Abstract

We present a new automated abstraction algorithm for sequential imperfect information games. While most prior automated abstraction algorithms employ a myopic expected-value computation as a similarity metric, our algorithm considers a higher-dimensional space consisting of histograms over abstracted classes of states from later stages of the game. This enables our bottom-up abstraction algorithm to automatically take into account positive and negative potential. We further improve the abstraction quality by making multiple passes over the abstraction, enabling the algorithm to narrow the scope of analysis to information that is relevant given abstraction decisions made for earlier parts of the game. We also present a custom indexing scheme based on suit isomorphisms that enables one to work on significantly larger games than before.

We apply the techniques to heads-up limit Texas Hold'em poker. Whereas all prior game theory-based work for Texas Hold'em poker used generic off-the-shelf linear program solvers for the equilibrium analysis of the abstracted game, we make use of a recently developed algorithm based on the excessive gap technique from convex optimization. This paper is, to our knowledge, the first to abstract and game-theoretically analyze all four betting rounds in one run (rather than splitting the game into phases). The resulting player beats the prior leading poker programs (GS2, Sparbot, and Vexbot) with statistical significance.

1 Introduction

Automatically determining effective strategies in stochastic environments with hidden information is an important and difficult problem. In multiagent systems, the problem is exacerbated because the outcome for each agent depends on the strategies of the other agents. Poker games are well-defined environments exhibiting many challenging properties, including adversarial competition, uncertainty, and stochasticity, and have been identified as an important testbed for research on these topics [4]. Consequently, many researchers have chosen poker as an application area in which to test new techniques. In particular, heads-up limit Texas Hold'em poker has recently received a large amount of research attention, e.g., [4, 3, 2, 6, 8]. It can be modeled as a two-person zero-sum game.

From a strategic perspective, two-person zero-sum games are attractive because the set of Nash equilibria for these games are interchangeable and offer a guaranteed security level. The interchangeable property states that if \((x, y)\) is a Nash equilibrium (where \(x\) is player 1’s mixed strategy and \(y\) is player 2’s mixed strategy) and \((x', y')\) is a Nash equilibrium, then \((x, y')\) and \((x', y)\) are also Nash equilibria. Among other things, this eliminates the equilibrium selection problem which occurs in some games where there are multiple equilibria. The guaranteed security level means that by playing a Nash equilibrium strategy, a player is guaranteed a certain minimum expected payoff, regardless of the strategy used by the other player. In two-person zero-sum games, the value that one player can guarantee is the negative of what the other player can guarantee, so Nash equilibria are also called minimax solutions.
From a computational perspective, two-person zero-sum games have the benefit that Nash equilibria can be computed in polynomial time. In particular, the equilibrium problem can modeled as a linear program (LP) and solved using any off-the-shelf LP solver [12, 10, 14].

Although minimax strategies for two-person zero-sum games can be computed efficiently in theory, there are two important reasons why new techniques are still needed to enable the application of game theory to large problems, such as poker. The first is that the games themselves are huge. For example, heads-up limit Texas Hold’em has a game tree with around $10^{18}$ nodes. Even explicitly representing this game would require an enormous (impractical) amount of memory. The second reason is that even in cases where a game can be represented in memory (for example, after abstracting the game to find a smaller, almost equivalent representation), the LP solvers that are currently fastest for these problems (CPLEX’s interior-point method) require an amount of memory that is several orders of magnitude larger than the representation of the game [7].

Most existing approaches handle these two problems by abstraction and splitting the game into two phases. These cause strategic errors. Our player differs from prior approaches along both of these lines. First, the abstraction in the prior approaches is typically crafted manually [3] or by myopic algorithms [6, 8]. In this paper, we develop an automated non-myopic abstraction algorithm that addresses not only the winning probability but also the future potential of different states. Second, we do not split the game into phases; instead, we tackle the entire four-round model holistically. This is made possible by our use of the excessive gap technique [11], which was recently specialized to equilibrium finding in two-person zero-sum imperfect information games [9].

2 Rules of the game

There are many variations of poker. Like most prior work, we focus on two-player (heads-up) limit Texas Hold’em. The rules are as follows. The basic monetary unit is the small bet, which we treat as two chips. Before any cards are dealt, the first player (i.e., small blind) puts one chip into the pot, and the second player (i.e., big blind) puts two chips into the pot. There are four betting rounds. In the first, each player is dealt two cards, face down (these are called the hole cards). The small blind may either call the big blind (add one chip to the pot), raise (three chips), or fold (zero chips). The players then alternate either calling the current bet (contributing two chips), raising the bet (four chips), or folding (zero chips). In the event of a fold, the folding 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 is limited to four in each round. In the second, third, and fourth rounds, three, one, and one community cards are dealt face up, respectively. In each of these rounds the big blind acts first; betting proceeds as in the first round. The bets in the last two rounds are twice as large as in the first two rounds. If the final round ends with neither player folding, the player who forms the best five-card hand using any of his two cards and the five community cards wins the chips in the pot; in the event of a tie, the players split the pot.

3 Prior approaches to Texas Hold’em poker

There has been a recent surge of research into developing effective computer programs for playing heads-up limit Texas Hold’em. We now describe the leading prior approaches.

The first successful game theory-based player for Texas Hold’em was constructed by modeling the game as two phases. For each phase, domain experts manually designed a coarse abstraction, which was then solved as an LP using an interior-point method. The player is competitive with advanced human players [3]. A player based on the techniques is available in the commercial software package Poker Academy Pro as Sparbot.

Opponent modeling is a technique in which a program attempts to identify and exploit opponents’ weaknesses [2, 13]. This can be done by building a model for predicting opponents’ actions based on observations made throughout game play. The most successful Texas Hold’em program from that line of research is Vexbot [2]. It combines opponent modeling with minimax search (a variant of minimax search which allows the players to move probabilistically according to some model to account for the presence of imperfect information).

Recently, the game theory-based player GS1 was presented, which featured automated abstraction and real-time equilibrium approximation [6]. The automated abstraction algorithm used for that player was a simple approximation
version of the GameShrink algorithm [7]. GS1 is competitive with Sparbot and Vexbot, but there is no statistically significant evidence to demonstrate that it is better or worse than them. Recently, the authors of GS1 introduced an improved automated abstraction algorithm and a method for computing leaf payoffs of truncated games which led to the newer player GS2 [8], which was shown to be better than GS1 (by a statistically significant margin) and competitive with Sparbot and Vexbot.

We will now proceed to presenting our player.

4 Deciding the coarseness of the abstraction

Before computing an abstraction, we need to determine the coarseness of the resulting abstraction. Ideally we would compute an abstraction as fine-grained as possible. However, we need to limit the fineness of the abstraction to ensure that we are able to compute an equilibrium approximation for the resulting abstracted game.

One important aspect of the abstraction is the branching factor. One intuitively desirable property is to have an abstraction where the relative amount of information revealed in each stage is similar to the relative amount revealed in the game under consideration. For example, it would likely not be effective to have an abstraction that only had one bucket for each of the first three rounds, but had 1000 buckets for the last round. Similarly, we don’t want to have 100 buckets in the first round if we are going to only have 100 buckets in the second, third, and fourth rounds, since then no new information would be revealed after the first round.

One implication of this reasoning is that the branching factor going into the flop (where three cards are dealt) should be greater than the branching factor going into the turn or river (where only one card is dealt in each round). Furthermore, it seems reasonable to require that the branching factor of the flop be at least the branching factor of the turn and river combined, since more information is revealed on the flop than on the turn and river together.

Based on these considerations, and based on some preliminary experiments to determine the problem size we could expect our equilibrium-finding algorithm to handle, we settled on an abstraction that has 20 buckets in the first round, 800 buckets in the second round, 4,800 buckets in the third round, and 28,800 buckets in the fourth round. This implies a branching factor of 40 for the flop, 6 for the turn, and 6 for the river.

5 Potential-aware automated abstraction

The most successful prior approach to automated abstraction in sequential games of imperfect information was based on a myopic expected-value computation [8], and used k-means clustering with integer programming to compute the abstraction. A state of the game was evaluated according to the probability of winning the hand. The algorithm clustered together states with similar probabilities of winning, and it started computing the abstraction from the first round and then down through the card tree. This top-down algorithm generated the abstraction for GS2.

That approach does not take into account the potential of hands. For example, certain poker hands are considered drawing hands in which the hand is currently weak, but has a chance of becoming very strong. An important type of drawing hand is one in which the player has four cards of a certain suit (five are required to make a flush); at the present stage the hand is not very strong, but could become so if the required card showed up later in the game. Since the strength of such a hand could potentially turn out to be much different later in the game, it is generally accepted among poker experts that such a hand should be played differently than another hand with the same chance of winning, but without as much potential to improve. However, if using the difference between probabilities of winning as the metric for performing the clustering, the automated abstraction algorithm would consider these two very different situations to be quite similar.

One possible approach to handling the problem that certain hands with the same probability of winning may have different potential would be to consider not only the expected strength of a hand, but also its variance. In other words, the algorithm would be able to differentiate between two hands that have the same probability of winning, but where

---

1In the manual abstraction used in Sparbot, there are six buckets of hands where the hands are selected based on likelihood of winning and one extra bucket for hands that an expert considered to have high potential [3]. In contrast, our approach is automated, and does its bucketing holistically based on a multi-dimensional notion of potential (so it does not separate buckets into ones based on winning probability and ones based on potential). Furthermore, its abstraction is drastically finer grained.
one hand faces more uncertainty about what its final strength will be, while the other hand strength is unlikely to change much. Although this would likely improve the basic abstraction algorithm, it does not take into account the different paths of information revelation that hands take in increasing or decreasing in strength. For example, two hands could have similar means and variances, but one hand may be determined after one more step, while the other hand needs two more steps before its final strength is determined.

To address this, we introduce an approach where we associate with each state of the game a histogram over future possible states. This representation can encode all the pertinent information from the rest of the game, such as the probability of winning, the variance of winning, and the paths of information revelation. (In such a scheme, the k-means clustering step requires a distance function to measure the dissimilarity between different states. The metric we use in this paper is the usual \(L_2\)-distance metric. Given a finite set \(S\) of states where each hand \(i\) is associated with histogram \(h_i\) over the future possible states \(S\), the \(L_2\)-distance between hands \(i\) and \(j\) is

\[
\text{dist}(i, j) = \left(\sum_{s \in S} (h_i(s) - h_j(s))^2\right)^{\frac{1}{2}}.
\]

There are at least two prohibitive problems with the vanilla approach as stated. First, there are a huge number of possible reachable future states, so the dimensionality of the histograms is too large to do meaningful clustering with a reasonable number of clusters (i.e., small enough to lead to an abstracted game that can be solved for equilibrium). Second, for any two states at the same level of the game, the descendant states are disjoint. Thus the histograms would have non-overlapping supports, so any two states would have maximum dissimilarity and thus no basis for clustering.

For both of these reasons (and for reducing memory usage and enhancing speed), we coarsen the domains of the histograms. First, instead of having histograms over individual states, we use histograms over abstracted states (clusters), which contain a number of states each. We will have, for each cluster, a histogram over clusters later in the game. Second, we restrict the histogram of each cluster to be over clusters at the next level of the game tree only (rather than over clusters at all future levels). However, we introduce a technique (a bottom-up pass of constructing abstractions up the tree) that allows the clusters at the next level to capture information from all later levels.

One way of constructing the histograms would be to perform a bottom-up pass of the card deal tree: abstracting level four (i.e., betting round 4) first, creating histograms for level 3 nodes based on the level 4 clusters, then abstracting level 3, creating histograms for level 2 nodes based on the level 3 clusters, and so on. This is indeed what we do to find the abstraction for level 1.

However, for later betting rounds, we improve on this algorithm further by leveraging our knowledge of the fact that abstracted children of any cluster at the level above should only include states that can actually be children of the states in that cluster. We do this by multiple bottom-up passes, one for each cluster at the level above. For example, if a cluster at level 1 contains only those states where the hand consists of two Aces, then when we are doing abstraction for level 2, the bottom-up pass for that level-1 cluster should only consider future states where the hand contains two Aces as the hole cards. This enables the abstraction algorithm to narrow the scope of analysis to information that is relevant given the abstraction that it made for earlier levels. The following subsections describe our abstraction algorithm in detail.

### 5.1 Computing the abstraction for round 1

The first piece of the abstraction we computed was for the first round, i.e., the pre-flop. In this round we have a target of 20 buckets, out of the \(\binom{52}{2} = 1,326\) possible combinations of cards. As discussed above, we will have, for each pair of hole cards, a histogram over clusters of cards at level 2. (These clusters are not necessarily the same that we will eventually use in the abstraction for level 2, discussed later.)

To obtain the level-2 clusters, we perform a bottom-up pass of the card tree as follows. Starting with the fourth round, we cluster the \(\binom{52}{2}^{\binom{50}{4}} = 2,809,475,760\) hands into 5 clusters\(^2\) based on the probability of winning. Next, we consider the \(\binom{52}{2}^{\binom{50}{3}} = 305,377,800\) third-round hands. For each hand we compute its histogram over the 5 level-4 clusters we computed. Then, we perform k-means clustering on these histograms to identify 10 level-3 clusters. We repeat a similar procedure for the \(\binom{52}{2}^{\binom{50}{2}} = 25,989,600\) hands in the second round to identify 20 level-2 clusters.

---

\(^2\)For this algorithm, the number of clusters at each level (5, 10, 20) was chosen to reflect the fact that when clustering data, the number of clusters needed to represent meaningful information should be at least the level of dimensionality of the data. So, the number of clusters on level \(k\) should be at least as great as on level \(k+1\).
Using those level-2 clusters, we compute the 20-dimensional histograms for each of the \( \binom{52}{2} \binom{47}{5} = 1,326 \) possible hands at level 1 (i.e., in the first betting round). Then we perform \( k \)-means clustering on these histograms to obtain the 20 buckets that constitute our abstraction for the first betting round.

5.2 Computing the abstraction for rounds 2 and 3

Just as we did in computing the abstraction for the first round, we start by performing a bottom-up clustering, beginning in the fourth round. However, instead of doing this bottom-up pass once, we do it once for each bucket in the first round. Thus, instead of considering all \( \binom{52}{2} \binom{50}{5} = 2,809,475,760 \) hands in each pass, we only consider those hands which contain as the hole cards those pairs that exist in the particular first-round bucket we are looking at.

At this point we have, for each first-round bucket, a set of second-round clusters. For each first-round bucket, we have to determine how many child buckets it should actually have. For each first-round bucket, we run \( k \)-means clustering on its second-round clusters for \( k \in \{1..80\} \). (In other words, we are clustering those second-round clusters (i.e., data points) into \( k \) clusters.) This yields, for each first-round bucket and each value of \( k \), an error measure for that bucket assuming it will have \( k \) children. (The error is the sum of each data point’s \( L_2 \) distance from the centroid of its assigned cluster.)

Based on our design of the coarseness of the abstraction, we know that we have a total limit of 800 children (i.e., buckets at level 2) to be spread across the 20 first-round buckets. As in the abstraction algorithm used by GS2 [8], we formulate and solve an integer program (IP) to determine how many children each first-round bucket should have (i.e., what \( k \) should be for that bucket). The IP simply minimizes the sum of the errors of the level-1 buckets under the constraint that their \( k \)-values do not sum to more than 800. (The optimal \( k \)-value for different level-1 buckets varied between 18 and 51.) This determines the final bucketing for the second betting round.

The bucketing for the third betting round is computed analogously. We use level-2 buckets as the starting point (instead of level-1 buckets), and in the integer program we allow a total of 4,800 buckets for the third betting round. (The optimal \( k \)-value for different level-2 buckets varied between 1 and 10.)

5.3 Computing the abstraction for round 4

In round 4 there is no need to use the sophisticated clustering techniques discussed above since the players will not receive any more information. Instead, we simply compute the fourth-round abstraction based on each hand’s probability of winning, exactly the way as was done for computing the abstraction for GS2 [8]. Specifically, for each third-round bucket, we consider all possible rollouts of the fourth round. Each of them constitutes a data point (whose value is computed as the probability of winning plus half the probability of tying), and we run \( k \)-means clustering on them for \( k \in \{1..18\} \). (The optimal \( k \)-value for different level-3 buckets varied between 1 and 14.) The error, for each third-round bucket and each \( k \), is the sum over the bucket’s data points, of the data point’s \( L_2 \) distance from the centroid of its cluster.

Finally, we run an IP to decide the \( k \) for each third-round bucket, with the objective of minimizing the sum of the third-round buckets’ errors under the constraint that the sum of those buckets’ \( k \)-values does not exceed 28,800 (which is the number of buckets allowed for the fourth betting round, as discussed earlier). This determines the final bucketing for the fourth betting round.\(^3\)

6 Exploiting suit isomorphisms

We now introduce ways in which we exploit suit isomorphisms. We first discuss a custom indexing scheme which dramatically reduces the space requirements of representing the abstraction. In the subsection after that, we present a way to exploit suit isomorphisms to speed up a key computation.

\(^3\)As discussed, our overall technique optimizes the abstraction one betting round at a time. A better abstraction could conceivably be obtained by optimizing all rounds together. However, that seems infeasible. First, the optimization problem would be nonlinear because the probabilities at a given level depend on the abstraction at all previous levels of the tree. Second, the number of decision variables in the problem would be exponential in the size of the card tree (even if the number of abstraction classes for each level is fixed). Third, one would have to solve a \( k \)-means clustering problem for each of those variables to determine its coefficient in the optimization problem.
6.1 Indexing for efficient abstraction representation

One challenge that is especially difficult when using a four-round model is that the number of distinct hands a player can face is huge. Our algorithm requires an integer index for each distinct hand in order to perform the lookup to see which abstracted bucket the given hand belongs to. The number of distinct hands, \((\binom{52}{2}) \cdot \binom{3}{1} \cdot \binom{4}{1} \cdot \binom{4}{1} \approx 5.6 \cdot 10^{10}\), is an order of magnitude too big to give each hand a unique index. For example, encoding the bucket for each hand using two bytes requires more than 104 gigabytes of storage. This would severely limit the practicality of the approach, since this storage is also required by our player at run-time.

We therefore introduce a more compact representation of the abstraction that capitalizes on a canonical representation of each hand based on suit symmetries. (This technique is valid since the rules of poker state that all suits are equally strong.) These canonical representations are computed using permutations (total orderings of the suits) and partial permutations (partial orderings of the suits), as we will describe later in this section.

The best size reduction one could hope for with this approach is a factor of \(4! = 24\), since we can map any permutation of the four suits to the same canonical hand. That this is not fully achievable is due to the fact that some hands are unaffected by some of the permutations of the suits, e.g. \(4♠4♥\) is equivalent to \(4♥4♠\), in which case there are less than 24 distinct hands mapping to the same canonical one. We call this phenomenon self symmetry.

Our approach uses the following concept. The colexicographical index [5] of a set of integers \(x = \{x_1 \ldots x_k\} \subset \{0..n - 1\}\), with \(x_i < x_j\) whenever \(i < j\), is \(colex(x) = \sum_{i=1}^{k} \binom{x_i}{i}\). This index has the important property that for a given \(n\), each of the \(\binom{n}{k}\) sets of size \(k\) has a distinct colexicographical index. Furthermore, these indices are compactly encoded as the integers from 0 to \(\binom{n}{k} - 1\).

We need to compute indices for hands from each of the four rounds. We compute these indices incrementally, using the index from round \(i\) to compute the index in round \(i + 1\). This approach gradually computes the permutations that map the given hand to its canonical representation. This incremental computation is useful both for providing a convenient way of computing the indices and for speeding up the index computation.

The index for the first round is computed, of course, using only the hole cards. If they are of the same suit, e.g. \(A♣7♠\), that suit is named “suit 1”, and we get the partial permutation \(♣ < \{♣, ♥, ♦\}\). If they are of different suits and different values, e.g. \(A♠7♥\), we name the suit of the card with the highest value “suit 1” and the other “suit 2”, resulting in the partial permutation \(♠ < ♥ < \{♥, ♦\}\). Lastly, if they have the same value, e.g. \(7♥7♥\), the hand is self symmetric, and we have the partial permutation \(\{♥\} < \{♥, ♦\}\). (At this point it is unspecified which of \(♥\) and \(♦\) is “suit 1” and which is “suit 2”.)

The later rounds also give rise to partial permutations, which are then used to refine the permutation of suits that were undecided in previous rounds. For instance if the hole cards are \(7♥7♥\) and the flop is \(3♥J♥A♥\), we refine the partial permutation \(\{♥\} < \{♥, ♦\}\) with \(♥ < ♦ < \{♣, ♠\}\) to get \(\{♥ < ♦\} < \{♣ < ♥\}\), i.e., \(♥ < ♦ < ♥ < ♦\).

Then, to compute the index from the (perhaps perhaps partial) permutation, our algorithm uses a case analysis which has far too many cases (60) to describe here. As an example, if the hole cards are \(7♥7♥\) and the flop is \(3♥J♥A♥\), then the analysis is in the category of one new suit in two cards and one old suit in a single card, breaking the self symmetry from the previous round. In this case the card with the old suit (♥) only has 12 possible canonical values (even though there are 24 ♠s and ♦s left in the deck), since no matter whether that new card would have been a ♠ or a ♥, its suit will now have become “suit 1”. In the same way, the other two cards only have \(\binom{13}{2}\) possible canonical values, since their suit will now have become “suit 3” no matter whether it is ♦ or ♣. Thus this case has \(\binom{13}{2} \cdot 12 = 936\) canonical hands representing four times that many actual hands, all of which share \(7♥7♥\) as the hole cards.

Each of the cases simply breaks the hand up into sets to be encoded with colexicographical indexing. In the case above, the sets are \{1,9\} and \{11\} (index 11 is the highest of the 12 cards in the group). Here, 1 corresponds to the three, 9 corresponds to the Jack, and 11 corresponds to an Ace. The index within this case is then computed as \(colex\{1,9\} \cdot 12 + colex\{11\}\). This is then combined with the index within the case of the first round. There are 13 possible pairs, and our sevens have index 5. We thus get \((colex\{1,9\} \cdot 12 + colex\{11\}) \cdot 13 + 5\). Finally, to get the index, this is added to a global offset associated with this particular case.

With this indexing scheme, the memory consumption of the index reduces by a factor of 23.35 (which is close to the optimistic upper bound of 24).
6.2 Using symmetries to speed up 9-card rollout

For each pair of buckets in the fourth round, we need to compute the expected number of wins, losses, and draws for hands randomly drawn from those buckets. The straightforward approach of generating all \( \binom{52}{2} \binom{50}{2} \binom{48}{3} \binom{45}{1} \binom{44}{1} \approx 5.56 \cdot 10^{13} \) possible ways the cards can be dealt would require more than a month of CPU time. Furthermore, this computation would have to be started from scratch when we consider a new, different abstraction. But since we know that the indexing scheme will do the “suit renaming” anyway, we are able to just generate cards for all possible indices, with a weight indicating how many symmetric situations the current cards are representing. Furthermore, we use the fact that the two sets of hole cards are symmetric to only generate those where Player 1’s cards are “less” than Player 2’s cards, using an arbitrary ordering of the hole cards. Doing all this gives us more than a factor 44 speed up of the 9-card rollout, bringing it down to less than a day.

7 Computing equilibrium strategies for the holistic abstracted four-round model

Once the abstraction has been computed, the difficult problem of computing equilibrium strategies for the abstracted game remains. The existing game-theory based players (GS1, GS2, and Sparbot) computed strategies by first splitting the game into two phases, and then solving the phases separately and then gluing together the separate solutions. In particular, GS1 considers rounds 1 and 2 in the first phase, and rounds 3 and 4 in the second phase. GS2 considers round 1, 2, and 3 in the first phase, and rounds 3 and 4 in the second phase. Sparbot considers rounds 1, 2, and 3 in the first phase, and rounds 2, 3, and 4 in the second phase. These approaches allow for finer-grained abstractions than what would be possible if a single, monolithic four-round model were used. However, the equilibrium finding algorithm used in each of those players was based on standard algorithms for LP which do not scale to a four-round model (except possibly for a trivially coarse abstraction).

Solving the (two) different phases separately causes important strategic errors in the player (in addition to those caused by lossy abstraction). First, it will play the first phase of the game inaccurately because it does not properly consider the later stages (the second phase) when determining the strategy for the first phase of the game. Second, it does not accurately play the second phase of the game because strategies for the second phase are derived based on beliefs at the end of the first phase, which are inaccurate.

Therefore, we want to solve for equilibrium while keeping the game in one holistic phase. To our knowledge, this is the first time this has been done. Using a holistic four-round model makes the equilibrium computation a difficult problem, particularly since our abstraction is very fine grained. As noted earlier, standard LP solvers (like CPLEX’s simplex method and CPLEX’s interior-point method) are insufficient for solving such a large problem. Instead, we used an implementation of Nesterov’s excessive gap technique algorithm [11], which was recently specialized for two-person zero-sum sequential games of imperfect information [9]. This algorithm is a gradient-based algorithm which requires \( O(1/\epsilon) \) iterations to compute an \( \epsilon \)-equilibrium, that is, a strategy for each player such that his incentive to deviate to another strategy is at most \( \epsilon \). This algorithm is an anytime algorithm since at every iteration it has a pair of feasible solutions, and the \( \epsilon \) does not have to be fixed in advance. After 24 days of computing on 4 CPUs running in parallel, the algorithm had produced a pair of strategies with \( \epsilon = 0.027 \) small bets.

8 Experiments

We tested our player against the leading prior poker programs: GS2, Sparbot, and Vexbot. For the learning opponent, Vexbot, we allowed it to learn throughout the 100,000 hands (rather than flushing its memory every so often as is customary in computer poker competitions).

Our player outperformed all three players, by statistically significant margins. Table 1 summarizes our results. 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 [1]. This value (for the actual number of hands played) is displayed in the last column. A win rate greater than this value indicates statistical significance.
To show how many hands it takes to get statistical significance, Figure 1 plots our player’s bankroll and the statistical significance curve $\frac{6}{\sqrt{N}}$ as a function of the number of hands played. Against GS2 and Vexbot, our player has a statistically significant margin after just a couple of thousand hands. It took about 30,000 hands to demonstrate a statistically significant victory over Sparbot.

Figure 1: Performance against GS2, Sparbot, Vexbot. Dashed curves indicate level above which there is statistical significance.

9 Conclusions and future research

We presented a potential-aware automated abstraction technique. It applies to a broad range of sequential imperfect information games. We also presented a custom indexing scheme based on suit isomorphisms that enables one to work on significantly larger games than was possible before.

We applied these to Texas Hold’em, and solved the abstracted game using a variant of the excessive gap technique. This is, to our knowledge, the first time that all four betting rounds have been abstracted and game-theoretically analyzed in one run (rather than splitting the game into phases). The resulting player beats the prior leading poker programs (GS2 [8], Sparbot [3], and Vexbot [2]) with statistical significance.

In the future, we would like to prove how close to optimal our program is, and to experiment with the tradeoff of finer abstraction versus quality (gap) in equilibrium solving.

References


