Game Theory-Based Opponent Modeling in Large Imperfect-Information Games*

Sam Ganzfried and Tuomas Sandholm
Computer Science Department
Carnegie Mellon University
{sganzfri, sandholm}@cs.cmu.edu

ABSTRACT

We develop an algorithm for opponent modeling in large extensive-form games of imperfect information. It works by observing the opponent's action frequencies and building an opponent model by combining information from a precomputed equilibrium strategy with the observations. It then computes and plays a best response to this opponent model; the opponent model and best response are both updated continually in real time. The approach combines game-theoretic reasoning and pure opponent modeling, yielding a hybrid that can effectively exploit opponents after only a small number of interactions. Unlike prior opponent modeling approaches, ours is fundamentally game theoretic and takes advantage of recent algorithms for automated abstraction and equilibrium computation rather than relying on domain-specific prior distributions, historical data, or a hand-crafted set of features. Experiments show that our algorithm leads to significantly higher win rates (than an approximate-equilibrium strategy) against several opponents in limit Texas Hold'em — the most studied imperfect-information game in computer science — including competitors from recent AAAI computer poker competitions.

Categories and Subject Descriptors

I.2.m [**Computing Methodologies**]: Artificial Intelligence

General Terms

Algorithms, Economics

Keywords

Game theory, multiagent learning

1. INTRODUCTION

While much work has been done in recent years on abstracting and computing equilibria in large extensive-form games, relatively little work has been done on exploiting suboptimal opponents (aka *opponent modeling*). While playing an equilibrium guarantees at least the value of the game in a two-player zero-sum game, often much higher payoffs can be obtained by deviating from equilibrium to exploit opponents who make significant mistakes. For example, against a poker opponent who always folds, the strategy of always raising will perform far better than any equilibrium strategy (which will sometimes fold with bad hands).

Texas Hold'em poker has emerged as the main testbed for evaluating algorithms in extensive-form games. In addition to its tremendous popularity, it also contains enormous strategy spaces, imperfect information, and stochastic events; such elements also characterize most of the challenging problems in computational game theory and multiagent systems. In light of these factors and the AAAI annual computer poker competition, poker has emerged as an important, visible challenge problem for AI as a whole, and multiagent systems in particular.

It is worth noting, however, that a fair amount of prior work has been done on opponent exploitation in significantly smaller games. For example, Hoehn et al. [7] run experiments on Kuhn poker, a small two-player poker variant with about 20 states in its game tree. Recent work has also been done on opponent exploitation in rock-paper-scissors [12] and the repeated prisoners' dilemma [2]. However, these algorithms do not scale to large games. In contrast, the game tree of limit Texas hold'em has about 10^{18} states.

A potential drawback of evaluating algorithms on one specific problem is that we run the risk of developing algorithms that are so game specific that they will not generalize to other settings. Heeding this risk, in this work we abandon many of the game-specific assumptions taken by prior approaches. Rather than relying on massive databases of human poker play [3, 14] and expert-generated features or prior distributions [7, 16], we will instead rely on game-theoretic concepts such as Nash equilibrium and best response, which apply to all games.

In addition, we require our algorithms to operate efficiently in real time (online), as opposed to algorithms that perform offline computations assuming they have access to a large number of samples of the opponent's strategy in advance [9, 13]. That prior work also assumed access to historical data which included the private information of the opponents (i.e., their hole cards) even when such information was only observed by the opponent. In many multiagent settings, an agent must play against opponents about whom he has little to no information in advance, and must learn to

*This material is based upon work supported by the National Science Foundation under IIS grants 0905390 and 0964579. We also acknowledge Intel Corporation and IBM for their machine gifts.

Cite as: Game Theory-Based Opponent Modeling in Large Imperfect-Information Games, Sam Ganzfried and Tuomas Sandholm, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Tumer, Yolum, Sonenberg and Stone (eds.), May, 2–6, 2011, Taipei, Taiwan, pp. 533-540.

exploit weaknesses in a small number of interactions. Thus, we assume we have no prior information on our opponent's strategy in advance, and our algorithms will operate online.

Our main algorithm, called *Deviation-Based Best Response (DBBR)*, works by noting deviations between the opponent's strategy and that of a precomputed approximate equilibrium strategy, and constructing a model of the opponent based on these deviations. Then it computes and plays a best response to this opponent model (in real time). Both the construction of the opponent model and the computation of a best response take time linear in the size of the game tree and can be performed quickly in practice. As discussed above, we evaluate our algorithm empirically on limit Texas Hold'em; it achieves significantly higher win rates against several opponents — including competitors from recent AAAI computer poker competitions — than an approximate equilibrium strategy does.

2. GAME THEORY BACKGROUND

In this section, we review relevant definitions and results from game theory.

2.1 Extensive-form games

An *extensive-form* game is a general model of multiagent sequential decision-making with imperfect information[1]. As with perfect-information games, extensive-form games consist primarily of a game tree; each non-terminal node has an associated player (possibly *chance*) that makes the decision at that node, and each terminal node has associated utilities for the players. Additionally, game states are partitioned into *information sets*, $I_i \in \mathcal{I}_i$, where a player cannot distinguish among the states in the same information set. Therefore, the player whose turn it is to move must choose actions with the same distribution at each state in the information set.

In this paper, we will only concern ourselves with two-player, zero-sum[2], extensive-form games (though our algorithm extends naturally to multiplayer and non-zero-sum games as well). Furthermore, we will make the standard assumption of *perfect recall*: no player forgets information that he previously knew.

A *history*, $h \in H$, is a sequence of actions. A (mixed) *strategy* for player i, σ_i, is a function that assigns a probability distribution over all actions at each information set belonging to i; by convention the opponent's strategy is denoted σ_{-i}. Let Σ_i denote the (mixed) strategy space of player i. A *strategy profile* σ is a vector of strategies, one for each player.

In this paper we will assume that the moves of all players other than chance are observed by all players; for example, in poker all moves other than the initial dealing of the cards are publicly observed. In this setting, we can partition all game states into *public history sets*, PH_i, where states in the same public history set correspond to the same history of publicly observed actions. Note that each public history set must consist of a set of information sets of player i. For public history set $n \in PH_i$, let A_n denote the set of actions of player i at n. In general when we omit subscripts, player i will be implied.

[1]Much of our description of extensive-form games is adapted from [11].

[2]An extensive-form game is zero-sum if the sum of the payoffs at each terminal node equals zero.

2.2 Best responses and Nash equilibria

Player i's *best response* to σ_{-i} is any strategy in

$$\arg\max_{\sigma'_i \in \Sigma_i} u_i(\sigma'_i, \sigma_{-i}).$$

A *Nash equilibrium* is a strategy profile σ such that σ_i is a best response to σ_{-i} for all i. An ϵ-*equilibrium* is a strategy profile in which each player achieves a payoff of within ϵ of his best response. Formally, an ϵ-equilibrium is a strategy profile σ^* such that, for all i, we have

$$u_i(\sigma^*_i, \sigma^*_{-i}) \geq \max_{\sigma_i \in \Sigma_i} u_i(\sigma_i, \sigma^*_{-i}) - \epsilon.$$

All finite games have at least one Nash equilibrium. In the case of zero-sum extensive-form games with perfect recall, there are efficient techniques for finding an ϵ-equilibrium, such as linear programming (LP) [10], the excessive gap technique (EGT) [6], and counterfactual regret minimization (CFR) [17]. However, the latter two scale to much larger games; they scale to 10^{12} states in the game tree, while the best current LP techniques do not scale beyond 10^8 states.

Best responses can be computed much more efficiently than Nash equilibria. Computing a best response involves a single matrix-vector multiplication followed by a traversal up the game tree, both of which take linear time in the size of the game tree.

2.3 Abstraction

Despite the tremendous progress in equilibrium-finding in recent years, many interesting real-world games (such as poker) are so large that even the best algorithms have no hope of computing an equilibrium directly. The standard approach of dealing with this is to apply an *abstraction* algorithm, which constructs a smaller game that is similar to the original game; then the smaller game is solved, and its solution is mapped to a strategy profile in the original game. The approach has been applied to two-player Texas Hold'em poker, first with a manually generated abstraction [1], and currently with abstraction algorithms [4]. Many abstraction algorithms work by coarsening the moves of chance, collapsing several information sets of the original game into single information sets of the abstracted game. We will sometimes refer to information sets in abstracted games as *buckets*.

The game tree of limit Texas hold'em has about 10^{18} states, and recent solution techniques can compute approximate equilibria for abstractions with up to 10^{12} states [5, 17]. Such algorithms typically take several weeks to compute an ϵ-equilibrium for reasonably small ϵ. On the other hand, best responses in such an abstraction can be computed in about an hour. If coarser abstractions are used, best responses can be computed in minutes or even seconds, and can potentially be used as a subroutine in adaptive real-time algorithms.

3. IMPOSSIBILITY OF SAFE EXPLOITATION

While deviating from equilibrium to exploit an opponent can often lead to a significantly higher payoff, it also runs the risk that the exploitative strategy can itself become exploitable. For example, the opponent could play a certain

strategy for several iterations to trick the exploiter, then exploit him in turn; this is referred to as the *get-taught-and-exploited problem* [15].

One might think that this problem can be avoided by only risking the amount won so far. For example, suppose we are repeating a two-player zero-sum game (with value zero) 100 times, and have won $50 so far through 50 iterations. Then if we attempt to exploit the opponent for the next 50 iterations by playing a strategy with exploitability at most $1 per iteration, it appears that we may be able to safely exploit the opponent by deviating from equilibrium while still guaranteeing the value of the game. Unfortunately, this intuition is not correct; it is possible that the opponent was in fact playing an equilibrium all along and that we were just lucky for the first 50 iterations. If we then deviate from equilibrium, our overall strategy could actually have a negative payoff in expectation against an equilibrium opponent. Formally:

PROPOSITION 1. *It is not possible to exploit an opponent by deviating from equilibrium while simultaneously guaranteeing obtaining the value of the game in expectation.*

Thus, we must turn to algorithms that are exploitable to some extent in the worst case if we hope to exploit the opponent more than any equilibrium strategy does.

4. DBBR: AN EFFICIENT REAL-TIME OPPONENT MODELING ALGORITHM

In this section we present our algorithm, *Deviation-Based Best Response (DBBR)*. It works by observing the opponent's action frequencies over the course of game, then using these observations to construct a model of the opponent's strategy. Essentially, we would like to conservatively assume that the opponent is playing the best (i.e., least exploitable) strategy that is consistent with our observations of his play. The obvious way to accomplish this would be to add linear constraints to the LP for finding an equilibrium [10] that force the opponent model to conform with our observations. However, as discussed in Section 2.3, such a computation could take several weeks, and would not be practical for real-time play in large games.

To obtain a more practical algorithm, we must find a faster way of constructing an opponent model from our observations. DBBR constructs the model by noting deviations of our opponent's observed action frequencies from equilibrium frequencies. For example, in poker suppose an equilibrium strategy raises 50% of the time when first to act, while the opponent raises only 30% of the time. While the opponent might be raising any 30% of hands, a safe guess might be to assume that he is raising his 'best' 30% of hands; we can construct such a strategy by starting with the equilibrium strategy, then removing the 'worst' 20% of hands from the raising range. Our algorithm is based on this intuition.

4.1 Overview of the algorithm

Pseudocode for a high-level overview of DBBR is given in Algorithm 1. In the first step, an approximate equilibrium σ^* of the game is precomputed offline. Next, when the game begins, the frequencies of the opponent's actions at different public history sets are recorded. These are used to compute the opponent's *posterior action probabilities*: the probabilities with which he chooses each action at each public history set $n \in PH_{-i}$. (We say that the elements of PH_{-i} are numbered according to breadth-first-search (BFS) traversal order.) Next, we compute the probability the opponent is in each bucket at n given our model of his play so far; we refer to these probabilities as the *posterior bucket probabilities*. We then compute a full model of the opponent's strategy by considering the deviations between the opponent's posterior action probabilities and those of σ^* at n. Based on these deviations, we iterate over all buckets and shift weight away from the action probabilities in σ^* until we obtain a strategy consistent with our model of the opponent's action probabilities. Finally, after we have iterated over all public history sets, we compute a best response to the opponent model. The next subsections will discuss the different components of the algorithm in detail.

Algorithm 1 High-level overview of *DBBR*

```
Compute an approximate equilibrium of the game.
Maintain counters from observing opponent's play throughout the match.
for n = 1 to |PH_{-i}| do
    Compute posterior action probabilities at n.
    Compute posterior bucket probabilities at n.
    Compute full model of opponent's strategy at n.
end for
return Best response to the opponent model.
```

4.2 Computing posterior action probabilities

In the course of our play against the opponent, we observe how often he chooses each action a at each public history set n; we denote this quantity by $c_{n,a}$. One idea would be to assume the opponent will play action a with probability

$$\frac{c_{n,a}}{\sum_{a'} c_{n,a'}}.$$

However, doing this could be problematic for a few reasons. First, we might not have any observations at a given set n, in which case this quantity would not even be defined. More generally, the quality of our observations might vary dramatically between public history sets; for example, we have a lot more confidence in sets for which we have 1000 observations than sets for which we have just 1 or 2, and we would like our algorithm to reflect this. A similar observation was the motivation behind a recent paper [8], though that work assumed that the opponent's private information was observable.

Our algorithm works by choosing a combination of the observed probability and the probability under the equilibrium strategy σ^*, where the weight on the observed frequencies is higher at public history sets for which we have more observations. Specifically, we use a Dirichlet prior distribution, where we assume we have seen N_{prior} fictitious hands at the given public history set for which the opponent played according to σ^*. Let $p^*_{n,a}$ denote the probability that σ^* plays action a at public history set n. We compute the posterior action probabilities, $\alpha_{n,a}$, as follows:

$$\alpha_{n,a} = \frac{p^*_{n,a} \cdot N_{prior} + c_{n,a}}{N_{prior} + \sum_{a'} c_{n,a'}}. \quad (1)$$

4.3 Computing posterior bucket probabilities

Since we are constructing the model of the opponent's strategy using a BFS ordering of the public history sets,

we assume that we have already set his strategy for all ancestors of the current set n (including the parent n'). Let $s_{n',b,a}$ denote our model of the probability that the opponent plays his portion of the strategy sequence leading to n', then chooses action a in bucket b at state n'; this quantity has already been computed by the time we get to n in the algorithm. We can use these probabilities to construct the posterior probability, $\beta_{n,b}$, that the opponent is in bucket b (i.e., in poker, the opponent has those private cards) at public history set n. Pseudocode for this procedure is given in Algorithm 2, where h_b denotes the probability that chance makes the moves needed to put the opponent in bucket b.

Algorithm 2 ComputeBucketProbs(n)

for $b = 1$ **to** $|B_n|$ **do**
  $n' \leftarrow parent(n)$
  $a \leftarrow$ action taken to get from n' to n.
  $\beta_{n,b} \leftarrow h_b \cdot s_{n',b,a}$
end for
Normalize the values β_n so they sum up to 1.

4.4 Computing the opponent model

In this section we will present three different techniques for computing the opponent model. Recall that our high-level goal is to compute the 'best' (i.e., least exploitable) strategy for the opponent that is consistent with our observations of his behavior. We could accomplish this by performing an equilibrium-like computation; however, such a computation is too challenging to be performed in real time.

Rather than find the strategy consistent with our observations that is least exploitable, we will instead find the strategy that is 'closest' to the precomputed equilibrium. It turns out that this can be accomplished efficiently in practice, and intuitively we would expect strategies closer to equilibrium to be less exploitable.

4.4.1 Weighted L_1-distance minimization

Recall that the L_1 distance between two vectors x and y is defined as

$$||x - y||_1 = \sum_{i=1}^{k} |x_i - y_i|. \tag{2}$$

While this function treats all indices of the vector equally, in some cases we might want to put more weight on some components than on others. If p is a probability distribution over the integers from 1 to k, we define the *weighted* L_1 *distance* between x and y as

$$\sum_{i=1}^{k} p_i \cdot |x_i - y_i|. \tag{3}$$

Now, suppose we are at public history set n, where $\beta_{n,b}$ denotes the posterior probability that we are in bucket b, as computed by Algorithm 2. If we let the y_i's in Equation 3 correspond to the equilibrium probabilities of taking each action, and let the p_i's correspond to the $\beta_{n,b}$'s, then we can formulate the problem of finding the strategy closest to the precomputed equilibrium, subject to the posterior action probabilities $\alpha_{n,a}$, as an L_1-distance minimization problem.

Formally, we can formulate the optimization problem as follows, for a given public history set n:

$$\begin{aligned}
\text{minimize} \quad & \sum_{b \in B_n} \sum_{a \in A_n} \left[\beta_{n,b} \cdot |x_{n,b,a} - \sigma^*_{n,b,a}|\right] && (4)\\
\text{subject to} \quad & \sum_{b \in B_n} [\beta_{n,b} \cdot x_{n,b,a}] = \alpha_{n,a} \text{ for all } a \in A_n \\
& \sum_{a \in A_n} x_{n,b,a} = 1 \text{ for all } b \in B_n \\
& 0 \le x_{n,b,a} \le 1 \text{ for all } a \in A_n, b \in B_n
\end{aligned}$$

Recall that B_n denotes the set of all buckets we could be in at public history set n, while A_n denotes the set of actions at n. The variables $x_{n,b,a}$ correspond to the model of the opponent's strategy that we are trying to compute. Note that we can do this optimization separately for each public history set n; it makes more sense to do many smaller optimizations than to do a huge one for all public history sets at once, since the computations of the actions taken at different states do not depend on each other.

So as discussed above, we will perform a separate optimization at each n according to the program of Equation 4. It turns out that this can be cast as a linear program (LP) and solved efficiently using CPLEX's dual simplex algorithm for solving LPs. Doing this for each public history set n yields the opponent model x. Note that the program could have many solutions, and that CPLEX will just output the first solution it encounters (and not necessarily the solution that performs best in practice). This means that there might actually exist a strategy that minimizes L_1 distance from equilibrium that performs better in practice than the strategy output by CPLEX.

4.4.2 Weighted L_2-distance minimization

While Section 4.4.1 uses the weighted L_1 distance to measure the proximity of two strategies, we could also use other distance metrics. In this section we will consider another common distance function: the weighted L_2 distance.

Similarly to Equation 2, the L_2 distance between x and y is defined as

$$||x - y||_2 = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}. \tag{5}$$

Analogously to the L_1 case, we define the *weighted* L_2 *distance* between x and y as

$$\sqrt{\sum_{i=1}^{k} p_i \cdot (x_i - y_i)^2}. \tag{6}$$

The new program for computing the opponent model at n is the following:

$$\begin{aligned}
\text{minimize} \quad & \sum_{b \in B_n} \sum_{a \in A_n} \left[\beta_{n,b} \cdot (x_{n,b,a} - \sigma^*_{n,b,a})^2\right] && (7)\\
\text{subject to} \quad & \sum_{b \in B_n} [\beta_{n,b} \cdot x_{n,b,a}] = \alpha_{n,a} \text{ for all } a \in A_n \\
& \sum_{a \in A_n} x_{n,b,a} = 1 \text{ for all } b \in B_n \\
& 0 \le x_{n,b,a} \le 1 \text{ for all } a \in A_n, b \in B_n
\end{aligned}$$

Note that we can omit the square root, since it is a monotonic operator. The resulting formulation in Equation 7 is a

quadratic program (QP), which can also be solved efficiently in practice using CPLEX. As in the L_1 case, we can formulate and solve a separate optimization problem for each public history set n to compute the opponent model x.

4.4.3 Our custom weight-shifting algorithm

While the previous two sections described how to compute an opponent model using two popular distance functions, perhaps we can do even better by designing our own custom algorithm that takes into account the conservative reasoning about the opponent that we discussed earlier. In this section we will describe such an algorithm. In particular, it takes into account the fact that we already know an approximate ranking of the buckets at each public history set from the approximate equilibrium σ^*.

For example, suppose the opponent is only raising 30% of the time when first to act, while σ^* raises 50% of the time in that situation (as given in the example at the beginning of this section). Instead of doing a full L_1 or L_2-minimization explicitly, we could use the following heuristic algorithm: sort all buckets by how often the opponent raises with them under σ^*, then greedily keep removing buckets from his raising range until the weighted sum (using the $\beta_{n,b}$'s as weights) equals 30%. This is a simple greedy algorithm, which can be run significantly more efficiently in practice than the L_1 and L_2-minimization procedures described in the last two subsections, which must repeatedly use CPLEX at runtime.

For simplicity, we present our algorithm for the case of three actions, although it extends naturally to any number of actions. First we initialize the opponent's strategy at n, σ_n, to the equilibrium σ^*. We also initialize our current model of his action probabilities γ_n to $p^*_{n,a}$, the equilibrium action probabilities.

Next, we check whether the opponent is taking action 3 more often than he should at n by comparing $\alpha_{n,3}$ to $\gamma_{n,3}$. If he is, we are going to want to increase the probabilities he plays action 3 in various buckets; otherwise, we will decrease these probabilities. For now, we will assume that $\alpha_{n,3} > \gamma_{n,3}$ (the other case is handled analogously).

We start by adding weight to the bucket that plays action 3 with the highest probability at n; denote this bucket by $\hat{b}$. If

$$\gamma_{n,3} + \beta_{n,\hat{b}} \cdot (1 - \sigma_{n,\hat{b},3}) < \alpha_{n,3}, \tag{8}$$

we set $\sigma_{n,\hat{b},3} = 1$, since that will not cause $\gamma_{n,3}$ to exceed $\alpha_{n,3}$ once it is adjusted. Otherwise, we increase $\sigma_{n,\hat{b},3}$ by $\frac{(\alpha_{n,3} - \gamma_{n,3})}{\beta_{n,\hat{b}}}$. (Recall that $\beta_{n,\hat{b}}$ denotes the posterior probability that the opponent holds bucket $\hat{b}$ at n, as computed in Algorithm 2.) Let Δ denote the amount by which we increase $\sigma_{n,\hat{b},3}$. We will also increase the action probability $\gamma_{n,3}$ by $\beta_{\hat{b}} \cdot \Delta$.

Next we must compensate for this increase of the probability of playing action 3 in bucket $\hat{b}$ by decreasing the probabilities of playing actions 1 and/or 2. Let $\underline{a}$ denote the action (1 or 2) played with lower probability in σ_n in bucket $\hat{b}$, and let $\overline{a}$ denote the other action. If $\sigma_{n,\hat{b},\underline{a}} \geq \Delta$, then we set $\sigma_{n,\hat{b},\underline{a}} = \sigma_{n,\hat{b},\underline{a}} - \Delta$ and update $\gamma_{n,\underline{a}}$ accordingly. Otherwise, we set $\sigma_{n,\hat{b},\underline{a}} = 0$ and remove the remaining probability $\Delta - \sigma_{n,\hat{b},\underline{a}}$ from $\sigma_{n,\hat{b},\overline{a}}$.

If the inequality of Equation 8 held above, then our opponent model probabilities still do not agree with the posterior action probabilities, and thus we must continue shifting probability mass; we continue by setting $\hat{b}$ to the bucket that plays action 3 with the second highest probability at n, and repeating the above procedure. Otherwise, we are done setting the probabilities for action 3, and we perform a similar procedure to shift weight between the probabilities that he plays actions 1 and 2 until they agree with α_n.

We have now constructed an opponent model that agrees with our posterior action probabilities. Note that we had to iterate over possibly all of the buckets at public history set n. Since each bucket is contained in only one public history set, the algorithm's run time is linear in the size of the game tree.

Additionally, although we presented this algorithm for the case of three actions at n, it easily generalizes to more actions. Rather than just designating $\overline{a}$ and $\underline{a}$, we will sort all actions in the order of how often they are played in bucket $\hat{b}$, and proceed through this list adjusting probabilities as in the three-action case.

4.5 Full algorithm

In practice, constructing an opponent model and computing a best response at each repetition of the game (e.g., hand in poker) might be too slow. This can be mitigated by doing so only every k repetitions. In addition, we may want to start off playing the equilibrium σ^* for several repetitions so that we can obtain a reasonable number of samples of the opponent's play, rather than trying to exploit him immediately. Overall, our full algorithm will have three parameters: T denotes how many repetitions to first play the equilibrium σ^* before starting to exploit, k denotes how often to recompute an opponent model and best response, and N_{prior} from Equation 1 is the parameter of the action probability prior distributions. Pseudocode for the algorithm is given in Algorithm 3, where M is the number of repetitions in the match.

Algorithm 3 DBBR(T,k,N_{prior})

```
for iter = 1 to T do
    Play according to the precomputed equilibrium strategy σ*
end for
opponent_model = ComputeOppModel(N_prior)
σ_BR = ComputeBestResponse(opponent_model)
for iter = T + 1 to M do
    if iter is a multiple of k then
        opponent_model = ComputeOppModel(N_prior)
        σ_BR = ComputeBestResponse(opponent_model)
    end if
    Play according to σ_BR
end for
```

5. EXPERIMENTS AND DISCUSSION

We used two-player Limit Texas Hold'em as our experimental domain. It is a large-scale game with 10^{18} states in the game tree. It is the most-studied full-scale poker game in computer science, and is also played by human professionals.

5.1 Limit Texas Hold'em

The rules of the game are as follows. Each player at the table is dealt two private *hole cards*, and the players initially have 1 and 2 chips invested in the pot respectively. Then

there is a round of betting, after which three cards (called the *flop*) are dealt face up in the middle of the table. Then there is another round of betting, followed by another card dealt face up (the *turn*); then one more round of betting, followed by a fifth card face up (the *river*), followed by a final round of betting.

During each betting round, each player has three possible options. (1) *fold*: pass and forfeit his chance of winning the pot. (2) *call*: put a number of chips equal to the size of the current bet into the pot. (3) *raise*: put a fixed number of additional chips in the pot beyond what was needed to call.

If one player folds during the course of betting, then the other player wins the entire pot. If neither player has folded, the player with the best five-card hand (constructed from his two hole cards and the five community cards) wins the pot. In case of a tie, the players split the pot evenly.

As in the AAAI computer poker competitions, in our experiments, each *match* consists of 3000 *duplicate hands*: 3000 hands are played normally, then the players switch positions and play the same 3000 hands (with no memory of the previous hands). This is a well-known technique for reducing the variance so that fewer hands are needed to obtain statistical significance. Whenever we match two players, we have them play several duplicate matches and report the standard error.

5.2 Experimental results

We ran our algorithm against several opponents; the results are shown in Table 1. The first four opponents — Random, AlwaysFold, AlwaysCall, and AlwaysRaise — play naïvely as their names suggest. GUS2 and Dr. Sahbak were entrants in the 2008 AAAI computer poker competition, and Tommybot was an entrant in the 2009 competition; we selected these bots to experiment against because they had the worst performances in the competitions, and we expect opponent modeling to provide the biggest improvement against weak opponents. Against stronger opponents one might prefer to always play the precomputed equilibrium rather than turning on the exploitation. This can be accomplished by periodically looking at the win rate, and only attempting to exploit the opponent if a win rate above some threshold is attained.

GS5 is a bot we entered in the 2009 AAAI computer poker competition that plays an approximate-equilibrium strategy. It was computed using an abstraction which had branching factors of 15, 40, 6, and 6 respectively in the four betting rounds. The parameter values we used in DBBR (as described in Section 4.5) were $T = 1000$, $k = 50$, $N_{prior} = 5$, with GS5 playing the role of the initial approximate-equilibrium strategy (i.e., we ran GS5 for the first 1000 hands of each match and recomputed an opponent model and best response every 50 hands subsequently). Since each match consists of 3000 duplicate hands, this means that GS5 and DBBR play the same strategy for the first third of each match.

We set $T = 1000$ since it is essential that our algorithm obtains a reasonable number of samples of the opponent's play (in different parts of the game tree) before attempting to exploit. As discussed in the next paragraph, our main motivation in setting k was to allow us to update the opponent model as frequently as we could while remaining under the competition time limit. For N_{prior}, we wanted to choose a small number so that our observations would quickly trump the prior for common public history sets, but so that the prior would have more weight if we had just one or two observations. Note that setting $N_{prior} = 5$ means that our prior and our observations will have equal weight in our model when we have observed the opponent's action 5 times at the given public history set. Changing the parameter values could certainly have a large effect on the results, and should be studied further.

Unfortunately GS5 was too large to use as the approximate-equilibrium strategy in our real-time opponent modeling updates. Therefore, we also precomputed an approximate-equilibrium σ^* that used a much smaller abstraction than GS5: the branching factors of its abstraction were 8, 12, 4, and 4. While σ^* is clearly an inferior strategy to GS5, it was small enough to allow us to construct opponent models and compute best responses in just a few seconds, keeping us within the time limit of the AAAI competition.

We experimented with all three of the approaches for computing the opponent model described in Section 4.4: the three algorithms DBBR-L_1, DBBR-L_2, and DBBR-WS (i.e., 'Weight-Shifting') correspond to the three different algorithms in that section. We ran all three of these algorithms against each of the opponents described above (with the exception of Tommybot, which we were not able to play against DBBR-L_1 and DBBR-L_2 due to technical issues).

As shown in Table 1, our main algorithm DBBR-WS performed significantly better against all of the opponents than GS5 did (in one case, the win rate was over twice as high). Furthermore, DBBR-WS beat GUS2 by more than any other bot in the 2008 competition did, and its win rates against Dr. Sahbak and Tommybot were surpassed by the win rate of just a single bot.

5.3 Comparing the opponent modeling algorithms

It is not totally clear from the results in Figure 1 which of the three algorithms for constructing the opponent model — L_1, L_2, or our weight-shifting algorithm — is best. For example, DBBR-WS obtains a win rate of 1.391 sb/h against AlwaysRaise while DBBR-L_1 obtains a win rate of 0.878 sb/h, but DBBR-L_1 obtains a win rate of 2.164 sb/h against Random while DBBR-WS obtains only 1.769 sb/h. Similarly, for all other pairings there exist opponents such that one bot achieves a higher win rate against one opponent, but not against the other opponent. So there is no clear total ordering of the three algorithms.

That being said, DBBR-L_2 does at least as well (or essentially the same) against all of the opponents as DBBR-L_1, except for Dr. Sahbak; this suggests that DBBR-L_2 is a stronger program. As between DBBR-L_2 and DBBR-WS, it really seems to depend on the opponent. DBBR-WS performs significantly better against AlwaysRaise, GUS2, and Dr. Sahbak and slightly better against AlwaysFold than DBBR-L_2; however, DBBR-L_2 performs significantly better against Random and slightly better against AlwaysCall than DBBR-WS. So DBBR-WS performs significantly better against three of the six opponents than DBBR-L_2 (and essentially the same against two opponents), suggesting that it is a better algorithm.

In addition, DBBR-WS performs significantly better against both of the actual opponents from the AAAI competition (GUS2 and Dr. Sahbak) than DBBR-L_2, which suggests that it might perform better in practice against realistic op-

	Random	AlwaysFold	AlwaysCall	AlwaysRaise	GUS2	Dr. Sahbak	Tommybot
GS5	0.854 ± 0.008	0.646 ± 0.0009	0.582 ± 0.005	0.791 ± 0.009	0.636 ± 0.004	0.665 ± 0.027	0.552 ± 0.008
DBBR-WS	1.769 ± 0.025	0.719 ± 0.002	0.930 ± 0.014	1.391 ± 0.034	0.807 ± 0.011	1.156 ± 0.043	1.054 ± 0.044
DBBR-L_1	2.164 ± 0.036	0.717 ± 0.002	0.935 ± 0.017	0.878 ± 0.032	0.609 ± 0.054	1.153 ± 0.074	
DBBR-L_2	2.287 ± 0.046	0.716 ± 0.002	0.931 ± 0.026	1.143 ± 0.084	0.721 ± 0.050	1.027 ± 0.072	

Table 1: Win rate in small bets/hand of the bot listed in the row. The ± given is the standard error (standard deviation divided by the square root of the number of hands).

ponents. This fact, combined with the fact that DBBR-WS is more efficient than the other algorithms, which have to perform many optimizations using CPLEX at runtime, suggest that DBBR-WS is a better algorithm to use in practice.

Note that this does not imply that the weighted L_1 and L_2 distance functions are poor distance metrics; it just means that the particular solution output by CPLEX does not do as well as the solution output by DBBR-WS. It is very possible that if CPLEX used different LP/QP algorithms, it might find a solution that does significantly better. This would certainly be a worthwhile avenue for future work.

5.4 Win rates over time

One might expect that DBBR[3] would immediately begin exploiting the opponents at hand 1001 — when it switches from playing an approximate equilibrium to opponent modeling — and that the win rate would increase steadily. In fact, this happened in the matches against most of the bots. For example, Figure 1(a) shows that DBBR's profits against AlwaysFold increase linearly over time, and Figure 1(d) shows that DBBR's win rate increases in a concave fashion.

Surprisingly, we observed a different behavior in the matches against AlwaysRaise and GUS2. In both of these matches, the win rate decreases significantly for the first several hundred hands before it starts to increase, as shown in Figure 1. This happens because the approximate-equilibrium strategy plays some action sequences with very low probability, leading it to not explore the opponent's full strategy space in the 1000 hands. This will lead to a significant disparity between the prior and actual strategies of the opponent at hand 1001 if the opponent's strategy differs significantly from the approximate equilibrium in those unexplored regions. This in turn may cause DBBR to think it can immediately exploit the opponent in certain ways, which turn out to be unsuccessful; but eventually as DBBR explores these sequences further and gathers more observations, it figures out successful exploitations.

The following hand from our experiments between DBBR and AlwaysRaise exemplifies this phenomenon. The hand was the 1006'th hand of the match. There were many raises and re-raises during the preflop, flop, and turn betting rounds. When the river card came, DBBR had only a ten high (a very weak hand in this situation). However, based on its observations during the first 1005 hands, it knew that AlwaysRaise had a very wide range of hands given this betting sequence, many of which were also weak hands (though probably still stronger than ten high). On the other hand, DBBR had very few observations of how AlwaysRaise responds to a series of raises on the river, since GS5 made those plays very rarely during the first 1000 hands; hence, DBBR resorted to the prior to model the opponent, which had the opponent folding all of his weak hands to a raise (since GS5 would do this). So DBBR thought that raising would get the opponent to fold most of his hands, while in reality AlwaysRaise continues to raise with all of his hands. In this particular hand, DBBR lost a significant amount of money due to the additional raises he made on the river with a very weak hand.

[3]The results in this section refer to our main algorithm, DBBR-WS.

6. CONCLUSION

We presented DBBR, an efficient real-time algorithm for opponent modeling and exploitation in large extensive-form games. It works by observing the opponent's action frequencies and building an opponent model by combining information from a precomputed equilibrium strategy with the observations. This enables the algorithm to combine game-theoretic reasoning and pure opponent modeling, yielding a hybrid that can effectively exploit opponents after a small number of interactions.

Our experiments in full-scale two-player limit Texas Hold-'em poker show that DBBR is effective in practice against a variety of opponents, including several entrants from recent AAAI computer poker competitions. DBBR achieved a significantly higher win rate than an approximate-equilibrium strategy against all of the opponents in our experiments. Furthermore, it achieved a higher win rate against the opponents from previous competitions than all of the entrants from that year's competition achieved (except for at most one). We compared three different algorithms for constructing the opponent model, and conclude that our custom weight-shifting algorithm outperforms algorithms that employ weighted L_1 and L_2-distance minimization.

While DBBR is able to effectively exploit weak opponents, it might actually become significantly exploitable to strong opponents (e.g., opponents who operate in a finer-grained abstraction). Thus, we would like to only attempt to exploit weak opponents, while playing the equilibrium against strong opponents. This can be accomplished by periodically looking at the win rate, and only attempting to exploit the opponent if a win rate above some threshold is attained. Our current work involves developing automated schemes that alternate between DBBR and equilibrium play based on the specific opponent at hand. In addition, DBBR could be extended to the setting where the opponent's private information from the previous game iteration is sometimes observed. Finally, future work could look at more robust versions of DBBR, where the opponent model allows the opponent to sometimes deviate from his observed action probabilities, or a safer strategy than the actual best response is used.

7. REFERENCES

[1] Darse Billings, Neil Burch, Aaron Davidson, Robert Holte, Jonathan Schaeffer, Terence Schauenberg, and Duane Szafron. Approximating game-theoretic optimal strategies for full-scale poker. *IJCAI*, 2003.

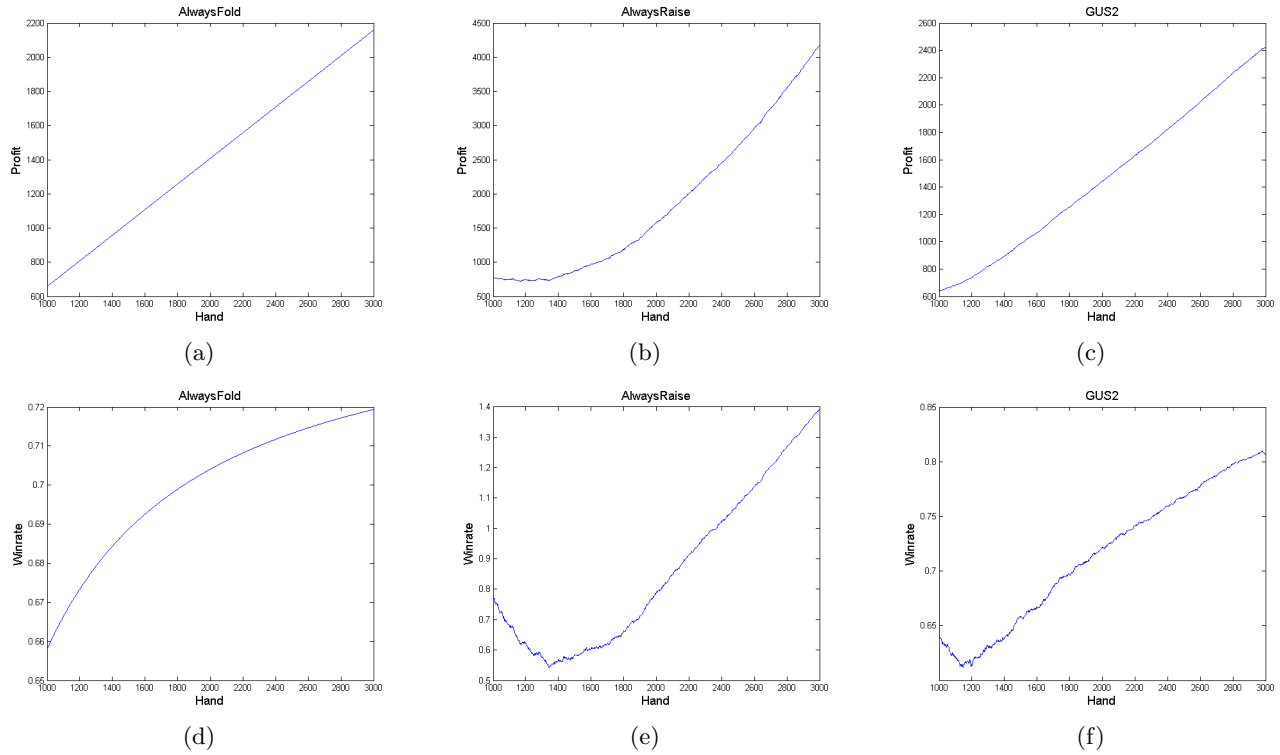

Figure 1: Profits and win rates over time of DBBR-WS against several opponents. Results against AlwaysFold are shown in Figures 1(a) and 1(d), results against AlwaysRaise are shown in Figures 1(b) and 1(e), and results against GUS2 are shown in Figures 1(c) and 1(f). The top three graphs show profit over time, and the bottom three show win rates over time.

[2] Doran Chakraborty and Peter Stone. Convergence, targeted optimality, and safety in multiagent learning. *ICML*, 2010.

[3] Aaron Davidson, Darse Billings, Jonathan Schaeffer, and Duane Szafron. Improved opponent modeling in poker. *IJCAI*, 2000.

[4] Andrew Gilpin and Tuomas Sandholm. A competitive Texas Hold'em poker player via automated abstraction and real-time equilibrium computation. *AAAI*, 2006.

[5] Andrew Gilpin, Tuomas Sandholm, and Troels Bjerre Sørensen. Potential-aware automated abstraction of sequential games, and holistic equilibrium analysis of Texas Hold'em poker. *AAAI*, 2007.

[6] Andrew Gilpin, Samid Hoda, Javier Peña, and Tuomas Sandholm. Gradient-based algorithms for finding Nash equilibria in extensive form games. *WINE*, 2007. Extended version in *Math. of OR*, 2010.

[7] Bret Hoehn, Finnegan Southey, Robert C. Holte, and Valeriy Bulitko. Effective short-term opponent exploitation in simplified poker. *AAAI*, 2005.

[8] Michael Johanson and Michael Bowling. Data biased robust counter strategies. *AISTATS*, 2009.

[9] Michael Johanson, Martin Zinkevich, and Michael Bowling. Computing robust counter-strategies. *NIPS*, 2007.

[10] Daphne Koller, Nimrod Megiddo, and Bernhard von Stengel. Efficient computation of equilibria for extensive two-person games. *GEB*, 1996.

[11] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte Carlo sampling for regret minimization in extensive games. *COLT workshop on Online Learning with Limited Feedback*, 2009.

[12] Peter McCracken and Michael Bowling. Safe strategies for agent modelling in games. *AAAI Fall Symposium on Artificial Multi-agent Learning*, 2004.

[13] Marc Ponsen, Marc Lanctot, and Steven de Jong. MCRNR: Fast computing of restricted Nash responses by means of sampling. *AAAI workshop on Interactive Decision Theory and Game Theory Workshop*, 2010.

[14] Marc Ponsen, Jan Ramon, Tom Croonenborghs, Kurt Driessens, and Karl Tuyls. Bayes-relational learning of opponent models from incomplete information in no-limit poker. *AAAI*, 2008.

[15] Tuomas Sandholm. Perspectives on multiagent learning. *Artificial Intelligence*, 2007.

[16] Finnegan Southey, Michael Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. Bayes' bluff: Opponent modelling in poker. *UAI*, 2005.

[17] Martin Zinkevich, Michael Bowling, Michael Johanson, and Carmelo Piccione. Regret minimization in games with incomplete information. *NIPS*, 2007.