Game Tree Search with Adaptation in Stochastic Imperfect Information Games

Darse Billings, Aaron Davidson, Terence Schauenberg, Neil Burch,
Michael Bowling, Robert Holte, Jonathan Schaeffer, and Duane Szafron

University of Alberta, Edmonton, Alberta, Canada T6G 2E8
{darse,davidson,terence,burch,bowling,holte,jonathan,duane}@cs.ualberta.ca

Abstract. Building a high-performance poker-playing program is a challenging project. The best program to date, PsOpti, uses game theory to solve a simplified version of the game. Although the program plays reasonably well, it is oblivious to the opponent's weaknesses and biases. Modeling the opponent to exploit predictability is critical to success at poker. This paper introduces Vexbot, a program that uses a game tree search algorithm to compute the expected value of each betting option, and does real-time opponent modeling to improve its evaluation function estimates. The result is a program that defeats PsOpti convincingly, and poses a much tougher challenge for strong human players.

1 Introduction

Modeling the preferences and biases of users is an important topic in recent artificial intelligence (AI) research. For example, this type of information can be used to anticipate user program commands [1], predict web buying and access patterns [2], and automatically generate customized interfaces [3]. It is usually easy to gather a corpus of data on a user (e.g., web page accesses), but mining that data to predict future patterns (e.g., the next web page request) is challenging. Predicting human strategies in a competitive environment is even more challenging.

The game of poker has become a popular domain for exploring challenging AI problems. This has led to the development of programs that are competitive with strong human players. The current best program, PsOpti [4], is based on approximating a game-theoretic Nash equilibrium solution. However, there remains a significant obstacle to overcome before programs can play at a world-class level: opponent modeling. As the world-class poker player used to test PsOpti insightfully recognized [4]:

> "*You have a very strong program. Once you add opponent modeling to it, it will kill everyone.*"

This issue has been studied in two-player perfect information games (e.g., [5–7]), but has not played a significant role in developing strong programs. In poker, however, opponent modeling is a critical facet of strong play. Since a player has imperfect information (does not know the opponent's cards), any information that a player can glean from the opponent's past history of play can be used to improve the quality of future decisions. Skillful opponent modeling is often the differentiating factor among world-class players.

Opponent modeling is a challenging learning problem, and there have been several attempts to apply standard machine learning techniques to poker. Recent efforts include: neural nets [8], reinforcement learning [9], and Bayesian nets [10], which have had only limited success. There are a number of key issues that make this problem difficult:

1. Learning must be rapid (within 100 games, preferably fewer). Matches with human players do not last for many thousands of hands, but the information presented over a short term can provide valuable insights into the opponent's strategy.
2. Strong players change their playing style during a session; a fixed style is predictable and exploitable.
3. There is only partial feedback on opponent decisions. When a player folds (a common scenario), their cards are not revealed. They might have had little choice with a very weak hand; or they might have made a very good play by folding a strong hand that was losing; or they might have made a mistake by folding the best hand. Moreover, understanding the betting decisions they made earlier in that game becomes a matter of speculation.

This paper presents a novel technique to automatically compute an exploitive counter-strategy in stochastic imperfect information domains like poker. The program searches an imperfect information game tree, consulting an opponent model at all opponent decision nodes and all leaf nodes. The most challenging aspects to the computation are determining: 1) the probability that each branch will be taken at an opponent decision node, and 2) the expected value (EV) of a leaf node. These difficulties are due to the hidden information and partial observability of the domain, and opponent models are used to estimate the unknown probabilities. As more hands are played, the opponent modeling information used by the tree search generally becomes more accurate, thus improving the quality of the evaluations. Opponents can and will change their style during a playing session, so old data needs to be gradually phased out.

This paper makes the following contributions:

1. *Miximax* and *Miximix*, applications of the Expectimax search algorithm to stochastic imperfect information adversarial game domains.
2. Using opponent modeling to refine expected values in an imperfect information game tree search.
3. Abstractions for compressing the large set of observable poker situations into a small number of highly correlated classes.
4. The program VEXBOT, which convincingly defeats strong poker programs including PSOPTI, and is competitive with strong human players.

2 Texas Hold'em Poker

Texas Hold'em is generally regarded as the most strategically complex poker variant that is widely played in casinos and card clubs. It is the game used in the annual World Series of Poker to determine the world champion.

A good introduction to the rules of the game can be found in [8]. The salient points needed for this paper are that each player has two private cards (hidden from the opponent), some public cards are revealed that are shared by all players (community cards), and each player is asked to make numerous betting decisions: either *bet/raise* (increase the stakes), *check/call* (match the current wager and stay in the game), or *fold* (quit and lose all money invested thus far). A game ends when all but one player folds, or when all betting rounds are finished, in which case each player reveals their private cards and the best poker hand wins (the *showdown*). The work discussed in this paper concentrates on two-player Limit Texas Hold'em.

Computer Poker Programs

The history of computer poker programs goes back more than 30 years to the initial work by Findler [11]. Some of the mathematical foundations go back to the dawn of game theory [12, 13]. Recently, most of the AI literature has been produced by the Computer Poker Research Group at the University of Alberta. Those programs—LOKI, POKI, and PSOPTI— illustrate an evolution of ideas that has taken place in the quest to build a program capable of world-class play:

- Rule-based [14]: Much of the program's knowledge is explicit in the form of expert-defined rules, formulas, and procedures.
- Simulations [15, 8]: Betting decisions are determined by simulating the rest of the game. Likely card holdings are dealt to the opponents and the hand is played out. A large sample of hands is played, and the betting decision with the highest expected value is chosen.
- Game theory [4]: Two-player Texas Hold'em has a search space size of $O(10^{18})$. This was abstracted down to a structurally similar game of size $O(10^7)$. Linear programming was used to find a Nash equilibrium solution to that game (using techniques described in [16]), and the solution was then mapped back onto real poker. The resulting solution – in effect, a strategy lookup table – has the advantage of containing no expert-defined knowledge. The resulting *pseudo-optimal* poker program, PSOPTI, plays reasonably strong poker and is competitive with strong players. However, the technique is only a crude approximation of equilibrium play. Strong players can eventually find the seams in the abstraction and exploit the resulting flaws in strategy. Nevertheless, this represented a large leap forward in the abilities of poker-playing programs.

As has been seen in many other game-playing domains, progressively stronger programs have resulted from better algorithms and *less* explicit knowledge.

3 Optimal versus Maximal Play

In the literature on game theory, a Nash equilibrium solution is often referred to as an *optimal strategy.* However, the adjective "optimal" is dangerously misleading when applied to a poker program, because there is an implication that an equilibrium strategy will perform better than any other possible solution. "Optimal" in the game theory sense has a specific technical meaning that is quite different.

A Nash equilibrium strategy is one in which no player has an incentive to deviate from the strategy, because the alternatives *could* lead to a worse result. This simply maximizes the minimum outcome (sometimes referred to as the *minimax solution* for two-player zero-sum games). This is essentially a defensive strategy that implicitly assumes the opponent is perfect in some sense (which is definitely not the case in real poker, where the opponents are highly fallible).

A Nash equilibrium player will not necessarily defeat a non-optimal opponent. For example, in the game of rock-paper-scissors, the equilibrium strategy is to select an action uniformly at random among the three choices. Using that strategy means that no one can defeat you in the long term, but it also means that you will not win, since you have an expected value of zero against *any* other strategy.

Unlike rock-paper-scissors, poker is a game in which some strategies are *dominated*, and could potentially lose to an equilibrium player. Nevertheless, even a relatively weak and simplistic strategy might break even against a Nash equilibrium opponent, or not lose by very much over the long term. There are many concrete examples of this principle, but one of the clearest demonstrations was seen in the game of Oshi-Zumo [17].

In contrast, a *maximal* player can make moves that are non-optimal (in the game-theoretic sense) when it believes that such a move has a higher expected value. The *best response* strategy is one example of a maximal player.

Consider the case of rock-paper-scissors where a opponent has played "rock" 100 times in a row. A Nash equilibrium program is completely oblivious to the other player's tendencies, and does not attempt to punish predictable play in any way. A maximal player, on the other hand, will attempt to exploit perceived patterns or biases. This always incurs some risk (the opponent *might* have been setting a trap with the intention of deviating on the 101st hand). A maximal player would normally accept this small risk, playing "paper" with a belief of positive expectation [18].

Similarly, a poker program can profitably deviate from an equilibrium strategy by observing the opponent's play and biasing its decision-making process to exploit the perceived weaknesses.

If PsOpti was based on a true Nash equilibrium solution, then no human or computer player could expect to defeat it in the long run. However, PsOpti is only an approximation of an equilibrium strategy, and it will not be feasible to compute a true Nash equilibrium solution for Texas Hold'em in the foreseeable future. There is also an important practical limitation to this approach. Since PsOpti uses a fixed strategy, and is oblivious to the opponent's strategy, a strong human player can systematically explore various options, probing for weaknesses without fear of being punished for using a highly predictable style. This kind of methodical exploration for the most effective counter-strategy is not possible against a rapidly adapting opponent.

Moreover, the key to defeating all human poker players is to exploit their highly non-optimal play. This requires a program that can observe an opponent's play and adapt to dynamically changing conditions.

4 *Miximax* and *Miximix* Search

Expectimax search is the counterpart of minimax search for domains with a stochastic element [19]. Expectimax combines the minimization and maximization nodes of minimax search with the addition of chance nodes, where a stochastic event happens (for example, a dice roll). The value of a chance node is the sum of the values of each of the children of that node, weighted by the probability of that event occurring (1/6 for each roll in the case of a die).

For perfect information stochastic games such as backgammon, an opponent decision node is treated as a normal max (or min) node. However, this cannot be done for imperfect information games like poker, because the nodes of the tree are not independent. Several opponent decision nodes belong to the same information set, and are therefore indistinguishable from each other. In poker, the information set is comprised of all the possible opponent hands, and our policy must be the same for all of those cases, since we do not know the opponent's cards. Furthermore, a player can, in general, use a *randomized mixed strategy* (a probability distribution over the possible actions), so taking the maximum (or minimum) value of the subtrees is not appropriate.

To extend Expectimax for poker, we handle all of the opponent decision nodes within a particular information set as a single node, implicitly maintaining a probability distribution over the range of possible hands they might hold. We cannot treat all possible combinations as having equal probability (for example, weak hands might be folded early and strong hands played through to the end). Imperfect information adds an extra challenge in evaluating leaf nodes in the search, since we can only estimate the relative probabilities for the opponent's possible holdings, rather than having exact values.

We have implemented two variants of Expectimax for search on poker trees, which we call *Miximax* and *Miximix*. These algorithms compute the expected value (EV) at decision nodes of an imperfect information game tree by modeling them as chance nodes with probabilities based on the information known or estimated about the domain, and the specific opponent.

The algorithm performs a full-width depth-first search to the leaf nodes of the imperfect information game tree. For two-player poker, the leaf nodes are terminal nodes that end the game — either at a showdown, or when one of the players folds. The search tree is the set of all possible betting sequences from the current state to the end of the game, over all possible outcomes of future chance nodes. At the showdown leaf nodes, the probability of winning is estimated with a heuristic evaluation function, and the resulting EV is backed-up the tree. This tree can be a fairly large (millions of nodes), but with efficient data structures and caching of intermediate calculations, it can be computed in real-time (about one second). In general, the search can be stopped at any depth, and the evaluation function used to estimate the EV of that subtree, as is done in traditional game-playing programs.

The EV calculation is used to decide which action the program should perform: bet/raise, check/call, or fold. Given the EV for each of our three possible actions, one could simply select the option with the maximum value. In that case, the tree will contain mixed nodes for the opponent's decisions and max nodes for our own decisions. Hence we call this algorithm *Miximax*.

However, always taking the maximum EV could lead to predictable play that might be exploited by an observant opponent. Instead, we could choose to use a mixed strategy ourselves. Although we (presumably) know the randomized policy we will use, it can be viewed as both players having mixed nodes, and we call this more general algorithm *Miximix*. (Thus *Miximax* is a special case of *Miximix* in which all of our own decision nodes use a *pure strategy*, choosing one action 100% of the time).

There are two unresolved issues:

1. How to determine the relative probabilities of the opponent's possible actions at each decision node. This is based on frequency counts of past actions at corresponding nodes (*ie.* given the same betting sequence so far).
2. How to determine the expected value of a leaf node. At fold nodes, computing the EV is easy — it is the net amount won or lost during the hand. At showdown nodes, a probability density function over the strength of the opponent's hand is used to estimate our probability of winning. This histogram is an empirical model of the opponent, based on the hands shown in corresponding (identical or similar) situations in the past.

In summary, the search tree consists of four types of nodes, each with different properties:

Chance nodes: For chance nodes in the game tree, the EV of the node is the weighted sum of the EVs of the subtrees associated with each possible outcome. In Texas Hold'em, chance outcomes correspond to the dealing of public board cards. To be perfectly precise, the probability of each possible chance outcome is dependent on the cards that each player will likely hold at that point in the game tree. However, since that is difficult or impossible to determine, we currently make the simplifying (but technically incorrect) assumption that the chance outcomes occur uniformly at random. Thus the EV of a chance node is simply the average EV over all of the expansions. Let $Pr(C_i)$ be the probability of each branch i of chance node C, and let n be the number of branches. The EV of node C is:

$$EV(C) \ = \sum_{1 \ \leq \ i \ \leq \ n} Pr(C_i) \times EV(C_i) \tag{1}$$

Opponent decision nodes: Let $Pr(O_i)$ be the estimated probability of each branch i (one of fold, call, or raise) at an opponent decision node O. The EV of node O is the weighted sum:

$$EV(O) \;=\; \sum_{i \,\epsilon\, \{f,c,r\}} Pr(O_i) \times EV(O_i) \tag{2}$$

Program decision nodes: At decision node U, we can use a mixed policy as above (*Miximix*), or we can always take the maximum EV action for ourselves (*Miximax*), in which case:

$$EV(U) \;=\; \max(EV(U_f), EV(U_c), EV(U_r)) \tag{3}$$

Leaf nodes: Let L be a leaf node, P_{win} be the probability of winning the pot, $L_{\$pot}$ be the size of the pot, and $L_{\$cost}$ be the cost of reaching the leaf node (normally half of $L_{\$pot}$). At terminal nodes resulting from a fold, P_{win} is either zero (if we folded) or one (if the opponent folded), so the EV is simply the amount won or lost during the hand. The net EV of a showdown leaf node is:

$$EV(L) \;=\; (P_{win} \times L_{\$pot}) - L_{\$cost} \tag{4}$$

5 EV Calculation Example

For each showdown leaf node of the game tree, we store a histogram of the *hand rank* (HR, a percentile ranking between 0.0 and 1.0, broken into 20 cells with a range of 0.05 each) that the opponent has shown in previous games with that exact betting sequence. We will use 10-cell histograms in this section to simplify the explanation.

For example, suppose we are Player 1 (P1), the opponent is Player 2 (P2), and the pot contains four small bets (sb) on the final betting round. We bet (2 sb) and are faced with a raise from P2. We want to know what distribution of hands P2 would have in this particular situation. Suppose that a corresponding 10-cell histogram has relative weights of [1 1 0 0 0 0 0 4 4 0], like that shown in Figure 1. This means that based on our past experience, there is a 20% chance that P2's raise is a bluff (a hand in the HR range 0.0-0.2), and an 80% chance that P2 has a hand in the HR range 0.7-0.9 (but not higher).

The histogram for the showdown node after we re-raise and P2 calls (bRrC) will be related, probably having a shape like [0 0 0 0 0 0 0 5 5 0], because P2 will probably fold if he was bluffing, and call with all legitimate hands. The action frequency data we have on this opponent will be consistent, perhaps indicating that after we re-raise, P2 will fold 20% of the time, call 80%, and won't re-raise (because it is not profitable to do so). The probability triple of action frequencies is $Pr(F, C, R) = \{0.2, 0.8, 0.0\}$.

To decide what action to take in this situation, we compute the expected value for each choice: EV(fold), EV(call), and EV(raise). EV(fold) is easily determined from the betting history — the game will have cost us -4 small bets.

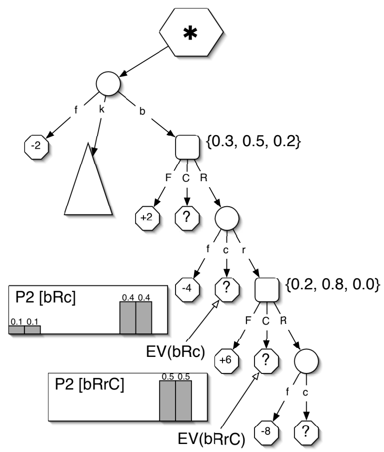

Fig. 1. Betting Tree for the EV Calculation Example

Table 1. Expected Values for call or raise for selected hand ranks.

HR	Pr(w|`c`)	EV(c)	Pr(w|`rC`)	EV(r)	Action
0.70	0.2	-3.6	0.0	-5.2	call
0.75	0.4	-1.2	0.25	-2.0	call
0.80	0.6	+1.2	0.5	+1.2	c or r
0.85	0.8	+3.6	0.75	+4.4	raise
0.90	1.0	+6.0	1.0	+7.6	raise

EV(call) depends on our probability of winning, which depends on the strength of our hand. If our hand rank is in the range 0.2-0.7, then we can only beat a bluff, and our chance of winning the showdown is $Pr(win|bRc) = 0.20$. Since the final pot will contain 12 sb, of which we have contributed 6 sb, the net EV(call) = -3.6 sb. Therefore, we would not fold a hand in the range 0.2-0.7, because we expect to lose less in the long run by calling (0.4 sb less).

If our hand rank is only HR = 0.1 (we were bluffing), then EV(call) = -4.8 sb, and we would be better off folding. If hand rank is HR = 0.8, then we can also beat half of P2's legitimate hands, yielding an expected profit of EV(call) = +1.2 sb. Table 1 gives the EV for calling with selected hand ranks in the range 0.7-0.9.

To calculate the expected value for re-raising, we must compute the weighted average of all cases in that subtree, namely: bRrF, bRrC, bRrRf, and bRrRc. Since the probability assigned to a P2 re-raise is zero, the two latter cases will not affect the overall EV and can be ignored. The share from bRrF is $0.2 \times 6 = +1.2$ sb, and is independent of our hand strength. The probability of winning after bRrC is determined by the histogram for that case, as before. Thus, in this example EV(raise) $= 1.2 + 0.8 \times (16 * Pr(win|rC) - 8)$, as shown in Table 1 for the same selected hand ranks.

As a consequence of this analysis, if we are playing a strictly maximizing strategy, we would decide to fold if our hand is weaker than HR = 0.167, call if it is in the range 0.167 to 0.80, and re-raise if it is stronger than HR = 0.80. Computing the viability of a possible bluff re-raise is done similarly.

6 Abstractions for Opponent Modeling

After each hand is played against a particular opponent, the observations made during that hand are used to update our opponent model. The action decisions made by the opponent are used to update the betting frequencies corresponding to the sequence of actions during the hand. When showdowns occur, the hand rank (HR) shown by the opponent is used to update a leaf node histogram, as illustrated in the previous section.

The *context tree* is an explicit representation of the imperfect information game tree, having the same skeletal structure with respect to decision nodes. Chance nodes in the tree are represented implicitly (all possible chance outcomes are accounted for during the EV calculation).

A leaf node of the context tree corresponds to all of the leaves of the game tree with the same betting sequence (regardless of the preceding chance nodes). Associated with this is an efficient data structure for maintaining the empirically observed action frequencies and showdown histograms for the opponent. For this we use a trie, based on the natural prefix structure of related betting sequences. Hash tables are used for low-overhead indexing.

6.1 Motivation for Multiple Abstractions

The *Miximax* and *Miximix* search algorithms perform the type of mathematical computation that underlies a theoretically correct decision procedure for poker. For the game of two-player Limit Texas Hold'em, there are $9^4 = 6561$ showdown nodes for each player, or 13122 leaf-level histograms to be maintained and considered. This fine level of granularity is desirable for distinguishing different contexts and ensuring a high correlation within each class of observations.

However, having so many distinct contexts also means that most betting sequences occur relatively rarely. As a result, many thousands of games may be required before enough data is collected to ensure reliable conclusions and effective learning. Moreover, by the time a sufficient number of observations have been made, the information may no longer be current.

This formulation alone is not adequate for practical poker. It is common for top human players to radically change their style of play many times over the course of a match. A worthy adversary will constantly use deception to disguise the strength of their hand, mask their intentions, and try to confuse our model of their overall strategy. To be effective, we need to accumulate knowledge very quickly, and have a preference toward more recent observations. Ideally, we would like to begin applying our experience (to some degree) immediately, and be basing decisions primarily on what we have learned over a scope of dozens or hundreds of recent hands, rather than many thousands. This must be an ongoing process, since we may need to keep up with a rapidly changing opponent.

Theoretically, this is a more challenging learning task than most of the problems studied in the machine learning and artificial intelligence literature. Unlike most Markov decision process (MDP) problems, we are not trying to determine a static property of the domain, but rather the dynamic characteristics of an adversarial opponent, where historical perspective is essential.

In order to give a preference toward more recent data, we gradually "forget" old observations using exponential history decay functions. Each time an observation is made in a given context, the previously accumulated data is diminished by a history decay factor, h, and the new data point is then added. Thus for $h = 0.95$, the most recent event accounts for 5% of the total weight, the last $1/(1-h) = 20$ observations account for $(1 - 1/e) = 0.63$ of the total, and so on.

6.2 Abstraction

In order to learn faster and base our inferences on more observations, we would like to combine contexts that we expect to have a high mutual correlation. This allows us to generalize the observations we have made, and apply that knowledge to other related situations. There are many possible ways of accomplishing these abstractions, and we will address only a few basic techniques.

An important consideration is how to handle the *zero frequency problem*, when there has yet to be any observations for a given context; and more generally, how to initialize the trie with good default data. Early versions of the system employed somewhat simplistic defaults, which resulted in rather unbalanced play early in a match. More recent implementations use default data based on rational play for both players, derived in a manner analogous to Nash equilibrium strategies.

The finest level of granularity is the context tree itself, where every possible betting sequence is distinct, and a different histogram is used for each. The opponent action frequencies are determined from the number of times each action was chosen at each decision node (again using a history decay factor to favour recent events). Unfortunately, having little data in each class will result in unreliable inferences.

One coarse-grained abstraction groups all betting sequences where the opponent made an equal number of bets and raises throughout the hand, ignoring what stage of the hand they were made. A finer-grained version of the same idea maintains an ordered pair for the number of bets and raises by each player.

However, testing reveals that an even courser-grained abstraction may be desirable. Summing the total number of raises by both players (no longer distinguishing which player initiated the action) yields only nine distinct classes. Despite the crudeness of this abstraction, the favorable effects of grouping the data is often more important than the lower expected correlations between those lines of play.

Another similar type of coarse-grained abstraction considers only the final size of the pot, adjusting the resolution (*ie.* the range of pot sizes) to provide whatever number of abstraction classes is desired.

An abstraction system can be hierarchical, in which case we also need to consider how much weight should be assigned to each tier of abstraction. This is based on the number of actual observations covered at each level, striving for an effective balance, which will vary depending on the opponent.

Our method of combining different abstraction classes is based on an exponential mixing parameter (say $m = 0.95$) as follows. Let the lowest-level context tree (no abstraction) be called A0, a fine-grained abstraction be called A1, a cruder amalgam of those classes be called A2, and the broadest classification level be called A3. Suppose the showdown situation in question has five data points that match the context exactly, in A0. This data is given a weight of $(1 - m^5) = 0.23$ of the total. If the next level of abstraction, A1, has 20 data points (including those from A0), it is assigned $(1 - m^{20}) = 0.64$ of the remaining weight, or about 50% of the total. The next abstraction level might cover 75 data points, and be given $(1 - m^{75}) = 0.98$ of the remainder, or 26% of the total. The small remaining weight is given to the crudest level of abstraction. Thus all levels contribute to the overall profile, depending on how relevant each is to the current situation.

7 Experiments

To evaluate the strength of VEXBOT, we conducted both computer *vs* human experiments, and a round-robin tournament of computer *vs* computer matches.

The field of computer opponents consisted of:

1) SPARBOT, the publicly available version of PSOPTI-4 [1], which surpassed all previous programs for two-player Limit Hold'em by a large margin [4].

2) POKI, a formula-based program that incorporates opponent modeling to adjust its hand evaluation. Although POKI is the strongest known program for the ten-player game, it was not designed to play the two-player game, and thus does not play that variation very well [8].

3) HOBBYBOT, a slowly adapting program written by a hobbyist, specifically designed to exploit POKI's flaws in the two-player game.

4) JAGBOT, a simple static formula-based program that plays a rational, but unadaptive game.

5) ALWAYS CALL and 6) ALWAYS RAISE, extreme cases of weak exploitable players, included as a simple benchmark.

The results of the computer *vs* computer matches are presented in Table 2. Each match consisted of at least 40,000 hands of poker. The outcomes are statistically significant, with a standard deviation of approximately ± 0.03 sb/hand.

[1] Available at www.cs.ualberta.ca/ games/.

Table 2. Computer *vs* computer matches (small bets per hand).

Program	Vexbot	Sparbot	Hobbot	Poki	Jagbot	A.Call	A.Raise
Vexbot		+0.052	+0.349	+0.601	+0.477	+1.042	+2.983
Sparbot	-0.052		+0.033	+0.093	+0.059	+0.474	+1.354
Hobbybot	-0.349	-0.033		+0.287	+0.099	+0.044	+0.463
Poki	-0.601	-0.093	-0.287		+0.149	+0.510	+2.139
Jagbot	-0.477	-0.059	-0.099	-0.149		+0.597	+1.599
Always Call	-1.042	-0.474	-0.044	-0.510	-0.597		=0.000
Always Raise	-2.983	-1.354	-0.463	-2.139	-1.599	=0.000	

Table 3. VEXBOT *vs* Human matches.

Num	Rating	sb/h	Hands
1	Expert	-0.022	3739
2	Intermediate	+0.136	1507
3	Intermediate	+0.440	821
4	Intermediate	+0.371	773

VEXBOT won every match it played, and had the largest margin of victory over each opponent. VEXBOT approaches the theoretical maximum exploitation against ALWAYS CALL and ALWAYS RAISE. No other programs came close to this level, despite those opponents being perfectly predictable.

Against SPARBOT, the strongest previous program for the two-player game, VEXBOT was able to find and exploit flaws in the pseudo-optimal strategy. The learning phase was much longer against SPARBOT than any other program, typically requiring several thousand hands. However, once an effective counter-strategy is discovered, VEXBOT will continue to win at that rate or higher, due to the oblivious nature of the game-theoretic player.

The testing against humans (Table 3) involves a smaller number of trials, and should therefore be taken only as anecdotal evidence (the outcomes being largely dominated by short-term swings in luck). Most humans players available for testing did not have the patience to play a statistically significant number of hands (especially when frustrated by losing). However, it is safe to say that VEXBOT easily exploited weaker players, and was competitive against expert level players. The results also consistently showed a marked increase in its win rate after the first 200-400 hands of the match, presumably due to the opponent-specific modeling coming into effect.

A more recent implementation of the *Miximax* algorithm was able to improve considerably on the VEXBOT results against computer opponents. That version defeated SPARBOT by +0.145 small bets per hand — three times the win rate of the previous version. [2] Moreover, its win rate against SPARBOT is comparable

[2] The improved version has not been described in detail in this paper because it is still in the process of being tested against quality human opposition.

to that of the most successful human player (the first author), and is more than three times the win rate achieved by the world-class player cited in [4].

Revised versions of both programs competed in the 2003 Computer Olympiad, with VEXBOT again dominating SPARBOT, winning the gold and silver medals respectively [20].

8 Conclusions and Future Work

Limit poker is primarily a game of mathematics and opponent modeling. We have built a program that "does the math" in order to make its decisions. As a result, many sophisticated poker strategies emerge without any explicit encoding of expert knowledge. The adaptive and exploitive nature of the program produces a much more dangerous opponent than is possible with a purely game-theoretic approach.

One limitation not yet adequately addressed is the need for effective defaults, to ensure that the program does not lose too much while learning about a new (unknown) opponent at the beginning of a match. If good default data is not easily derivable by direct means, there are several ways that existing programs can be combined to form hybrids that are less exploitable than any of the component programs in isolation.

With a smoothly adapting program and a good starting point, it may be possible to use self-play matches and automated machine learning to constantly refine the default data. In principle, that process could eventually approach a game-theoretic near-optimal default that is much closer to a true Nash equilibrium strategy than has been obtained to date.

The performance of VEXBOT can be improved further in numerous ways. While we believe that the modeling framework is theoretically sound, the parameter settings for the program could be improved considerably. Beyond that, there is a lot of room for improving the context tree abstractions, to obtain higher correlations among grouped sequences.

Only the two-player variant has been studied so far. Generalization of these techniques to handle the multi-player game should be more straight-forward than with other approaches, such as those using approximations for game-theoretic solutions.

Refinements to the architecture and algorithms described in this paper will undoubtedly produce increasingly strong computer players. It is our belief that these programs will have something to teach all human poker players, and that they will eventually surpass all human players in overall skill.

9 Acknowledgments

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta's Informatics Circle of Research Excellence (iCORE), and by the Alberta Ingenuity Center for Machine

Learning (AICML). The first author was supported in part by an Izaak Walton Killam Memorial Scholarship and by the Province of Alberta's Ralph Steinhauer Award of Distinction.

References

1. Horvitz, E., Breese, L., Heckerman, D., Hovel, D., Rommeke, K.: The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In: UAI. (1998) 256–265
2. Brusilovsky, P., Corbett, A., de Rosis, F., eds.: User Modeling 2003. Springer-Verlag (2003)
3. Weld, D., Anderson, C., Domingos, P., Etzioni, O., Lau, T., Gajos, K., Wolfman, S.: Automatically personalizing user interfaces. In: IJCAI. (2003) 1613–1619
4. Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T., Szafron, D.: Approximating game-theoretic optimal strategies for full-scale poker. In: IJCAI. (2003) 661–668
5. Jansen, P.: Using Knowledge about the Opponent in Game-Tree Search. PhD thesis, Computer Science, Carnegie-Mellon University (1992)
6. Carmel, D., Markovitch, S.: Opponent modeling in adversary search. In: AAAI. (1996) 120–125
7. Iida, H., Uiterwijk, J., van den Herik, J., Herschberg, I.: Potential applications of opponent-model search. ICCA Journal **16** (1993) 201–208
8. Billings, D., Davidson, A., Schaeffer, J., Szafron, D.: The challenge of poker. Artificial Intelligence **134** (2002) 201–240
9. Dahl, F.: A reinforcement learning algorithm to simplified two-player Texas Hold'em poker. In: ECML. (2001) 85–96
10. Korb, K., Nicholson, A., Jitnah, N.: Bayesian poker. In: UAI. (1999) 343–350
11. Findler, N.: Studies in machine cognition using the game of poker. CACM **20** (1977) 230–245
12. von Neumann, J., Morgenstern, O.: The Theory of Games and Economic Behavior. Princeton University Press (1944)
13. Kuhn, H.W.: A simplified two-person poker. Contributions to the Theory of Games **1** (1950) 97–103
14. Billings, D., Papp, D., Schaeffer, J., Szafron, D.: Opponent modeling in poker. In: AAAI. (1998) 493–499
15. Billings, D., Peña, L., Schaeffer, J., Szafron, D.: Using probabilistic knowledge and simulation to play poker. In: AAAI. (1999) 697–703
16. Koller, D., Pfeffer, A.: Representations and solutions for game-theoretic problems. Artificial Intelligence (1997) 167–215
17. Buro, M.: Solving the oshi-zumo game. In: Advances in Computer Games 10. (2004) 361–366
18. Billings, D.: The first international RoShamBo programming competition. International Computer Games Association Journal **23** (2000) 3–8, 42–50
19. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall (2003)
20. Billings, D.: Vexbot wins poker tournament. International Computer Games Association Journal **26** (2003) 281