A competitive Texas Hold'em poker player via automated abstraction and real-time equilibrium computation

Andrew Gilpin
Computer Science Department
Carnegie Mellon University
gilpin@cs.cmu.edu

Tuomas Sandholm
Computer Science Department
Carnegie Mellon University
sandholm@cs.cmu.edu

March 31, 2006

Abstract

We present our game theory-based heads-up Texas Hold'em poker player. To overcome the computational obstacles stemming from Texas Hold'em's gigantic game tree, our player employs *automated* abstraction techniques to reduce the complexity of the strategy computations. In addition to this state-space abstraction, our player uses round-based abstraction in conjunction with both offline and real-time equilibrium approximation. Texas Hold'em consists of four betting rounds. Our player solves a large linear program (offline) to compute strategies for the abstracted first and second rounds. After the second betting round, our player updates the probability of each possible hand based on the observed betting actions in the first two rounds as well as the revealed cards. Using these updated probabilities, our player computes in *real-time* an equilibrium approximation for the last two abstracted rounds. We demonstrate that our player, which does not directly incorporate any poker-specific expert knowledge, is competitive with leading poker-playing programs which do incorporate such domain-specific knowledge, as well as with advanced human players.

1 Introduction

In environments with more than one agent, the outcome of one agent may depend on the actions of the other agents. Consequently, in determining what action to take, an agent must consider the possible actions of the other agents. Game theory provides the mathematical foundation for explaining how rational agents should behave in such settings. Unfortunately, even in settings where game theory provides definitive guidance of an agent's optimal behavior, the computational problem of determining these strategies remains difficult. In this paper, we develop computational methods for applying game theory-based solutions to a large real-world game of imperfect information.

For sequential games with imperfect information, one could try to find an equilibrium using the normal (matrix) form, where every contingency plan of the agent is a pure strategy for the agent. Unfortunately (even if equivalent strategies are replaced by a single strategy [13]) this representation is generally exponential in the size of the game tree [24]. The *sequence form* is an alternative that results in a more compact representation [18, 11, 24]. For two-player zero-sum games, there is a polynomial-sized (in the size of the game tree) linear programming formulation based on the sequence form such that strategies for players 1 and 2 correspond to primal and dual variables. Thus, a minimax solution[1] for reasonably-sized two-player zero-sum games can be computed using this method [24, 11, 12].[2] That approach alone scales to games with around a million nodes [7], but Texas Hold'em with its 10^{18} nodes is way beyond the reach of that method. In this paper, we present techniques that allow us to approach the problem from a game-theoretic point of view, while mitigating the computational problems faced.

[1] Minimax solutions are robust in that there is no equilibrium selection problem: an agent's minimax strategy guarantees at least the agent's minimax value even if the opponent fails to play his minimax strategy. Throughout this paper, we are referring to a minimax solution when we use the term equilibrium.

[2] Recently this approach was extended to handle computing *sequential equilibria* as well [16].

1.1 Prior research on poker

Poker is an enormously popular card game played around the world. The 2005 World Series of Poker had over $103 million in total prize money, including $56 million for the main event. Increasingly, poker players compete in online casinos, and television stations regularly broadcast poker tournaments. Poker has been identified as an important research area in AI due to the uncertainty stemming from opponents' cards, opponents' future actions, and chance moves, among other reasons [5]. In this paper, we develop new techniques for constructing a poker-playing program.

Almost since the field's founding, game theory has been used to analyze different aspects of poker [14, 17, 1, 23, pp. 186–219]. However, this work was limited to tiny games that could be solved by hand. More recently, AI researchers have been applying the computational power of modern hardware to computing game theory-based strategies for larger games. Koller and Pfeffer (1997) determined solutions to poker games with up to 140,000 nodes using the sequence form and linear programming. For a medium-sized (3.1 billion nodes) variant of poker called Rhode Island Hold'em, game theory-based solutions have been developed using a lossy abstraction followed by linear programming [20], and recently optimal strategies for this game were determined using lossless automated abstraction followed by linear programming [7].

The problem of developing strong players for Texas Hold'em is much more challenging. The most notable game theory-based player for Texas Hold'em used expert-designed manual abstractions and is competitive with advanced human players [4]. It is available in the commercial product *Poker Academy Pro* as *Sparbot*.

In addition to game theory-based research, there has also been recent work in the area of *opponent modelling* in which a poker-playing program attempts to identify and exploit weaknesses in the opponents [22, 9, 3]. The most successful Texas Hold'em program from that line of research is *Vexbot*, also available in *Poker Academy Pro*.

Our player differs from the above in two important aspects. First, it does not directly incorporate any poker-specific domain knowledge. Instead, it analyzes the structure of the game tree and automatically determines appropriate abstractions. Unlike the prior approaches, ours 1) does not require expert effort, 2) does not suffer from errors that might stem from experts' biases and inability to accurately express their knowledge (of course, the algorithmically generated abstractions are not perfect either), and 3) yields better and better poker players as computing speed increases over time (because finer abstractions are automatically found and used; in the prior approaches an expert would have to be enlisted again to develop finer abstractions). Second, our player performs both offline and real-time equilibrium computation. *Sparbot* only performs offline computation, and *Vexbot* primarily performs real-time computation. Detailed offline computation allows our player to accurately evaluate strategic situations early in the game, while the real-time computation enables our player to perform computations that are focused on specific portions of the game tree, based on observed events, and thus allows more refined abstractions to be used in the later stages than if offline computation were used for the later stages (where the game tree has exploded to be enormously wide).

In our experimental results section, we present evidence to show that our player, which does not directly use any poker-specific domain knowledge, and which does not attempt to identify and exploit weaknesses in opponents, performs competitively against *Sparbot*, *Vexbot*, and advanced human players.

2 Rules of Texas Hold'em poker

There are many different variations of Texas Hold'em. One parameter is the number of players. As most prior work on poker, we focus on the setting with two players, called *heads-up*. Another difference between variations is the betting structure. Again, as most prior research, we focus on *low-limit* poker, in which the betting amounts adhere to a restricted format. (Other popular variants include *no-limit*, in which players may bet any amount up to their current bankroll, and *pot-limit*, in which players may bet any amount up to the current size of the pot.)

Before any cards are dealt, the first player, called the *small blind*, contributes one chip to the pot; the second player (*big blind*) contributes two chips.[3] Each player is dealt two *hole cards* from a randomly shuffled standard deck of 52 cards. Following the deal, the players participate in the first of four betting rounds, called the *pre-flop*. The small blind acts first; she may either call the big blind (contribute one chip), raise (three chips), or fold (zero chips). The players then alternate either calling the current bet, raising the bet by two chips, or folding. In the event of a fold, the folding

[3]The exact monetary value of a chip is irrelevant to our player and so we refer only to the quantity of chips.

player forfeits the game and the other player wins all of the chips in the pot. Once a player calls a bet, the betting round finishes. The number of raises allowed is limited to four in each round.

The second round is called the *flop*. Three *community cards* are dealt face-up, and a betting round takes place with bets equal to two chips. The big blind player is the first to act, and there are no blind bets placed in this round.

The third and fourth rounds are called the *turn* and the *river*. In each round, a single card is dealt face-up, and a betting round similar to the flop betting round takes place, but with bets equal to four chips.

If the betting in the river round ends with neither player folding, then the *showdown* takes place. Each player uses the seven cards available (their two hole cards along with the five community cards) to form the best five-card poker hand, where the hands are ranked in the usual order. The player with the best hand wins the pot; in the event of a tie, the players split the pot.

3 Strategy computation for the pre-flop and flop

Our player computes the strategies for the pre-flop and flop offline. There are two distinct phases to the computation: the automated abstraction and the equilibrium approximation. We discuss these in the following subsections, respectively.

3.1 Automated abstraction for the pre-flop and flop

For automatically computing a state-space abstraction for the first and second rounds, we use the *GameShrink* algorithm [7] which is designed for situations where the game tree is much too large for an equilibrium-finding algorithm to handle. *GameShrink* takes as input a description of the game, and outputs a smaller representation that approximates the original game. By computing an equilibrium for the smaller, abstracted game, one obtains an equilibrium approximation for the original game.

We control the coarseness of the abstraction that *GameShrink* computes by a threshold parameter. The abstraction can range from lossless (threshold = 0), which results in an equilibrium for the original game, to complete abstraction (threshold = ∞), which treats all nodes of the game as the same. The original method for using a threshold in *GameShrink* required a weighted bipartite matching computation (for heuristically determining whether two nodes are strategically similar) in an inner loop. To avoid that computational overhead, we use a faster heuristic. Letting w_1, l_1 and w_2, l_2 be the expected numbers of wins and losses (against a roll-out of every combination of remaining cards) for the two hands, we define two nodes to be in the same abstraction class if $|w_1 - w_2| + |l_1 - l_2| \leq$ threshold. We vary the abstraction threshold in order to find the finest-grained abstraction for which we are able to compute an equilibrium.

In the first betting round, there are $\binom{52}{2} = 1{,}326$ distinct possible hands. However, there are only 169 strategically different hands. For example, holding A♠A♣ is no different (in the pre-flop phase) than holding A♢A♡. Thus, any pair of Aces may be treated similarly.[4] *GameShrink automatically* discovers these abstractions.

In the second round, there are $\binom{52}{2}\binom{50}{3} = 25{,}989{,}600$ distinct possible hands. Again, many of these hands are strategically similar. However, applying *GameShrink* with the threshold set to zero results in a game which is still too large for an equilibrium-finding (LP) algorithm to handle. Thus we use a positive threshold that yields an abstraction that has 2,465 distinct hands.[5]

The finest abstraction that we are able to handle depends on the available hardware. As hardware advances become available, our algorithm will be able to immediately take advantage of the new computing power simply by specifying a different abstraction threshold as input to *GameShrink*. (In contrast, expert-designed abstractions have to be manually redesigned to get finer abstractions.)

To speed-up *GameShrink*, we precomputed several databases. First, a `handval` database was constructed. It has $\binom{52}{7} = 133{,}784{,}560$ entries. Each entry corresponds to seven cards and stores an encoding of the hand's rank, enabling

[4] This observation is well-known in poker, and in fact optimal strategies for pre-flop (1-round) Texas Hold'em have been computed using this observation [19].

[5] In order for our player to be able to consider such a wide range of hands in the flop round, we limit (in our players model, but not in the evaluation) the number of raises in the flop round to three instead of four. For a given abstraction, this results in a smaller linear program. Thus, we are able to use an abstraction with a larger number of distinct flop hands. This restriction was also used in the *Sparbot* player and has been justified by the observation that four raises rarely occurs in practice [4].

rapid comparisons to determine which of any two hands is better (ties are also possible). These comparisons are used in many places by our algorithms.

To compute an index into the `handval` database, we need a way of mapping 7 integers between 0 and 51 to a unique integer between 0 and $\binom{52}{7} - 1$. We do this using the *colexicographical ordering* of subsets of a fixed size [6] as follows. Let $\{c_1, \ldots, c_7\}$, $c_i \in \{0, \ldots, 51\}$, denote the 7 cards and assume that $c_i < c_{i+1}$. We compute a unique index for this set of cards as follows:

$$index(c_1, \ldots, c_7) = \sum_{i=1}^{7} \binom{c_i}{i}.$$

We use similar techniques for computing unique indices in the other databases.

Another database, `db5`, stores the expected number of wins and losses for five-card hands (the number of draws is inferred from this). This database has $\binom{52}{2}\binom{50}{3} = 25{,}989{,}600$ entries, each corresponding to a pair of hole cards along with a triple of flop cards. In computing the `db5` database, our algorithm makes heavy use of the `handval` database. The `db5` database is used to quickly compare how strategically similar a given pair of flop hands are. This enables *GameShrink* to run much faster, which allows us to compute and evaluate several different levels of abstraction.

By using the above precomputed databases, we are able to run *GameShrink* in about four hours for a given abstraction threshold. Being able to quickly run the abstraction computation allowed us to evaluate several different abstraction levels before settling on the most accurate abstraction for which we could compute an equilibrium approximation. After evaluating several abstraction thresholds, we settled on one that yielded an abstraction that kept all the 169 pre-flop hands distinct and had 2,465 classes of flop hands.

3.2 Equilibrium computation for the pre-flop and flop

Once we have computed an abstraction, we are ready to perform the equilibrium computation for that abstracted game. Two-person zero-sum games can be solved via linear programming using the sequence form representation of games. Building the linear program itself, however, is a non-trivial computation. It is desirable to be able to quickly perform this operation so that we can apply it to several different abstractions (as described above) in order to evaluate the capability of each abstraction, as well as to determine how difficult each of the resulting linear programs are to solve.

The difficulty in constructing the linear program lies primarily in computing the expected payoffs at the leaf nodes. Each leaf corresponds to two pairs of hole cards, three flop cards, as well as the betting history. Considering only the card history (the betting history is irrelevant for the purposes of computing the expected number of wins and losses), there are $\binom{52}{2}\binom{50}{2}\binom{48}{3} \approx 2.8 \cdot 10^{10}$ different histories. Evaluating each leaf requires rolling out the $\binom{45}{2} = 990$ possible turn and river cards. Thus, we would have to examine about $2.7 \cdot 10^{13}$ different combinations, which would make the LP construction slow (a projected 36 days on a 1.65 GHz CPU).

To speed up this LP creation, we precomputed a database, `db223`, that stores for each pair of hole cards, and for each flop, the expected number of wins for each player (losses and draws can be inferred from this). This database thus has

$$\frac{\binom{52}{2}\binom{50}{2}}{2}\binom{48}{3} = 14{,}047{,}378{,}800$$

entries. The compressed size of `db223` is 8.4 GB and it took about a month to compute. We store the database in one file per flop combination, and we only load into memory one file at a time, as needed. By using this database, our player can quickly and exactly determine the payoffs at each leaf for any abstraction. Once the abstraction is computed (as described in the previous subsection), we can build the LP itself in about an hour. This approach determines the payoffs *exactly*, and does not rely on any randomized sampling.

Using the abstraction described above yields a linear program with 243,938 rows, 244,107 columns, and 101,000,490 non-zeros. We solved the LP using the barrier method of ILOG CPLEX. This computation used 18.8 GB RAM and took 7 days, 3 hours. Our player uses the strategy computed in this way for the pre-flop and flop betting rounds. (Because our approximation does not involve any lossy abstraction on the pre-flop cards, we expect the resulting pre-flop strategies to be almost optimal, and certainly a better approximation than what has been provided in previous computations that only consider pre-flop actions [19].)

4 Strategy computation for the turn and river

Once the turn card is revealed, there are two betting rounds remaining. At this point, there are a wide number of histories that could have occurred in the first two rounds. There are 7 possible betting sequences that could have occurred in the pre-flop betting round, and 9 possible betting sequences that could have occurred in the flop betting round. In addition to the different betting histories, there are a number of different card histories that could have occurred. In particular, there are $\binom{52}{4} = 270{,}725$ different possibilities for the four community cards (three from the flop and one from the turn). The large number of histories makes computing an accurate equilibrium approximation for the final two rounds for every possible first and second round history prohibitively hard. Instead, our player computes in *real-time* an equilibrium approximation for the final two rounds based on the observed history for the current hand. This enables our player to perform computations that are focused on the specific remaining portion of the game tree, and thus allows more refined abstractions to be used in the later stages than if offline computation were used for the later stages (where the game tree has exploded to be enormously wide).

There are two parts to this real-time computation. First, our player must compute an abstraction to be used in the equilibrium approximation. Second, our player must actually compute the equilibrium approximation. These steps are similar to the two steps taken in the offline computation of the pre-flop and flop strategies, but the real-time nature of this computation poses additional challenges. We address each of these computations and how we overcame the challenges in the following two subsections.

4.1 Automated abstraction for the turn and river

The problem of computing abstractions for each of the possible histories is made easier by the following two observations: (1) the appropriate abstraction (even a theoretical lossless one) does not depend on the betting history (but does depend on the card history, of course); and (2) many of the community card histories are equivalent due to suit isomorphisms. For example, having 2♠3♠4♠5♠ on the board is equivalent to having 2♣3♣4♣5♣ as long as we simply relabel the suits of the hole cards and the (as of yet unknown) river card. Observation 2 reduces the number of abstractions that we need to compute (in principle, one for each of the $\binom{52}{4}$ flop and turn card histories, but reduced to 135,408).

Although *GameShrink* can compute one of these abstractions *for a given abstraction threshold* in just a few seconds, we perform these abstraction computations off-line for two reasons. First, since we are going to be playing in real-time, we want the strategy computation to be as fast as possible. Given a small fixed limit on deliberation time (say, 15 seconds), saving even a few seconds could lead to a major relative improvement in strategy quality. Second, we can set the abstraction threshold differently for each combination of community cards in order to capitalize on the finest abstraction for which the equilibrium can still be solved within a reasonable amount of time. One abstraction threshold may lead to a very coarse abstraction for one combination of community cards, while leading to a very fine abstraction for another combination. Thus, for each of the 135,408 cases, we perform several abstraction computations with different abstraction parameters in order to find an abstraction close to a target size (which we experimentally know the real-time equilibrium solver (LP solver) can solve (exactly or approximately) within a reasonable amount of time). Specifically, our algorithm first conducts binary search on the abstraction threshold for round 3 (the turn) until *GameShrink* yields an abstracted game with about 25 distinct hands for round 3. Our algorithm then conducts binary search on the abstraction threshold for round 4 (the river) until *GameShrink* yields an abstracted game with about 125 distinct hands for round 4. Given faster hardware, or more deliberation time, we could easily increase these two targets.

Using this procedure, we computed all 135,408 abstractions in about one month using six general-purpose CPUs.

4.2 Real-time equilibrium computation for the turn and river

Before we can construct the linear program for the turn and river betting rounds, we need to determine the probabilities of holding certain hands. At this point in the game the players have observed each other's actions leading up to this point. Each player action reveals some information about the type of hand the player might have.

Based on the strategies computed for the pre-flop and flop rounds, and based on the observed history, we apply Bayes' rule to estimate the probabilities of the different pairs of hole cards that the players might be holding. Letting

h denote the history, Θ denote the set of possible pairs of hole cards, and s_i denote the strategy of player i, we can derive the probability that player i holds hole card pair θ_i as follows:

$$\Pr[\theta_i \mid h, s_i] = \frac{\Pr[h \mid \theta_i, s_i]\Pr[\theta_i]}{\Pr[h \mid s_i]} = \frac{\Pr[h \mid \theta_i, s_i]\Pr[\theta_i]}{\sum_{\theta_i' \in \Theta} \Pr[h \mid \theta_i', s_i]}$$

Since we already know $\Pr[h \mid \theta_i, s_i]$ (we can simply look at the strategies, s_i, computed for the first two rounds), we can compute the probabilities above. (Of course, the resulting probabilities might not be exact because the strategies for the pre-flop and flop rounds might not constitute an exact equilibrium since, as discussed above, they were computed without considering a fourth possible raise on the flop or any betting in rounds 3 and 4, and abstraction was used.)

Once the turn card is dealt out, our program creates a separate thread to construct and solve the linear problem corresponding to the abstraction of the rest of that game. When it is time for our player to act, the LP solve is interrupted, and the current solution is accessed to get the strategy to use at the current time. When the algorithm is interrupted, we save the current basis which allows us to continue the LP solve from the point at which we were interrupted. The solve then continues in the separate thread (if it has not already found the optimal solution). In this way, our strategy (vector of probabilities) keeps improving in preparation for making future betting actions in rounds 3 and 4.

There are two different versions of the simplex algorithm for solving an LP: *primal simplex* and *dual simplex*. The primal simplex maintains primal feasibility, and searches for dual feasibility. (Once the primal and dual are both feasible, the solution is optimal.) Similarly, dual simplex maintains dual feasibility, and searches for primal feasibility. (Dual simplex can be thought of as running primal simplex on the dual LP.) When our player is playing as player 1, the dual variables correspond to our strategies. Thus, to ensure that at any point in the execution of the algorithm we have a feasible solution, our player uses dual simplex to perform the equilibrium approximation when she is player 1. Similarly, she uses the primal simplex algorithm when she is player 2. If the player is given an arbitrarily long time to deliberate, it would not matter which algorithm was used since at optimality both primal and dual solutions are feasible. But since we are also interested in interim solutions, it is important to always have feasibility for the solution vector in which we are interested. Our conditional choice of the primal or dual simplex method ensures exactly this.

One subtle issue is that our player occasionally runs off the equilibrium path. For example, suppose it is our player's turn to act, and the current LP solution indicates that she should bet; thus our player bets, and the LP solve continues. It is possible that as the LP solve continues, it determines that the best thing to have done would have been to check instead of betting. If the other player re-raises, then our player is in a precarious situation: the current LP solution is stating that she should not have bet in the first place, and consequently is not able to offer any guidance to the player since she is in an information set that is reached with probability zero. It is also possible for our player to determine during a hand whether the opponent has gone off of the equilibrium path, but this rarely happens because their cards are hidden. In these situations, our player simply calls the bet.

5 Experimental results

We tested our player against two of the strongest prior poker-playing programs, as well as against a range of humans.

5.1 Computer opponents

The first computer opponent we tested our player against was *Sparbot* [4]. *Sparbot* is also based on game theory. The main difference is that *Sparbot* considers three betting rounds at once (we consider two), but requires a much coarser abstraction. Also, all of *Sparbot's* computations are performed offline and it is hard-wired to never fold in the pre-flop betting round. Our results against *Sparbot* are illustrated in Figure 1 (left). When tested on 10,000 hands, we won 0.007 small bets per hand on average.

A well-known challenge is that comparing poker strategies requires a large number of hands in order to mitigate the role of luck. The variance of heads-up Texas Hold'em has been empirically observed to be $\pm 6/\sqrt{N}$ small bets per hand when N hands are played [2].[6] So, our victory over *Sparbot* is within the estimated variance of ± 0.06.

[6] One way to reduce the variance would be to play each hand twice (while swapping the players in between), and to fix the cards that are dealt.

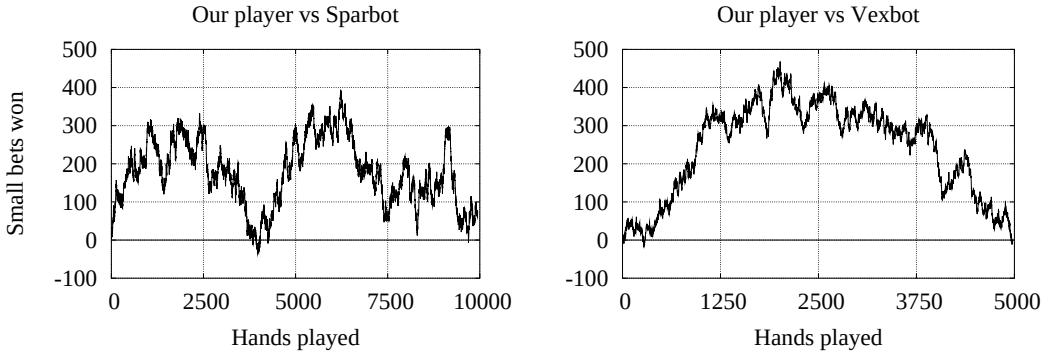

Figure 1: Our program versus *Sparbot* (left) and *Vexbot* (right).

The second computer opponent we played was *Vexbot* [3]. It searches the game tree, using a model of the opponent to estimate the probabilities of certain actions as well as the expected value of leaf nodes. It is designed to adapt to the particular weaknesses of the opponent, and thus, when facing a fixed strategy such as the one used by our player, it should gradually improve its strategy. Figure 1 (right) indicates that our player wins initially, but ends up in a tie after 5,000 hands. It is possible that *Vexbot* is learning an effective counter strategy to our player, although the learning process appears to take a few thousand hands.

When playing against computer opponents, our player was limited to 60 seconds of deliberation time, though it only used about 9 seconds on average. The results presented are what we had collected at submission time. We are continually running experiments to get more statistically meaningful results.

5.2 Human opponents

We also conducted experiments against human players, each of whom has considerable poker-playing experience. Each participant was asked to describe themselves as either "intermediate" or "expert". The experts play regularly and have significantly positive winnings in competitive play, mainly in online casinos. (In poker, unlike other competitive games such as chess, there is no ranking system for players.)

Our player was competitive with the humans (Table 1). However, due to the large variance present in poker, there does not appear to be strong evidence declaring it to be an overall winner or an overall loser. With human opponents it is difficult to play a large enough number of hands to make any definitive statements. Although our player ended up losing an average of 0.02 small bets per hand, this is well within the variance (± 0.15 small bets per hand when 1,576 hands are played). Interestingly, our player won 0.01 small bets per hand on average against the expert players.

When playing against human opponents, our player was limited to 15 seconds of deliberation time, though it only used about 4 seconds on average.

6 Other related research on abstraction

Abstraction techniques have been used in artificial intelligence research before. In contrast to our work, most (but not all) research involving abstraction has been for single-agent problems (*e.g.* [10, 15]). One of the first pieces of

This functionality is not available in *Poker Academy Pro*, and the opponent players are available only via that product, so we have as yet been unable to perform these experiments.

Even controlling for the deal of cards would not result in an entirely fair experiment for several reasons. First, the strategies used by the players are randomized, so even when the cards are held fixed, the outcome could possibly be different. Second, in the case where one of the opponents is doing opponent modeling, it may be the case that certain deals early in the experiment lend themselves to much better learning, while cards later in the experiment lend themselves to much better exploitation. Thus, the *order* in which the fixed hands are dealt matters. Third, controlling for cards would not help in experiments against humans, because they would know the cards that will be coming in the second repetition of a card sequence.

Player	small bets per hand	# hands
Intermediate player 1	0.20	71
Intermediate player 2	-0.09	166
Intermediate player 3	-0.40	100
Intermediate player 4	0.09	86
Expert player 1	-0.35	429
Expert player 2	0.19	325
Expert player 3	0.33	251
Expert player 4	0.09	148
Overall:	-0.02	1576

Table 1: Small bets per hand won by our player against humans.

research utilizing abstraction in multi-agent settings was the development of *partition search*, which is the algorithm behind GIB, the world's first expert-level computer bridge player [8]. In contrast to other game tree search algorithms which store a particular game position at each node of the search tree, partition search stores *groups* of positions that it determines are similar. (Typically, the similarity of two game positions is computed by ignoring the less important components of each game position and then checking whether the abstracted positions are similar—in some domain-specific sense—to each other.) Partition search can lead to substantial speed improvements over α-β-search. However, it is not game theory-based (it does not consider information sets in the game tree), and thus does not solve for the equilibrium of a game of imperfect information, such as poker.[7]

7 Conclusions

We presented a game theory-based heads-up Texas Hold'em poker player that was generated without any domain knowledge. To overcome the computational challenges posed by the huge game tree, we combined automated abstraction and real-time equilibrium approximation to develop our player. We compute strategies for the first two rounds of the game in a massive offline computation with abstraction followed by LP. For the last two rounds, our algorithm precomputes abstracted games of different granularity for the different card history equivalence classes. Also for the last two rounds, our algorithm deduces the probability distribution over the two players' hands from the strategies computed for the first two rounds and from both player's betting history. When round three actually begins, our algorithm performs an anytime real-time equilibrium approximation (using LP) that is focused on the relevant portion of the game tree (*i.e.*, on one of the abstracted games only) using the new prior.

Our program beat both of the prior state-of-the-art poker programs (although not with great statistical significance due to the variance in poker). This indicates that it is possible to build a poker program without any domain knowledge that is at least as strong as the best poker programs that were built using extensive domain knowledge. Our program is also competitive against experienced human players.

Future research includes developing additional techniques on top of the ones presented here, with the goal of developing even better programs for playing large sequential games of imperfect information.

References

[1] R. Bellman and D. Blackwell. Some two-person games involving bluffing. *Proc. of the National Academy of Sciences*, 35:600–605, 1949.

[2] D. Billings. Web posting at Poker Academy Forums, Meerkat API and AI Discussion, December 2005. http://www.poker-academy.com/forums/viewtopic.php?t=1872.

[7] Bridge is also a game of imperfect information, and partition search does not find the equilibrium for that game either. Instead, partition search is used in conjunction with statistical sampling to simulate the uncertainty in bridge. There are also other bridge programs that use search techniques for perfect information games in conjunction with statistical sampling and expert-defined abstraction [21]. Such (non-game-theoretic) techniques are unlikely to be competitive in poker because of the greater importance of information hiding and bluffing.

[3] D. Billings, M. Bowling, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Game tree search with adaptation in stochastic imperfect information games. In *Computers and Games*. Springer-Verlag, 2004.

[4] D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, 2003.

[5] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[6] B. Bollobás. *Combinatorics*. Cambridge University Press, 1986.

[7] A. Gilpin and T. Sandholm. Finding equilibria in large sequential games of imperfect information. Technical Report CMU-CS-05-158, Carnegie Mellon University, 2005.

[8] M. L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Stockholm, Sweden, 1999.

[9] B. Hoehn, F. Southey, R. C. Holte, and V. Bulitko. Effective short-term opponent exploitation in simplified poker. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 783–788, July 2005.

[10] C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[11] D. Koller, N. Megiddo, and B. von Stengel. Efficient computation of equilibria for extensive two-person games. *Games and Economic Behavior*, 14(2):247–259, 1996.

[12] D. Koller and A. Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94(1):167–215, July 1997.

[13] H. W. Kuhn. Extensive games. *Proc. of the National Academy of Sciences*, 36:570–576, 1950.

[14] H. W. Kuhn. A simplified two-person poker. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games*, volume 1 of *Annals of Mathematics Studies, 24*, pages 97–103. Princeton University Press, 1950.

[15] C.-L. Liu and M. Wellman. On state-space abstraction for anytime evaluation of Bayesian networks. *SIGART Bulletin*, 7(2):50–57, 1996. Special issue on Anytime Algorithms and Deliberation Scheduling.

[16] P. B. Miltersen and T. B. Sørensen. Computing sequential equilibria for two-player games. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 107–116, 2006.

[17] J. F. Nash and L. S. Shapley. A simple three-person poker game. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games*, volume 1, pages 105–116. Princeton University Press, 1950.

[18] I. Romanovskii. Reduction of a game with complete memory to a matrix game. *Soviet Mathematics*, 3:678–681, 1962.

[19] A. Selby. Optimal heads-up preflop poker, 1999. http://www.archduke.demon.co.uk/simplex/.

[20] J. Shi and M. Littman. Abstraction methods for game theoretic poker. In *Computers and Games*, pages 333–345. Springer-Verlag, 2001.

[21] S. J. J. Smith, D. S. Nau, and T. Throop. Computer bridge: A big win for AI planning. *AI Magazine*, 19(2):93–105, 1998.

[22] F. Southey, M. Bowling, B. Larson, C. Piccione, N. Burch, D. Billings, and C. Rayner. Bayes' bluff: Opponent modelling in poker. In *Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 550–558, July 2005.

[23] J. von Neumann and O. Morgenstern. *Theory of games and economic behavior*. Princeton University Press, 1947.

[24] B. von Stengel. Efficient computation of behavior strategies. *Games and Economic Behavior*, 14(2):220–246, 1996.