whose policy schema is Ω. The degrees of cooperation of the rules in π are estimated by a learning function (e.g., the learning methods in Section 10.6), which computes the degrees of cooperation according to the current history. For example, suppose the other player's strategy is M_{TFTT}, the given policy schema is $\Omega = \{(C, C), (C, D), (D, C), (D, D)\}$, and the current history is $H = \{(C, C), (D, C), (C, C), (D, C), (D, C), (D, D), (C, D), (C, C)\}$. If we use a learning method which computes the degrees of cooperation by averaging the number of time the next action is C when a condition holds, then the hypothesized policy is $\pi = \{(C, C) \rightarrow 1.0, (C, D) \rightarrow 1.0, (D, C) \rightarrow 0.66, (D, D) \rightarrow 0.0\}$. Notice that the rule $(D, C) \rightarrow 0.66$ does not accurately model M_{TFTT}; this probabilistic rule is just an approximation of what M_{TFTT} does when the condition (D, C) holds. This approximation is inaccurate as long as the policy schema contains (D, C)—there is no complete policy for M_{TFTT} whose policy schema contains (D, C). If we want to model M_{TFTT} correctly, we need a different policy schema that allows us to specify more complicated rules.

We interpret a hypothesized policy as a belief of what the other player will do in the next few iterations in response to our next few moves. This belief does not necessarily hold in the long run, since the other player can behave differently at different time in a game. Even worse, there is no guarantee that this belief is true in the next few iterations. Nonetheless, hypothesized policies constructed by DBS usually have a high degree of accuracy in predicting what the other player will do.

This belief is subjective—it depends on the choice of the policy schema and the learning function. We formally define this subjective viewpoint as follows. The *hypothesized policy space* spanned by a policy schema Ω and a learning function $L : \Omega \times \mathcal{H} \to [0, 1]$ is a set of policies $\Pi = \{\pi(H) : H \in \mathcal{H}\}$, where $\pi(H) = \{Cond \to L(Cond, H) : Cond \in \Omega\}$. Let H be a history of a game in which the other player's strategy is M. The set of all possible hypothesized policies for M in this game is $\{\pi(H_k) : H_k \in \mathsf{prefixes}(H)\} \subseteq \Pi$, where $\mathsf{prefixes}(H)$ is the set of all prefixes of H, and H_k is the prefix of length k of H. We say $\pi(H_k)$ is the *current* hypothesized policy of M in the iteration k. A rule $Cond \to p$ in $\pi(H_k)$ describes a particular *behavior* of the other player's strategy in the iteration k. The behavior is *deterministic* if p is either zero or one; otherwise, the behavior is *random* or *probabilistic*. If $\pi(H_k) \neq \pi(H_{k+1})$, we say there is a change of the hypothesized policy in the iteration $k + 1$, and the behaviors described by the rules in $(\pi(H_k) \setminus \pi(H_{k+1}))$ have changed.

10.5 Derived Belief Strategy

In the ordinary Iterated Prisoner's Dilemma (i.e., without any noise), if we know the other player's strategy and how many iterations in a game, we can compute an optimal strategy against the other player by trying every possible sequence of moves to see which sequence yields the highest score, assuming we have sufficient computational power. However, we are missing both pieces of information. So it is impossible for us to compute an optimal strategy, even with sufficient computing resource. Therefore, we can at most predict the other player's moves based on the history of a game, subject to the fact that the game may terminate any time.

Some strategies for the Iterated Prisoner's Dilemma do not predict the other player's moves at all. For example, Tit-for-Tat and GRIM react deterministically to the other player's previous moves according to fixed sets of rules, no matter how the other player actually plays. Many strategies adapt to the other player's strategy over the course of the game: for example, Pavlov [Kraines and Kraines, 1989] adjusts its degree of cooperation according to the history of a game. However, these strategies do not take any prior information about the other player's strategy as an input; thus they are unable to make use of this important piece of information even when it is available.

Let us consider a class of strategies that make use of a model of the other

player's strategy to make decisions. Figure 10.1 shows an abstract representation of these strategies. Initially, these strategies start out by assuming that the other player's strategy is TFT or some other strategy. In every iteration of the game, the model is updated according to the current history (using UpdateModel). These strategies decide which move it should make in each iteration using a move generator (GenerateMove), which depends on the current model of the other player's strategy of the iteration.

```
Procedure StrategyUsingModelOfTheOtherPlayer()
    π ← InitialModel()            // the current model of the other player
    H ← ∅                         // the current history
    a ← GenerateMove(π, H)        // the initial move

    Loop until the end of the game
        Output our move a and obtain the other player's move b
        H ← ⟨H, (a, b)⟩
        π ← UpdateModel(π, H)
        a ← GenerateMove(π, H)
    End Loop
```

Fig. 10.1 An abstract representation of a class of strategies that generate moves using a model of the other player.

DBS belongs to this class of strategies. DBS maintains a model of the other player in form of a hypothesized policy throughout a game, and makes decisions based on this hypothesized policy. The key issue for DBS in this process is how to maintain a good approximation of the other player's strategy, despite that some actions in the history are affected by noise. A good approximation will increase the quality of moves generated by DBS, since the move generator in DBS depends on an accurate model of the other player's behavior.

The approach DBS uses to minimize the effect of noise on the hypothesized policy has been discussed in Section 10.2: temporarily tolerate possible misbehaviors by the other player, and then update the hypothesized policy only if DBS believes that the misbehavior is due to a genuine change of behaviors. Figure 10.2 shows an outline of the implementation of this approach in DBS. As we can see, DBS does not maintain the hypothesized policy explicitly; instead, DBS maintains three sets of rules: the default rule set (R_d), the current rule set (R_c), and the probabilistic rule set (R_p). DBS combines these rule sets to form a hypothesized policy for move gen-

eration. In addition, DBS maintains several auxiliary variables (promotion counts and violation counts) to facilitate the update of these rule sets. We will explain every line in Figure 10.2 in detail in the next section.

Procedure **DerivedBeliefStrategy**()
1. $R_d \leftarrow \pi_{TFT}$ // *the default rule set*
2. $R_c \leftarrow \emptyset$ // *the current rule set*
3. $a_0 \leftarrow C$; $b_0 \leftarrow C$; $H \leftarrow \langle (a_0, b_0) \rangle$; $\pi = R_d$; $k \leftarrow 1$; $v \leftarrow 0$
4. $a_1 \leftarrow \mathsf{MoveGen}(\pi, H)$
5. Loop until the end of the game
6. Output a_k and obtain the other player's move b_k
7. $r^+ \leftarrow ((a_{k-1}, b_{k-1}) \rightarrow b_k)$
8. $r^- \leftarrow ((a_{k-1}, b_{k-1}) \rightarrow (\{C, D\} \setminus \{b_k\}))$
9. If $r^+, r^- \notin R_c$, then
10. If $\mathsf{ShouldPromote}(r^+) = \mathsf{true}$, then insert r^+ into R_c.
11. If $r^+ \in R_c$, then set the violation count of r^+ to zero
12. If $r^- \in R_c$ and $\mathsf{ShouldDemote}(r^-) = \mathsf{true}$, then
13. $R_d \leftarrow R_c \cup R_d$; $R_c \leftarrow \emptyset$; $v \leftarrow 0$
14. If $r^- \in R_d$, then $v \leftarrow v + 1$
15. If $v > RejectThreshold$, or $(r^+ \in R_c$ and $r^- \in R_d)$, then
16. $R_d \leftarrow \emptyset$; $v \leftarrow 0$
17. $R_p \leftarrow \{(Cond \rightarrow p') \in \psi_{k+1} : Cond$ not appear in R_c or $R_d\}$
18. $\pi \leftarrow R_c \cup R_d \cup R_p$ // *construct a hypothesized policy*
19. $H \leftarrow \langle H, (a_k, b_k) \rangle$; $a_{k+1} \leftarrow \mathsf{MoveGen}(\pi, H)$; $k \leftarrow k + 1$
20. End Loop

Fig. 10.2 An outline of the DBS strategy. ShouldPromote first increases r^+'s promotion count, and then if r^+'s promotion count exceeds the promotion threshold, ShouldPromote returns true and resets r^+'s promotion count. Likewise, ShouldDemote first increases r^-'s violation count, and then if r^-'s violation count exceeds the violation threshold, ShouldPromote returns true and resets r^-'s violation count. R_p in Line 17 is the probabilistic rule set; ψ_{k+1} in Line 17 is calculated from Equation 10.1.

10.6 Learning Hypothesized Policies in Noisy Environments

We will describe how DBS learns and maintains a hypothesized policy for the other player's strategy in this section. Section 10.6.1 describes how DBS uses *discounted frequencies* for each behavior to estimate the degree of

cooperation of each rule in the hypothesized policy. Section 10.6.2 explains why using discounted frequencies alone are not sufficient for constructing an accurate model of the other player's strategy in the presence of noise, and how symbolic noise detection and temporary tolerance can help overcome the difficulty in using discounted frequencies alone. Section 10.6.3 presents the induction technique DBS uses to identify deterministic behaviors in the other player. Section 10.6.4 illustrates how DBS defers judgment about whether an anomaly is due to noise. Section 10.6.5 discusses how DBS updates the hypothesized policy when it detects a change of behavior.

10.6.1 *Learning by Discounted Frequencies*

We now describe a simple way to estimate the degree of cooperation of the rules in the hypothesized policy. The idea is to maintain a *discounted frequency* for each behavior: instead of keeping an ordinary frequency count of how often the other player cooperates under a condition in the past, DBS applies discount factors based on how recent each occurrence of the behavior was.

Given a history $H = \{(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\}$, a real number α between 0 and 1 (called the *discount factor*), and an initial hypothesized policy $\pi_0 = \{Cond_1 \rightarrow p_1^0, Cond_2 \rightarrow p_2^0, \ldots, Cond_n \rightarrow p_n^0\}$ whose policy schema is $\mathcal{C} = \{Cond_1, Cond_2, \ldots, Cond_n\}$, the *probabilistic policy* at iteration $k+1$ is $\psi_{k+1} = \{Cond_1 \rightarrow p_1^{k+1}, Cond_2 \rightarrow p_2^{k+1}, Cond_n \rightarrow p_n^{k+1}\}$, where p_i^{k+1} is computed by the following equation:

$$p_i^{k+1} = \frac{\sum_{0 \le j \le k} \left(\alpha^{k-j} g_j\right)}{\sum_{0 \le j \le k} \left(\alpha^{k-j} f_j\right)} \tag{10.1}$$

and where

$$g_j = \begin{cases} p_i^0 \text{ if } j = 0, \\ 1 \quad \text{if } 1 \le j \le k, \, Cond_i(H_{j-1}) = \mathsf{True} \text{ and } b_j = C, \\ 0 \quad \text{otherwise;} \end{cases}$$

$$f_j = \begin{cases} p_i^0 \text{ if } j = 0, \\ 1 \quad \text{if } 1 \le j \le k, \, Cond_i(H_{j-1}) = \mathsf{True}, \\ 0 \quad \text{otherwise;} \end{cases}$$

$$H_{j-1} = \begin{cases} \emptyset & \text{if } j = 1, \\ \{(a_1, b_1), (a_2, b_2), \ldots, (a_{j-1}, b_{j-1})\} & \text{otherwise.} \end{cases}$$

In short, the current history H has $k+1$ possible prefixes, and f_j is basically

a boolean function indicating whether the prefix of H up to the $j-1$'th iteration satisfies $Cond_i$. g_j is a restricted version of f_j.

When $\alpha = 1$, p_i is approximately equal to the frequency of the occurrence of $Cond_i \rightarrow p_i$. When α is less than 1, p_i becomes a weighted sum of the frequencies that gives more weight to recent events than earlier ones. For our purposes, it is important to use $\alpha < 1$, because it may happen that the other player changes its behavior suddenly, and therefore we should forget about its past behavior and adapt to its new behavior (for instance, when GRIM is triggered). In the competition, we used $\alpha = 0.75$.

An important question is how large a policy schema to use for the hypothesized policy. If the policy schema is too small, the policy schema won't provide enough detail to give useful predictions of the other player's behavior. But if the policy schema is too large, DBS will be unable to compute an accurate approximation of each rule's degree of cooperation, because the number of iterations in the game will be too small. In the competition, we used a policy schema of size 4: $\{(C,C),(C,D),(D,C),(D,D)\}$. We have found this to be good enough for modeling a large number of strategies.

It is essential to have a good initial hypothesized strategy because at the beginning of the game the history is not long enough for us to derive any meaningful information about the other player's strategy. In the competition, the initial hypothesized policy is $\pi_{TFT} = \{(C,C) \rightarrow 1.0, (C,D) \rightarrow 1.0, (D,C) \rightarrow 0.0, (D,D) \rightarrow 0.0\}$.

10.6.2 *Deficiencies of Discounted Frequencies in Noisy Environments*

It may appear that the probabilistic policy learned by the discounted-frequency learning technique should be inherently capable of tolerating noise, because it takes many, if not all, moves in the history into account: if the number of terms in the calculation of the average or weighted average is large enough, the effect of noise should be small. However, there is a problem with this reasoning: it neglects the effect of multiple occurrences of noise within a small time interval.

A mis-implementation that alters the move of one player would distort an established pattern of behavior observed by the other player. The general effect of such distortion to the Equation 10.1 is hard to tell—it varies with the value of the parameters and the history. But if several distortions occur within a small time interval, the distortion may be big enough to alter the probabilistic policy and hence change our decision about what move

to make. This change of decision may potentially destroy an established pattern of mutual cooperation between the players.

At first glance, it might seem rare for several noise events to occur at nearly the same time. But if the game is long enough, the probability of it happening can be quite high. The probability of getting two noise events in two consecutive iterations out of a sequence of i iterations can be computed recursively as $X_i = p(p + qX_{i-2}) + qX_{i-1}$, providing that $X_0 = X_1 = 0$, where p is the probability of a noise event and $q = 1-p$. In the competition, the noise level was $p = 0.1$ and $i = 200$, which gives $X_{200} = 0.84$. Similarly, the probabilities of getting three and four noises in consecutive iterations are 0.16 and 0.018, respectively.

In the 2005 competition, there were 165 players, and each player played each of the other players five times. This means every player played 825 games. On average, there were 693 games having two noises in two consecutive iterations, 132 games having three noises in three consecutive iterations, and 15 games having four noises in four consecutive iterations. Clearly, we did not want to ignore situations in which several noises occur nearly at the same time.

Symbolic noise detection and temporary tolerance outlined in Section 10.2 provide a way to reduce the amount of susceptibility to multiple occurrences of noise in a small time interval. Deterministic rules enable DBS to detect anomalies in the observed behavior of the other player. DBS temporarily ignores the anomalies which may or may not be due to noise, until a better conclusion about the cause of the anomalies can be drawn. This temporary tolerance prevents DBS from learning from the moves that may be affected by noise, and hence protects the hypothesized policy from the influence of errors due to noise. Since the amount of tolerance (and the accuracy of noise detection) can be controlled by adjusting parameters in DBS, we can reduce the amount of susceptibility to multiple occurrences of noise by increasing the amount of tolerance, at the expense of a higher cost of noise detection—losing more points when a change of behavior occurs.

10.6.3 *Identifying Deterministic Rules Using Induction*

As we discussed in Section 10.2, deterministic behaviors are abundant in the Iterated Prisoner's Dilemma. Deterministic behaviors can be modeled by deterministic rules, whereas random behavior would require probabilistic rules.

A nice feature about deterministic rules is that they have only two

possible degrees of cooperation: zero or one, as opposed to an infinite set of possible degrees of cooperation of the probabilistic rules. Therefore, there should be ways to learn deterministic rules that are much faster than the discounted frequency method described earlier. For example, if we knew at the outset which rules were deterministic, it would take only one occurrence to learn each of them: each time the condition of a deterministic rule was satisfied, we could assign a degree of cooperation of 1 or 0 depending on whether the player's move was C or D.

The trick, of course, is to determine which rules are deterministic. We have developed an inductive-reasoning method to distinguish deterministic rules from probabilistic rules during learning and to learn the correct degree of cooperation for the deterministic rules.

In general, induction is the process of deriving general principles from particular facts or instances. To learn deterministic rules, the idea of induction can be used as follows. If a certain kind of behavior occurs repeatedly several times, and during this period of time there is no other behavior that contradicts to this kind of behavior, then we will hypothesize that the chance of the same kind of behavior occurring in the next few iterations is pretty high, regardless of how the other player behaved in the remote past.

More precisely, let $K \geq 1$ be a number which we will call the *promotion threshold*. Let $H = \langle (a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k) \rangle$ be the current history. For each condition $Cond_j \in \mathcal{C}$, let I_j be the set of indexes such that for all $i \in I_j$, $i < k$ and $Cond_j(\langle (a_1, b_1), (a_2, b_2), \ldots, (a_i, b_i) \rangle) = \mathsf{True}$. Let $\hat{I}_j$ be the set of the largest K indexes in I_j. If $|I_j| \geq K$ and for all $i \in \hat{I}_j$, $b_{i+1} = C$ (i.e., the other player chose C when the previous history up to the i'th iteration satisfies $Cond_j$), then we will hypothesize that the other player will choose C whenever $Cond_j$ is satisfied; hence we will use $Cond_j \rightarrow 1$ as a deterministic rule. Likewise, if $|I_j| \geq K$ and for all $i \in \hat{I}_j$, $b_{i+1} = D$, we will use $Cond_j \rightarrow 0$ as a deterministic rule. See Line 7 to Line 10 in Figure 10.2 for an outline of the induction method we use in DBS.

The induction method can be faster at learning deterministic rules than the discounted frequency method that regards a rule as deterministic when the degree of cooperation estimated by discounted frequencies is above or below certain thresholds. As can be seen in Figure 10.3, the induction method takes only three iterations to infer the other player's moves correctly, whereas the discounted frequency technique takes six iterations to obtain a 95% degree of cooperation, and it never becomes 100%.[6] We may

[6]If we modify Equation 10.1 to discard the early outcomes of a game, the degree of cooperation of a probabilistic rule can attain 100%.

want to set the threshold in the discounted frequency method to be less than 0.8 to make it faster than the induction method. However, this will increase the chance of incorrectly identifying a random behavior as deterministic.

A faster learning speed allows us to infer deterministic rules with a shorter history, and hence increase the effectiveness of symbolic noise detection by having more deterministic rules at any time, especially when a change of the other player's behavior occurs. The promotion threshold K controls the speed of the identification of deterministic rules. The larger the value of K, the slower the speed of identification, but the less likely we will mistakenly hypothesize that the other player's behavior is deterministic.

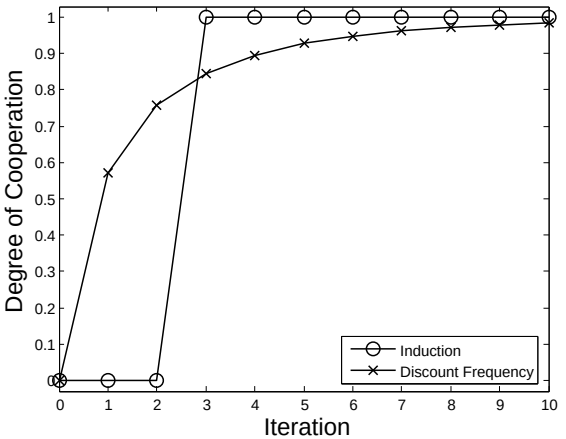

Fig. 10.3 Learning speeds of the induction method and the discounted frequency method when the other player always cooperates. The initial degree of cooperation is zero, the discounted rate is 0.75, and the promotion threshold is 3.

10.6.4 *Symbolic Noise Detection and Temporary Tolerance*

Once DBS has identified the set of deterministic rules, it can readily use them to detect noise. As we said earlier, if the other player's move violate a deterministic rule, it can be caused either by noise or by a change in the other player's behavior, and DBS uses an evidence collection process to figure out which is the case. More precisely, once a deterministic rule $Cond_i \rightarrow o_i$ is violated (i.e., the history up to the previous iteration satisfies $Cond_i$ but the other player's move in the current iteration is different from o_i), DBS keeps the violated rule but marks it as violated. Then DBS

starts an *evidence collection process* that in the implementation of our competition entries is a violation counting: for each violated probabilistic rule DBS maintains a counter called the *violation count* to record how many violations of the rule have occurred (Line 12).[7] In the subsequent iterations, DBS increases the violation count by one every time a violation of the rule occurs. However, if DBS encounters a positive example of the rule, DBS resets the violation count to zero and unmark the rule (Line 11). If any violation count excesses a threshold called the *violation threshold*, DBS concludes that the violation is not due to noise; it is due to a change of the other player's behavior. In this case, DBS invokes a special procedure (described in Section 10.6.5) to handle this situation (Line 13).

This evidence collection process takes advantages of the fact that the pattern of moves affected by noise is often quite different from the pattern of moves generated by the new behavior after a change of behavior occurs. Therefore, it can often distinguish noise from a change of behavior by observing moves in the next few iterations and gather enough evidence.

As discussed in Section 10.6.2, we want to set a larger violation threshold in order to avoid the drawback of the discount frequency method in dealing with several misinterpretations caused by noise within a small time interval. However, if the threshold is too large, it will slow down the speed of adaptation to changes in the other player's behavior. In the competition, we entered DBS several times with several different violation thresholds; and in the one that performed the best, the violation threshold was 4.

10.6.5 *Coping with Ignorance of the Other Player's New Behavior*

When the evidence collection process detects a change in the other player's behavior, DBS knows little about the other player's new behavior. How DBS copes with this ignorance is critical to its success.

When DBS knows little about the other player's new behavior when it detects a change of the other player's behavior, DBS temporarily uses the previous hypothesized policy as the current hypothesized policy, until it deems that this substitution no longer works. More precisely, DBS maintains two sets of deterministic rules: the *current rule set* R_c and the *default rule set* R_d. R_c is the set of deterministic rules that is learned after the change of behavior occurs, while R_d is the set of deterministic rules

[7] We believe that a better evidence collection process should be based on statistical hypothesis testing.

before the change of behavior occurs. At the beginning of a game, R_d is π_{TFT} and R_c is an empty set (Line 1 and Line 2). When DBS constructs a hypothesized policy π for move generation, it uses every rule in R_c and R_d. In addition, for any missing rule (i.e., the rule those condition are different from any rule's condition in R_c or R_d), we regard it as a probabilistic rule and approximate its degree of cooperation by Equation 10.1 (Line 17). These probabilistic rules form the probabilistic rule set $R_p \subseteq \psi_{k+1}$.

While DBS can insert any newly found deterministic rule in R_c, it insert rules into R_d *only when* the evidence collection process detects a change of the other player's behavior. When it happens, DBS copies all the rules in R_c to R_d, and then set R_c to an empty set (Line 13).

The default rule set is designed to be *rejected*: we maintain a violation count to record the number of violations to any rule in R_d. Every time any rule in R_d is violated, the violation count increased by 1 (Line 14). Once the violation count exceeds a *rejection threshold*, we drop the default rule set entirely (set it to an empty set) and reset the violation count (Line 15 and Line 16). We also reject R_d whenever any rule in R_c contradicts any rule in R_d (Line 15).

We preserve the rules in R_c mainly for sake of providing a smooth transition: we don't want to convert all deterministic rules to probabilistic rules at once, as it might suddenly alter the course of our moves, since the move generator in DBS generates moves according to the current hypothesized policy only. This sudden change in DBS's behavior can potentially disrupt the cooperative relationship with the other player. Furthermore, some of the rules in R_c may still hold, and we don't want to learn them from scratch.

Notice that symbolic noise detection and temporary tolerance makes use of the rules in R_c but not the rules in R_d, although DBS makes use of the rules in both R_c and R_d when DBS decides the next move (Line 18). We do not use R_d for symbolic noise detection and temporary tolerance because when DBS inserts rules into R_d, a change of the other player's behavior has already occurred—there is little reason to believe that anomalies detected using the rules in R_d are due to noise. Furthermore, we want to turn off symbolic noise detection and temporary tolerance temporarily when a change of behavior occurs, in order to identify a whole new set of deterministic rules from scratch.

10.7 The Move Generator in DBS

We devised a simple and reasonably effective move generator for DBS. As shown in Figure 10.1, the move generator takes the current hypothesized policy π and the current history $H_{current}$ whose length is $l = |H_{current}|$, and then decides whether DBS should cooperate in the current iteration. It is difficult to devise a good move generator, because our move could lead to a change of the hypothesized policy and complicate our projection of the long-term payoff. Perhaps, the move generator should take the other player's model of DBS into account [Carmel and Markovitch, 1994]. However, we found that by making the assumption that hypothesized policy will not change for the rest of the game, we can devise a simple move generator that generates fairly good moves. The idea is that we compute the *maximum expected score* we can possibly earn for the rest of the game, using a technique that combines some ideas from both game-tree search and Markov Decision Processes (MDPs). Then we choose the first move in the set of moves that leads to this maximum expected score as our move for the current iteration.

To accomplish the above, we consider all possible histories whose prefix is $H_{current}$ as a tree. In this tree, each path starting from the root represents a possible history, which is a sequence of past outcomes in $H_{current}$ plus a sequence of possible outcomes in future iterations. Each node on a path represents the outcome of an iteration of a history. Figure 10.4 shows an example of such a tree. The root node of the tree represents the outcome of the first iteration.

Let $outcome(S)$ be the outcome represented by a node S. Let $\langle S_0, S_1, \ldots, S_k \rangle$ be a sequence of nodes on the path from the root S_0 to S_k. We define the *depth* of S_k to be $k - l$, and the *history* of S_k be $H(S_k) = \langle outcome(S_1), outcome(S_2), \ldots, outcome(S_k) \rangle$. S_i is called the *current node* if the depth of S_i is zero; the current node represents the outcome of the last iteration and $H(S_i) = H_{current}$. As we do not know when the game will end, we assume *it will go for N^* more iterations*; thus each path in the tree has length of at most $l + N^*$.

Our objective is to compute a non-negative real number called the *maximum expected score* $E(S)$ for each node S with a non-negative depth. Like a conventional game tree search in computer chess or checkers, the maximum expected scores are defined recursively: the maximum expected score of a node at depth i is determined by the maximum expected scores of its children nodes at depth $i + 1$. The maximum expected score of a node S

of depth N^* is assumed to be the value computed by an *evaluation function* f. This is a mapping from histories to non-negative real numbers, such that $E(S) = f(H(S))$. The maximum expected score of a node S of depth k, where $0 \leq k < N^*$, is computed by the *maximizing rule*: suppose the four possible nodes after S are S_{CC}, S_{CD}, S_{DC}, and S_{DD}, and let p be the degree of cooperation predicted by the current hypothesized policy π (i.e., p is the right-hand side of a rule $(Cond \rightarrow p)$ in π such that $H(S)$ satisfies the condition $Cond$). Then $E(S) = \max\{E_C(S), E_D(S)\}$, where $E_C(S) = p(u_{CC} + E(S_{CC})) + (1-p)(u_{CD} + E(S_{CD}))$ and $E_D(S) = p(u_{DC}+E(S_{DC}))+(1-p)(u_{DD}+E(S_{DD}))$. Furthermore, we let $move(S)$ be the decision made by the maximizing rule at each node S, i.e., $move(S) = C$ if $E_C(S) \geq E_D(S)$ and $move(S) = D$ otherwise. By applying this maximizing rule recursively, we obtain the maximum expected score of every node with a non-negative depth. The move that we choose for the current iteration is $move(S_i)$, where S_i is the current node.

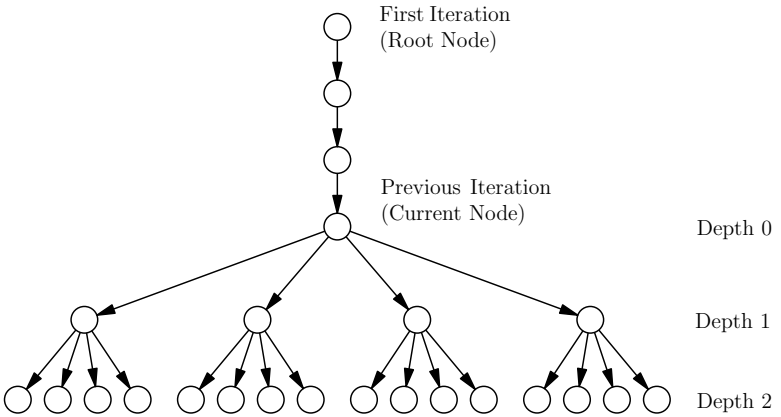

Fig. 10.4 An example of the tree that we use to compute the maximum expected scores. Each node denotes the outcome of an iteration. The top four nodes constitute a path representing the current history $H_{current}$. The length of $H_{current}$ is $l = 2$, and the maximum depth N^* is 2. There are four edges emanating from each node S after the current node; each of these edges corresponds to a possible outcome of the iteration after S. The maximum expected scores (not shown) of the nodes with depth 2 are set by an evaluation function f; these values are then used to calculate the maximum expected scores of the nodes with depth 1 by using the maximizing rule. Similarly, the maximum expected scores of the current node is calculated using four maximum expected scores of the nodes with depth 1.

The number of nodes in the tree increases exponentially with N^*. Thus,

the tree can be huge—there are over a billion nodes when $N^* \geq 15$. It is infeasible to compute the maximum expected score for every node one by one. Fortunately, we can use dynamic programming to speed up the computation. As an example, suppose the hypothesized policy is $\pi = \{(C,C) \to p_{CC}, (C,D) \to p_{CD}, (D,C) \to p_{DC}, (D,D) \to p_{DD}\}$, and suppose the evaluation function f returns a constant $f_{o_1 o_2}$ for any history that satisfies the condition (o_1, o_2), where $o_1, o_2 \in \{C, D\}$. Then, given our assumption that the hypothesized policy does not change, it is not hard to show by induction that all nodes whose histories have the same length and satisfy the same condition have the same maximum expected score. By using this property, we construct a table of size $4 \times (N^* + 2)$ in which each entry, denoted by $E^k_{o_1 o_2}$, stores the maximum expected score of the nodes whose histories have length $l + k$ and satisfy the condition (o_1, o_2), where $o_1, o_2 \in \{C, D\}$. We also have another table of the same size to record the decisions the procedure makes; the entry $m^k_{o_1 o_2}$ of this table is the decision being made at $E^k_{o_1 o_2}$. Initially, we set $E^{N+1}_{CC} = f_{CC}$, $E^{N+1}_{CD} = f_{CD}$, $E^{N+1}_{DC} = f_{DC}$, and $E^{N+1}_{DD} = f_{DD}$. Then the maximum expected scores in the remaining entries can be computed by the following recursive equation:

$$E^k_{o_1 o_2} = \max \big(p_{o_1 o_2}(u_{CC} + E^{k+1}_{CC}) + (1 - p_{o_1 o_2})(u_{CD} + E^{k+1}_{CD}), \\ p_{o_1 o_2}(u_{DC} + E^{k+1}_{DC}) + (1 - p_{o_1 o_2})(u_{DD} + E^{k+1}_{DD})\big),$$

where $o_1, o_2 \in \{C, D\}$. Similarly, $m^k_{o_1 o_2} = C$ if $(p_{o_1 o_2}(u_{CC} + E^{k+1}_{CC}) + (1 - p_{o_1 o_2})(u_{CD} + E^{k+1}_{CD})) \geq (p_{o_1 o_2}(u_{DC} + E^{k+1}_{DC}) + (1 - p_{o_1 o_2})(u_{DD} + E^{k+1}_{DD})$ and $m^k_{o_1 o_2} = D$ otherwise. If the outcome of the previous iteration is (o_1, o_2), we pick $m^0_{o_1 o_2}$ as the move for the current iteration. The pseudocode of this dynamic programming algorithm is shown in Figure 10.5.

10.8 Competition Results

The 2005 IPD Competition was actually a set of four competitions, each for a different version of the IPD. The one for the Noisy IPD was Category 2, which used a noise level of 0.1.

Of the 165 programs entered into the competition, eight of them were provided by the organizer of the competition. These programs included ALLC (always cooperates), ALLD (always defects), GRIM (cooperates until the first defection of the other player, and thereafter it always defects), NEG (cooperate (or defect) if the other player defects (or cooperates) in

Procedure MoveGen(π, H)

$\langle p_{CC}, p_{CD}, p_{DC}, p_{DD} \rangle \leftarrow \pi$

$\{(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\} \leftarrow H$

$(a_0, b_0) \leftarrow (C, C)$; $(a, b) \leftarrow (a_k, b_k)$

$\langle E_{CC}^{N^*+1}, E_{CD}^{N^*+1}, E_{DC}^{N^*+1}, E_{DD}^{N^*+1} \rangle \leftarrow \langle f_{CC}, f_{CD}, f_{DC}, f_{DD} \rangle$

For $k = N^*$ down to 0

For each (o_1, o_2) in $\{(C, C), (C, D), (D, C), (D, D)\}$

$F_{o_1 o_2}^k \leftarrow p_{o_1 o_2}(u_{CC} + E_{CC}^{k+1}) + (1 - p_{o_1 o_2})(u_{CD} + E_{CD}^{k+1})$

$G_{o_1 o_2}^k \leftarrow p_{o_1 o_2}(u_{DC} + E_{DC}^{k+1}) + (1 - p_{o_1 o_2})(u_{DD} + E_{DD}^{k+1})$

$E_{o_1 o_2}^k \leftarrow \max(F_{o_1 o_2}^k, G_{o_1 o_2}^k)$

If $F_{o_1 o_2}^k \geq G_{o_1 o_2}^k$, then $m_{o_1 o_2}^k \leftarrow C$

If $F_{o_1 o_2}^k < G_{o_1 o_2}^k$, then $m_{o_1 o_2}^k \leftarrow D$

End For

End For

Return m_{ab}^0

Fig. 10.5 The procedure for computing a recommended move for the current iteration. In the competition, we set $N^* = 60$, $f_{CC} = 3$, $f_{CD} = 0$, $f_{DC} = 5$, and $f_{DD} = 1$.

the previous iteration), RAND (defects or cooperates with the 1/2 probability), STFT (suspicious TFT, which is like TFT except it defects in the first iteration) TFT, and TFTT. All of these strategies are well known in the literature on IPD.

The remaining 157 programs were submitted by 36 different participants. Each participant was allowed to submit up to 20 programs. We submitted the following 20:

- **DBS.** We entered nine different versions of DBS into the competition, each with a different set of parameters or different implementation. The one that performed best was DBSz, which makes use of the exact set of features we mentioned in this chapter. Versions that have fewer features or additional features did not do as well.
- **Learning of Opponent's Strategy with Forgiveness (LSF).** Like DBS, LSF is a strategy that learns the other player's strategy during the game. The difference between LSF and DBS is that LSF does not make use of symbolic noise detection. It uses the discount frequency (Equation 10.1) to learn the other player's strategy, plus a forgiveness strategy that decides when to cooperate if mutual defection occurs. We entered one instance of LSF. It placed around the 30'th in three of the

runs and around 70'th in the other two runs. We believe the poor ranking of LSF is due to the deficiency of using discount frequency alone as we discussed at the beginning of Section 10.6.

- **Tit-for-Tat Improved (TFTI).** TFTI is a strategy based on a totally different philosophy from DBS's. It is not an opponent-modeling strategy, in the sense that it does not model the other player's behavior using a set of rules. Instead, it is a variant of TFT with a sophisticated forgiveness policy that aims at overcoming some of the deficiencies of TFT in noisy environments. We entered ten instantiations of TFTI in the competition, each with a different set of parameters or some differences in the implementation. The best of these, TFTIm, did well in the competition (see Table 10.1), but not as well as DBS.

Three of the other participants each entered the full complement of twenty programs: Wolfgang Kienreich, Jia-wei Li, and Perukrishnen Vytelingum. All three of them appear to have adopted the *master-and-slaves strategy* that was first proposed by Vytelingum's team from the University of Southampton. A master-and-slaves strategy is not a strategy for a single program, but instead for a team of collaborating programs. One of the programs in such a team is the *master*, and the remaining programs are *slaves*. The basic idea is that at the start of a run, the master and slaves would each make a series of moves using a predefined protocol, in order to identify themselves to each other. From then on, the master program would always play "defect" when playing with the slaves, and the slave programs would always play "cooperate" when playing with the master, so that the master would gain the highest possible payoff at each iteration. Furthermore, a slave would alway plays "defect" when playing with a program other than the master, in order to try to minimize that player's score.

Wolfgang Kienreich's master program was CNGF (CosaNostra Godfather), and its slaves were 19 copies of CNHM (CosaNostra Hitman). Jia-wei Li's master program was IMM01 (Intelligent Machine Master 01), and its slaves were IMS02, IMS03, ..., IMS20 (Intelligent Machine Slave n, for $n = 02, 03, \ldots 20$). Perukrishnen Vytelingum's master program was BWIN (S2Agent1_ZEUS), and its slaves were BLOS2, BLOS3, ..., BLOS20 (like BWIN, these programs also had longer names based on the names of ancient Greek gods).

We do not know what strategies the other participants used in their programs.

10.8.1 *Overall Average Scores*

Category 2 (IPD with noise) consisted of five runs. Each run was a round-robin tournament in which each program played with every program, including itself. Each program participated in 166 games in each run (recall that there is one game in which a player plays against itself, which counts as two games for that player). Each game consisted of 200 iterations. A program's score for a game is the sum of its payoffs over all 200 iterations (note that this sum will be at least 0 and at most 1000). The program's total score for an entire run is the sum of its scores over all 166 games. On the competition's website, there is a ranking for each of the five runs, each program is ranked according to its total score for the run.

A program's average score within a run is its total score for the run divided by 166. The program's overall average score is its average over all five runs, i.e., its total over all five runs divided by $830 = 5 \times 166$.

The table in Table 10.1 shows the average scores in each of the five runs of the top twenty-five programs when the programs are ranked by their overall average scores. Of our nine different versions of DBS, all nine of them are among the top twenty-five programs, and they dominate the top ten places. This phenomenon implies that DBS's performance is insensitive to the parameters in the programs and the implementation details of an individual program. The same phenomenon happens to TFTI—nine out of ten programs using TFTI are ranked between the 11th place and the 25th place, and the last one is at the 29th place.

10.8.2 *DBS versus the Master-and-Slaves Strategies*

Recall from Table 10.1: that DBSz placed third in the competition: it lost only to BWIN and IMM01, the masters of two master-and-slaves strategies. DBS does not use a master-and-slaves strategy, nor does it conspire with other programs in any other way—but in contrast, BWIN's and IMM01's performance depended greatly on the points fed to them by their slaves. In particular,

(1) If we average the score of each master with the scores of its slaves, we get 379.9 for BWIN and 351.7 for IMM01, both of which are considerably less than DBSz's score of 408.
(2) A more extensive analysis [Au and Nau, 2005] shows that if the size of each master-and-slaves team had been limited to less than or equal to 10, DBSz would have outperformed BWIN and IMM01 in the compe-

Table 10.2 Percentages of different interactions. *"All but M&S"* means all 105 programs that did not use master-and-slaves strategies, and *"all"* means all 165 programs in the competition.

Player 1	Player 2	(C,C)	(C,D)	(D,C)	(D,D)
BWIN	BWIN's slaves	12%	5%	64%	20%
IMM01	IMM01's slaves	10%	6%	47%	38%
CNGF	CNGF's slaves	2%	10%	10%	77%
BWIN's slaves	*all but M&S*	5%	9%	24%	62%
IMM01's slaves	*all but M&S*	7%	9%	23%	61%
CNGF's slaves	*all but M&S*	4%	8%	24%	64%
TFT	*all but M&S*	33%	20%	20%	27%
DBSz	*all but M&S*	54%	15%	13%	19%
TFTT	*all but M&S*	55%	20%	11%	14%
TFT	*all*	23%	19%	16%	42%
DBSz	*all*	36%	14%	11%	39%
TFTT	*all*	38%	21%	10%	31%
all but M&S	*all but M&S*	31%	19%	19%	31%
all	*all*	13%	16%	16%	55%

tition, even without averaging the score of each master with its slaves.

The reason for the above two phenomena is that the master-and-slaves strategies did not cooperate the other players as much as they did amongst themselves. In particular, Table 10.2 gives the percentages of each of the four possible interactions when any program from one group plays with any program from another group. Note that:

- When BWIN and IMM01 play with their slaves, about 64% and 47% of the interactions are (D,C), but when non-master-and-slaves strategies play with each other, only 19% of the interactions are (D,C).
- When the slave programs play with non-master-and-slaves programs, over 60% of interactions are (D,D), but when non-master-and-slaves programs play with other non-master-and-slaves programs, only 31% of the interactions are (D,D).
- The master-and-slaves strategies decrease the overall percentage of (C,C) from 31% to 13%, and increase the overall percentage of (D,D) from 31% to 55%.

10.8.3 *A comparison between DBSz, TFT, and TFTT*

Next, we consider how DBSz performs against TFT and TFTT. Table 10.2 shows that when playing with another cooperative player, TFT cooperates ((C,C) in the table) 33% of the time, DBSz does so 54% of the time, and

TFTT does so 55% of the time. Furthermore, when playing with a player who defects, TFT defects ((D, D) in the table) 27% of the time, DBSz does so 19% of the time, and TFTT does so 14% of the time. From this, one might think that DBSz's behavior is somewhere between TFT's and TFTT's.

But on the other hand, when playing with a player who defects, DBSz cooperates ((C, D) in the table) only 15% of the time, which is a lower percentage than for TFT and TFTT (both 20%). Since cooperating with a defector generates no payoff, this makes TFT and TFTT perform worse than DBSz overall. DBSz's average score was 408 and it ranked 3rd, but TFTT's and TFT's average scores were 388.4 and 388.2 and they ranked 30th and 33rd.

10.9 Related Work

Early studies of the effect of noise in the Iterated Prisoner's Dilemma focused on how TFT, a highly successful strategy in noise-free environments, would do in the presence of noise. TFT is known to be vulnerable to noise; for instance, if two players use TFT at the same time, noise would trigger long sequences of mutual defections [Molander, 1985]. A number of people confirmed the negative effects of noise to TFT [Molander, 1985; Bendor, 1987; Mueller, 1987; Axelrod and Dion, 1988; Nowak and Sigmund, 1990; Bendor et al., 1991]. Axelrod found that TFT was still the best decision rule in the rerun of his first tournament with a one percent chance of misperception [Axelrod, 1984, page 183], but TFT finished sixth out of 21 in the rerun of Axelrod's second tournament with a 10 percent chance of misperception [Donninger, 1986]. In Competition 2 of the 2005 IPD competition, the noise level was 0.1, and TFT's overall average score placed it 33rd out of 165.

The oldest approach to remedy TFT's deficiency in dealing with noise is to be more forgiving in the face of defections. A number of studies found that more forgiveness promotes cooperation in noisy environments [Bendor et al., 1991; Mueller, 1987]. For instance, Tit-For-Two-Tats (TFTT), a strategy submitted by John Maynard Smith to Axelrod's second tournament, retaliates only when it receives two defections in two previous iterations. TFTT can tolerate isolated instances of defections caused by noise and is more readily to avoid long sequences of mutual defections caused by noise. However, TFTT is susceptible to exploitation of its generosity and

was beaten in Axelrod's second tournament by TESTER, a strategy that may defect every other move. In Competition 2 of the 2005 IPD Competition, TFTT ranked 30—a slightly better ranking than TFT's. In contrast to TFTT, DBS can tolerate not only an isolated defection but also a sequence of defections caused by noise, and at the same time DBS monitors the other player's behavior and retaliates when exploitation behavior is detected (i.e., when the exploitation causes a change of the hypothesized policy, which initially is TFT). Furthermore, the retaliation caused by exploitation continues until the other player shows a high degree of remorse (i.e., cooperations when DBS defects) that changes the hypothesized policy to one with which DBS favors cooperations instead of defections.

[Molander, 1985] proposed to mix TFT with ALLC to form a new strategy which is now called Generous Tit-For-Tat (GTFT) [Nowak and Sigmund, 1992]. Like TFTT, GTFT avoids an infinite echo of defections by cooperating when it receives a defection in certain iterations. The difference is that GTFT forgives randomly: for each defection GTFT receives it randomly choose to cooperate with a small probability (say 10%) and defect otherwise. DBS, however, does not make use of forgiveness explicitly as in GTFT; its decisions are based entirely on the hypothesized policy that it learned. But temporary tolerance can be deemed as a form of forgiveness, since DBS does not retaliate immediately when a defection occurs in a mutual cooperation situation. This form of forgiveness is carefully planned and there is no randomness in it.

Another way to improve TFT in noisy environments is to use contrition: unilaterally cooperate after making mistakes. One strategy that makes use of contrition is Contrite TFT (CTFT) [Sugden, 1986; Boyd, 1989; Wu and Axelrod, 1995], which does not defect when it knows that noise has occurred and affected its previous action. However, this is less useful in the Noisy IPD since a program does not know whether its action is affected by noise or not. DBS does not make use of contrition, though the effect of temporary tolerance resembles contrition.

A family of strategies called "Pavlovian" strategies, or simply called Pavlov, was found to be more successful than TFT in noisy environments [Kraines and Kraines, 1989; Kraines and Kraines, 1993; Kraines and Kraines, 1995; Nowak and Sigmund, 1993]. The simplest form of Pavlov is called Win-Stay, Lose-Shift [Nowak and Sigmund, 1993], because it cooperates only after mutual cooperation or mutual defection, an idea similar to Simpleton [Rapoport and Chammah, 1965]. When an accidental defection occurs, Pavlov can resume mutual cooperation

in a smaller number of iterations than TFT [Kraines and Kraines, 1989; Kraines and Kraines, 1993]. Pavlov learns by conditioned response through rewards and punishments; it adjusts its probability of cooperation according to the previous outcome. Like Pavlov, DBS learns from its past experience and makes decisions accordingly. DBS, however, has an intermediate step between learning from experience and decision making: it maintains a model of the other player's behavior, and uses this model to reason about noise. Although there are probabilistic rules in the hypothesized policy, there is no randomness in its decision making process.

For readers who are interested, there are several surveys on the Iterated Prisoner's Dilemma with noise [Axelrod and Dion, 1988; Hoffmann, 2000; O'Riordan, 2001; Kuhn, 2001].

The use of opponent modeling is common in games of imperfect information such as Poker [Billings et al., 1998; Barone and While, 1998; Barone and While, 1999; Barone and While, 2000; Davidson et al., 2000; Billings et al., 2003] and RoShamBo [Egnor, 2000]. One entry in Axelrod's original IPD tournament used opponent modeling, but it was not successful. There have been many works on learning the opponent's strategy in the non-noisy IPD [Dyer, 2004; Hingston and Kendall, 2004; Powers and Shoham, 2005]. By assuming the opponent's next move depends only on the outcomes of the last few iterations, these works model the opponent's strategy as probabilistic finite automata, and then use various learning methods to learn the probabilities in the automata. For example, [Hingston and Kendall, 2004] proposed an adaptive agent called an opponent modeling agent (OMA) of order n, which maintains a summary of the moves made up to n previous iterations. Like DBS, OMA learns the probabilities of cooperations of the other player in different situations using an updating rule similar to the Equation 10.1, and generates a move based on the opponent model by searching a tree similar to that shown in Figure 10.4. The opponent model in [Dyer, 2004] also has a similar construct. The main way they differ from DBS is how they learn the other player's strategy, but there are several other differences: for example, the tree they used has a maximum depth of 4, whereas ours has a depth of 60.

The agents of both [Hingston and Kendall, 2004] and [Dyer, 2004] learned the other player's strategy by exploration—deliberately making moves in order to probe the other player's strategy. The use of exploration for learning opponent's behaviors was studied by [Carmel and Markovitch, 1998], who developed a lookahead-based exploration strategy to balance

between exploration and exploitation and avoid making risky moves during exploration. [Hingston and Kendall, 2004] and [Dyer, 2004] used a different exploration strategy than [Carmel and Markovitch, 1998]; [Hingston and Kendall, 2004] introduced noise to 1% of their agent's moves (they call this method the trembling hand), whereas the agent in [Dyer, 2004] makes decisions at random when it uses the opponent's model and finds a missing value in the model. Both of their agents used a random opponent model at the beginning of a game.

DBS does not make deliberate moves to attempt to explore the other player's strategy, because we believe that this is a high-risk, low-payoff business in IPD. We believe it incurs a high risk because many programs in the competition are adaptive; our defections made in exploration may affect our long-term relationship with them. We believe it has a low payoff because the length of a game is usually too short for us to learn any non-trivial strategy completely. Moreover, the other player may alter its behavior at the middle of a game, and therefore it is difficult for any learning method to converge. It is essentially true in noisy IPD, since noise can provoke the other player (e.g., GRIM). Furthermore, our objective is to cooperate with the other players, not to exploit their weakness in order to beat them. So as long as the opponent cooperates with us there is no need to bother with their other behaviors. For these reasons, DBS does not aim at learning the other player's strategy completely; instead, it learns the other player's recent behavior, which is subject to change. In contrast to the OMA strategy described earlier in this section, most of our DBS programs cooperated with each other in the competition.

Our decision-making algorithm combines elements of both minimax game tree search and the value iteration algorithm for Markov Decision Processes. In contrast to [Carmel and Markovitch, 1994], we do not model the other player's model of our strategy; we assume that the hypothesized policy does not change for the rest of the game. Obviously this assumption is not valid, because our decisions can affect the decisions of the other players in the future. Nonetheless, we found that the moves returned by our algorithm are fairly good responses. For example, if the other player behaves like TFT, the move returned by our algorithm is to cooperate regardless of the previous outcomes; if the other player does not behave like TFT, our algorithm is likely to return defection, a good move in many situations.

To the best of our knowledge, ours is the first work on using opponent models in the IPD to detect errors in the execution of another agent's

actions.

10.10 Summary and Future Work

For conflict prevention in noisy environments, a critical problem is to distinguish between situations where another player has misbehaved intentionally and situations where the misbehavior was accidental. That is the problem that DBS was formulated to deal with. DBS's impressive performance in the 2005 Iterated Prisoner's Dilemma competition occurred because DBS was better able to maintain cooperation in spite of noise than any other program in the competition.

To distinguish between intentional and unintentional misbehaviors, DBS uses a combination of symbolic noise detection plus temporary tolerance: if an action of the other player is inconsistent with the player's past behavior, we continue as if the player's behavior has not changed, until we gather sufficient evidence to see whether the inconsistency was caused by noise or by a genuine change in the other player's behavior.

Since clarity of behavior is an important ingredient of long-term cooperation in the IPD, most IPD programs have behavior that follows clear deterministic patterns. The clarity of these patterns made it possible for DBS to construct policies that were good approximations of the other players' strategies, and to use these policies to fend off noise.

We believe that clarity of behavior is also likely to be important in other multi-agent environments in which agents have to cooperate with each other. Thus it seems plausible that techniques similar to those used in DBS may be useful in those domains.

In the future, we are interested in studying the following issues:

- The evidence collection process takes time, and the delay may invite exploitation. For example, the policy of temporary tolerance in DBS may be exploited by a "hypocrite" strategy that behaves like TFT most of the time but occasionally defects even though DBS did not defect in the previous iteration. DBS cannot distinguish this kind of intentional defection from noise, even though DBS has built-in mechanism to monitor exploitation. We are interested to seeing how to avoid this kind of exploitation.
- In multi-agent environments where agents can communicate with each other, the agents might be able to detect noise by using a predefined communication protocol. However, we believe there is no protocol that

is guaranteed to tell which action has been affected by noise, as long as the agents cannot completely trust each other. It would be interesting to compare these alternative approaches with symbolic noise detection to see how symbolic noise detection could enhance these methods or vice versa.

- The type of noise in the competition assumes that no agent know whether an execution of an action has been affected by noise or not. Perhaps there are situations in which some agents may be able to obtain partial information about the occurrence of noise. For example, some agents may obtain a plan of the malicious third party by counter-espionage. We are interested to see how to utilize these information into symbolic noise detection.
- It would be interesting to put DBS in an evolutionary environment to see whether it can survive after a number of generations. Is it evolutionarily stable?

Acknowledgment. This work was supported in part by ISLE contract 0508268818 (subcontract to DARPA's Transfer Learning program), UC Berkeley contract SA451832441 (subcontract to DARPA's REAL program), and NSF grant IIS0412812. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

This work is based on an earlier work: *Accident or Intention: That Is the Question (in the Noisy Iterated Prisoner's Dilemma)*, in AAMAS'06 (May 8–12 2006) ©ACM, 2006.

We would like to thank the anonymous reviewers for their comments.

Bibliography

Au, T.-C. and Nau, D. (2005). An Analysis of Derived Belief Strategy's Performance in the 2005 Iterated Prisoner's Dilemma Competition. Technical Report CSTR-4756/UMIACS-TR-2005-59, University of Maryland, College Park.

Axelrod, R. (1984). *The Evolution of Cooperation*. Basic Books.

Axelrod, R. (1997). *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*. Princeton University Press.

Axelrod, R. and Dion, D. (1988). The further evolution of cooperation. *Science*, 242(4884):1385–1390.

Barone, L. and While, L. (1998). Evolving adaptive play for simplified poker. In *Proceedings of IEE International Conference on Computational Intelligence (ICEC-98)*, pages 108–113.

Barone, L. and While, L. (1999). An adaptive learning model for simplified poker using evolutionary algorithms. In *Proceedings of the Congreess of Evolutionary Computation (GECCO-1999)*, pages 153–160.

Barone, L. and While, L. (2000). Adaptive learning for poker. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 566–573.

Bendor, J. (1987). In good times and bad: Reciprocity in an uncertain world. *American Journal of Politicial Science*, 31(3):531–558.

Bendor, J., Kramer, R. M., and Stout, S. (1991). When in doubt... cooperation in a noisy prisoner's dilemma. *The Journal of Conflict Resolution*, 35(4):691–719.

Billings, D., Burch, N., Davidson, A., Holte, R., and Schaeffer, J. (2003). Approximating game-theoretic optimal strategies for full-scale poker. In *IJCAI*, pages 661–668.

Billings, D., Papp, D., Schaeffer, J., and Szafron, D. (1998). Opponent modeling in poker. In *AAAI*, pages 493–499.

Boyd, R. (1989). Mistakes allow evolutionary stability in the repeated prisoner's dilemma game. *Journal of Theoretical Biology*, 136:47–56.

Carmel, D. and Markovitch, S. (1994). The M* algorithms: Incorporating opponent models into adversary search. Technical Report CIS9402, Computer Science Department Technion.

Carmel, D. and Markovitch, S. (1998). How to explore your opponent's strategy (almost) optimally. In *Proceedings of the Third International Conference on Multi-Agent Systems*, pages 64–71.

Davidson, A., Billings, D., Schaeffer, J., and Szafron, D. (2000). Improved opponent modeling in poker. In *Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI'2000)*, pages 1467–1473.

Donninger, C. (1986). *Paradoxical Effects of Social Behavior*, chapter Is it always efficient to be nice?, pages 123–134. Heidelberg: Physica Verlag.

Dyer, D. W. (2004). Opponent modelling and strategy evolution in the iterated prisoner's dilemma. Master's thesis, School of Computer Science and Software Engineering, The University of Western Australia.

Egnor, D. (2000). Iocaine powder explained. *ICGA Journal*, 23(1):33–35.

Hingston, P. and Kendall, G. (2004). Learning versus evolution in iterated prisoner's dilemma. In *Proceedings of the Congress on Evolutionary Computation (CEC'04)*.

Hoffmann, R. (2000). Twenty years on: The evolution of cooperation revisited. *Journal of Artificial Societies and Social Simulation*, 3(2).

Kraines, D. and Kraines, V. (1989). Pavlov and the prisoner's dilemma. *Theory and Decision*, 26:47–79.

Kraines, D. and Kraines, V. (1993). Learning to cooperate with pavlov an adaptive strategy for the iterated prisoner's dilemma with noise. *Theory and Decision*, 35:107–150.

Kraines, D. and Kraines, V. (1995). Evolution of learning among pavlov strategies in a competitive environment with noise. *The Journal of Conflict Resolution*, 39(3):439–466.

Kuhn, S. T. (2001). Prisoner's dilemma. http://karmak.org/archive/2002/11/Prisoner's Dilemma.html Stanford Encyclopedia of Philosophy.

Molander, P. (1985). The optimal level of generosity in a selfish, uncertain environment. *The Journal of Conflict Resolution*, 29(4):611–618.

Mueller, U. (1987). Optimal retaliation for optimal cooperation. *The Journal of Conflict Resolution*, 31(4):692–724.

Nowak, M. and Sigmund, K. (1990). The evolution of stochastic strategies in the prisoner's dilemma. *Acta Applicandae Mathematicae*, 20:247–265.

Nowak, M. and Sigmund, K. (1993). A strategy of win-stay, lose-shift that outperforms tit-for-tat in the prisoner's dilemma game. *Nature*, 364:56–58.

Nowak, M. A. and Sigmund, K. (1992). Tit for tat in heterogeneous populations. *Nature*, 355:250–253.

O'Riordan, C. (2001). Iterated prisoner's dilemma: A review. Technical Report NUIG-IT-260601, Department of Information Technology, National University of Ireland, Galway.

Powers, R. and Shoham, Y. (2005). Learning against opponents with bounded memory. In *IJCAI*.

Rapoport, A. and Chammah, A. M. (1965). *Prisoner's dilemma*. University of Michigan Press.

Sugden, R. (1986). *The economics of rights, co-operation and welfare*. Blackwell.

Wu, J. and Axelrod, R. (1995). How to cope with noise in the iterated prisoner's dilemma. *Journal of Conflict Resolution*, 39:183–189.