University of Alberta

Library Release Form

Name of Author: Aaron Davidson

Title of Thesis: Opponent Modeling in Poker: Learning and Acting in a Hostile and Uncertain Environment

Degree: Master of Science

Year this Degree Granted: 2002

Aaron Davidson
502-11111 87th Ave
Edmonton, AB, Canada, T6G 0X9

Date: ________________

all your base are belong to us

University of Alberta

Opponent Modeling in Poker: Learning and Acting in a Hostile and Uncertain Environment

by

Aaron Davidson

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Summer 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Opponent Modeling in Poker: Learning and Acting in a Hostile and Uncertain Environment** submitted by Aaron Davidson in partial fulfillment of the requirements for the degree of **Master of Science**.

Jonathan Schaeffer
Supervisor

Duane Szafron
Supervisor

Robert Holte

Witold Pedrycz
External Examiner

Date: ________________

Abstract

Artificial intelligence research has had great success in many classic games such as chess, checkers, and othello. In these perfect-information domains, alpha-beta search is sufficient to achieve a high level of play. However Artificial intelligence research has long been criticized for focusing on deterministic domains of perfect information – many problems in the real world exhibit properties of imperfect or incomplete information and non-determinism.

Poker, the archetypal game studied by game-theorists, is a challenging domain containing both imperfect information and non-deterministic elements. The random shuffling of cards provides a great deal of uncertainty. The cards held by an opponent or in the remaining deck provide the imperfect information. A poker player has only a small incomplete description of the entire game-state. Furthermore, opponents play deceptively in order to hide information about the cards they hold. These properties prevent traditional game-tree search methods from working. In order to build a high-performance poker playing agent, a multitude of difficult problems must be solved in machine learning, search and simulation, and belief management.

Poki, a state-of-the-art poker playing program is discussed, with focus on the problem of opponent modeling. Several methods of opponent modeling are evaluated, including neural networks and decision trees. *Poki* can currently play poker at the level of an average human player.

We introduce a new method of directly searching imperfect information game trees, called *miximax*. Like minimax search, we do a search of the game-tree. Since the objective best moves of the opponent cannot be determined (we do not know their cards) we cannot choose a best move in the search. Instead, we use our knowledge of the opponent to choose their mixed strategy. Instead of using a *min*, we use a *mix* of their choices. The result is a reliable estimate of the expected values of each of our choices.

Acknowledgements

There are so many people to thank, so lets get this party started!

First of all, Darse Billings – if a thesis could have a co-author, it would be you. Your support, ideas, insight and friendship has been invaluable. Thanks of course, to my supervisors Jonathan Schaeffer and Duane Szafron. Thanks for hiring me to work with the Poker Group, three years ago – it was an excellent opportunity, a challenging project, and a terrific journey. Thanks to Lourdes Peña and Denis Papp for their blood sweat and tears laying the ground work in *Loki*.

Thank you Christine Thesen, for supporting me lovingly, through all of this hard work. And thank you Vega for being such a cute kitty cat. Thanks Jason Spencer for being a great friend and a good guy to bounce ideas off of. Thanks to my parents Bill & Maion Davidson, for being great, supportive parents.

And to everyone else that has put up with me – friends and family and collegues, thanks. Oh! And thanks to all the people who helped test *Poki* by playing it online for countless hours.

Did I mention thanks?

Contents

List of Figures

List of Tables

Chapter 1

Introduction

My work is a game, a very serious game.

– M. C. Escher (1898-1972)

Games are a natural choice for artificial intelligence (AI) research. They present unique challenges that require complicated problem solving to overcome. Games typically have well defined rules and definitions. Consequently, game states and transitions are usually easy to represent in a symbolic computer program. By definition, games have a built-in performance measure. This facilitates the quantification of a player's strength. Because of these features, games have been a main focus of artificial intelligence research since its inception [50]. The example that would naturally come to mind for most people is that of chess – a game that has had a long history in AI research [28]. Another classic example is Samuel's checkers program that used machine learning to achieve an average level of play [39].

Artificial intelligence research has had great success in many classic games such as chess [10], checkers [40], and Othello [9]. AI for these games has matured to the point that they can play at the level of strong grandmasters, and beyond. Using game tree search algorithms, programs have challenged and defeated human world champions in these games. These are deterministic domains of perfect information and their difficulty lies in the huge size of the search space. Computers excel at searching deeply into these game trees, which have a moderate branching factor. Recursive minimax search is ideally suited to these well-defined, noise-free domains. In Table 1.1, these types of games fall into a single category – deterministic games of perfect information. Until recently, games in the other three categories have remained largely unresearched.

The late 1990's have seen research progress in several games of imperfect information. Tesauro's backgammon program, *TD-Gammon*, which uses temporal difference learning, has had success at overcoming the non-deterministic aspect of

	deterministic	**non-deterministic**
perfect information	chess, checkers, go	backgammon
imperfect information	battleship, roshambo	poker, bridge, scrable

Table 1.1: Examples of four different types of games.

the game [48, 49]. Sheppard's *Maven* program for playing Scrabble is stronger than the best humans [43]. Ginsberg has produced a bridge program (*GIB*) that is world-champion caliber at declarer play (but not at bidding or defense) [20]. All of these programs have had to use new, innovative techniques to achieve high-performance in these imperfect-information or non-deterministic domains.

Presumably, the ultimate goal for much of artificial intelligence research is to develop useful systems that can adaptively make intelligent decisions in large uncertain domains, much like the world we inhabit. Artificial intelligence research has often been criticized for being too narrowly focused on systems that are only of use in a 'perfect' world. Techniques that work in small discrete domains often flounder in real-world situations. Reality is inherently noisy and unpredictable. There are no oracles. The world does not give away information for free.

1.1 Poker as a Testbed for AI

Continuing in the tradition of using games as a testbed for artificial intelligence, poker is an excellent game to explore decision making under conditions of imperfect information [3]. The poker domain contains a wide variety of challenges to standard learning and decision-making algorithms. The domain is inherently noisy and uncertain. Poker has high variance in the outcomes due to the *non-determinism* (the random shuffling of the deck), and *imperfect information* (the unknown cards held by the opponent). Furthermore, if a player folds, we *never* know what cards they were holding. These factors impose a large hurdle for reinforcement learning algorithms, since it is often difficult to determine if an action was positive or negative.

To complicate matters, poker games are normally played at a fast pace (typically lasting only a few minutes per hand), so players must make each decision in at most a few seconds.

All reasoning in poker must be probabilistic, as things are rarely ever certain. The cumulative sum of a series of games matter more than any individual game. In addition to the probabilistic aspects of poker, there are also psychological elements to address in observing and determining how our opponents play the game. Opponents are often deceptive, deliberately playing in ways which are difficult to learn and model.

In order to play strong poker, one must be able to understand the mathematics and strategy of the game, as well as understand the opponent's understanding of the game (and further, the opponent's perception of us, and so on...). A large part of the allure of using poker as a testbed for artificial intelligence research is the human aspect of the game. A crucial part of a poker game is modeling the opponent – trying to learn how they think, how they believe we think, and determining what tactics we can use to take advantage of this knowledge.

Artificial intelligence has long been criticized for working in small toy domains of perfect knowledge [14]. Poker has properties that are ideal for exploring many areas of AI that have traditionally been avoided or ignored.

1.2 Texas Hold'em Poker

Texas Hold'em is a common variant of poker played in casinos and championship poker tournaments. It has a high skill-to-luck ratio, compared to other types of poker, and it is known to be one of the most strategically rich variants of poker. The game is played with a standard deck of fifty two unique playing cards with four suits (clubs ($\clubsuit$), diamonds ($\diamondsuit$), hearts ($\heartsuit$), spades ($\spadesuit$)), and thirteen ranks (in increasing value: deuce (2), trey (3), four (4), five (5), six (6), seven (7), eight (8), nine (9), ten (T), jack (J), queen (Q), king (K), ace (A)).

Anywhere from two to twelve players can play (ten is most common in casinos), making for a dynamic game with a wide range of situations. Each player is dealt a hand consisting of two private *hole cards*. Once each player is dealt their cards, a round of betting ensues. The first two players after the dealer each pay a forced bet called the *blind* (instead of an *ante*) to start the betting. The first blind is half the minimum bet size, and is referred to as the *small blind*. The next player, the *big blind*, pays a full forced bet. During a round of betting players may either fold their hand, call the current bet, or raise the bet.

After the first betting round, called the *pre-flop* stage, three community *board* cards are dealt. This stage is called the *flop*. After another round of betting, a fourth board card is dealt called the *turn*, and is followed by another round of betting. Finally a fifth board card is dealt, the *river*, and we enter the last betting round.

Any players remaining in the hand when the betting is finished must go to a *showdown*. In the showdown, players must show their hole cards to the table. The player with the best five card poker hand (see Table 1.2) made from the best combination of the two hole cards and the five board cards wins the money in the *pot* (if there is a tie the money is divided amongst the winners). A player may win the pot uncontested if all of the opponents fold before the showdown, in which case the winner is not required to reveal his or her cards.

In *limit* Hold'em, the bet increment sizes are pre-determined. The limit is defined by a bet increment size called a *small bet*. During the pre-flop and flop betting

Hand Type	Example	Frequency
Straight Flush	Q♢ J♢ T♢ 9♢ 8♢	0.0015%
Four of a Kind	5♡ 5♣ 5♢ 5♠ K♣	0.0240%
Full House	5♡ 5♣ 5♢ K♠ K♣	0.1441%
Flush	2♣ 3♣ A♣ J♣ 7♣	0.1965%
Straight	5♢ 6♠ 7♢ 8♡ 9♢	0.3925%
Three of a Kind	9♢ 9♣ 9♡ 2♡ 6♢	2.1129%
Two Pair	A♡ A♣ J♡ J♠ 4♢	4.7539%
One Pair	J♣ J♠ 3♡ 6♠ K♢	42.257%
High Card	Q♣ 6♡ 3♡ T♢ 4♢	49.882%

Table 1.2: Poker hand type frequencies. There are $\binom{52}{5} = 2{,}598{,}960$ different five-card poker hands. The probability of being dealt a particular type of hand is shown above.

rounds, all bets are equal to one small bet. For the last two betting rounds this bet size doubles to a *big bet* (two small bets). For example, a game with a small bet of \$10 would be called a \$10/\$20 Hold'em game.

A more complex version of Texas Hold'em is *no-limit*, where players may bet as large an amount as they wish. Due to the high variability of betting and increased strategy, we do not currently consider no-limit Texas Hold'em. Limit Hold'em should be well understood before working on this more difficult version of the game.

1.2.1 An Example Hand of Texas Hold'em

In order to give a clear example of how the game is played, and to illustrate some of the strategy, the following example hand of Texas Hold'em is given as a guide. While no single hand of poker can exemplify all of the strategy and situations that occur in the game, this example gives a flavour for how it is played, and is intended for readers who are unfamiliar with the game.[1]

Suppose we (*Poki*) are playing in a five player game of Texas Hold'em, with a \$10/\$20 limit. We are the *button*, which means we are the last to act in each round of betting. Being last is advantageous, since we get to see the actions of all the other players before making our betting decision. Conversely, the player first to act, a position known as *under the gun*, has a disadvantage. *Poki* will play against four opponents, who we will refer to as Alice, Bill, Chris, and Dave. Alice is the small blind.

Alice is the player after the button, so she posts the \$5 small blind and Bill posts the \$10 big blind. Each player is dealt two private *hole* cards. *Poki* is dealt T♠-9♠. The first player to act, Chris, calls the big blind for \$10, and the next player, Dave, folds his hand. It is now our turn. Since we have a *suited-connector* hand (two cards

[1] Throughout this thesis, poker terminology printed in *italics* is described in the glossary.

Player	Preflop		Flop	Turn	River	Net Gain
Alice	Blind $5	Raise $10	Bet $10	Bet $20	Bet $20	+$110
Bill	Blind $10	Fold				-$10
Chris	Call $10	Call $10	Call $10	Fold		-$30
Dave	Fold					-$0
Poki	Call $10	Call $10	Call $10	Call $20	Call $20	-$70

Table 1.3: Betting history of example hand (actions proceed one column at a time).

of the same suit and neighbors in rank), we feel we have a positive expectation and call. Alice raises, putting another $15 into the pot (she has now put a total of $20 into the pot), and Bill, the big-blind folds (forfeiting his $10 big-blind). Chris calls Alice's raise, and then it is back to us. We have already committed $10 to the pot, so we call Alice's raise as well. This ends the first betting round. There is $70 in the pot and there are three players left.

The flop comes down J♠-8♣-4♠. Although this does not pair either of our cards, we have a very high potential to make a straight or a flush on the turn or river. Alice is first to act and bets $10. Chris is next and calls the bet. We figure we have more than a 30% chance that the turn card will give us a flush or straight (a very strong hand) and the pot is $90, so we are getting 9-to-1 odds for a call.[2] We decide that with our huge potential, it is well worth the $10 and we call. With such a high potential it would even be prudent to raise. The pot is now $100.

The turn card is a J♢ which still does not help us. Alice bets $20, and Chris folds. It is known from past experience that Chris is a fairly *tight* player who doesn't like to take many risks. He might have had a hand like a pair of nines, and he now feels that he is beat because of the presence of the two Jacks on the board. Since Alice has bet, she may have a Jack, giving her three-of-a-kind, or perhaps a big *pocket-pair* such as two Queens. We suspect this might also be the case, but we have observed Alice to be an aggressive player and she frequently bluffs. The pot is now $120, so our odds are now 6-to-1, which are more than ample to justify calling the $20. Figure 1.1 shows a snapshot of the game at this point.

The river card is the T♡ which gives us two pair. Again, Alice bets $20. We don't think we necessarily have the best hand, but the pot is large enough that we should call in case Alice is bluffing (to consider folding with a pot this large, we must be more than 80% certain of having the worst hand). We call and Alice shows A♡-J♡, giving her three-of-a-kind. We toss our hand in the *muck* and let Alice take the pot of $180. Table 1.3 summarizes the history of this game.

[2]Suppose we played this situation the same way 100 times. In 30 of these instances, we would make a strong hand and win $90, for a total of $\$90 \times 30 = \$2700$. The 70 times we called $10 and did not make our *draw*, we lose $10 each time, for a total loss of $700. By calling, we would net a total of $2000 in 100 hands, or $20 per hand.

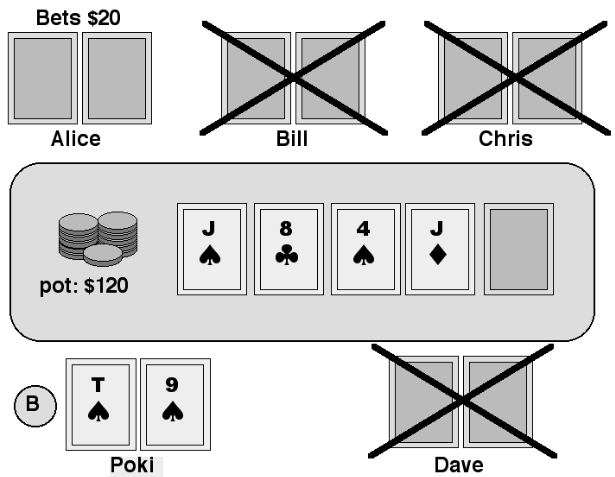

Figure 1.1: A snapshot of the example hand on the turn. It is *Poki*'s turn to act.

1.3 Opponents

Players come in all types imaginable. Part of modeling an opponent is classifying where they lie on the various spectrums of playing styles. One must try to discover from observation the general strategy an opponent employs in their play. For example this might involve determining that a player is *tight* (playing only strong hands), or *loose* (playing many marginal hands). We can measure how *passive* or *aggressive* the player is, by observing how often they bet and raise hands they consider to be strong. We can also measure how observant a player is, and how quickly they adapt to other player's strategies.

For the most part, many of these expert-defined values are easy to calculate. From a scientific standpoint, it would be far more interesting to have a system learn these relevant features of opponents from scratch, rather than having expert knowledge of these features directly encoded. Computing a measure of tightness or aggressiveness is easy to do for an overall measure of a player's style, but a good player mixes or frequently switches styles depending on the particular context of a hand. In this case, overall measures do not provide enough useful information.

Many tactics are deployed by opponents in order to make themselves difficult to model. For instance, if a player has only a mediocre hand, they may bluff, representing a strong hand, hoping their opponent will fold. Or a player may under-represent a very strong hand early in the game to sucker more players into staying in the hand, which later leads to a higher pay-off from the pot.

It is not how many games are won that is important, but how much money is won over all hands played. In order to cause opponents to build flawed models, a player may purposefully play a few relatively unimportant hands terribly. Later, this *table image* can be exploited.

Opponent modeling is the task of learning how an opponent plays poker. A complete opponent model can tell us several things. It can tell us the general styles that the opponent plays. It can infer from an opponent's actions a probability distribution over the cards that they may hold. Given a hand and a game situation, the model should also be able to mimic playing poker the same way the opponent would. This, in essence, is the goal of this thesis research.

1.4 Research Goals / Thesis Contributions

The generalized aim of this thesis is to develop ways of modeling uncertain environments and how to make intelligent decisions within them.

This thesis builds heavily upon the past thesis research by Darse Billings, Denis Papp, and Lourdes Peña. They have all participated in building the foundation for this computer poker research – the poker-playing agent *Loki*. Darse Billings, who is both a researcher and a poker expert, laid the foundations of our computer poker AI [3]. Denis Papp brought many of these ideas to life in the first incarnation of *Loki* [34]. Lourdes Peña extended *Loki* with a new adaptive, simulation-based betting strategy [35].

The organization of this thesis is as follows:

- Chapter 2 gives an overview of the game theoretic properties of poker, and demonstrates some of these elements using simpler games. Other poker research is also discussed.

- Chapter 3 explains basic algorithms used for computer poker. This includes hand evaluation, betting strategies, and the overall architecture of our poker playing program. The ideas presented have been embodied in a poker playing agent called *Poki*, a Java-based descendant of the C++ poker program *Loki*. *Poki* was a complete reimplementation of *Loki*, in a much cleaner object-oriented architecture. Numerous improvements were made, most notably in the opponent modeling systems.

- Chapter 4 explores the opponent modeling problem in poker. The opponent modeling in *Loki* was fairly crude. Many machine learning techniques are discussed – leading to major improvements in *Poki's* opponent modeling abilities. This chapter represents a large portion of this thesis' contributions

- Chapter 5 discusses the experiments and results obtained from self-play experiments, offline opponent modeling analysis, and online games against human players.

- In Chapter 6 the results of the simulation based betting strategy are discussed. A new search method for imperfect-information game trees is introduced and evaluated.

- Chapter 7 presents conclusions, provides a summary of the thesis contributions, and presents suggested avenues for future research.

Chapter 2

Games in Theory and in Practice

The laws of probability, so true in general, so fallacious in particular.

– Edward Gibbon (1737-1794)

Research on imperfect information games can be divided into two camps. The older of the two is *game theory*, pioneered by theoreticians such as John Von Neumann and John Nash [52]. Game theory is concerned with the mathematical and logical structure of non-cooperative games, and how to make rational choices within them. Game theory is often applied in economics, politics, and classic games.

The other camp uses a more empirical approach with the goal of studying and creating high-performance systems for playing games. This thesis research belongs in this latter category. However, in building high-performance game playing systems, one must be aware of the theory behind the game. Game theory can tell us what to expect from various situations in the game, as well as which strategies may be strong or weak. This section will deal with the theoretical issues surrounding poker. To begin, we will look at some simpler games that share some of the game-theoretic properties of poker.

A *non-cooperative game* is one where the interests of each player are strictly opposed, and a *zero sum game* is scored such that all player's scores sum to zero. If one player gains, it is by the equivalent loss of the other players. Games of *imperfect information* require a player to make decisions without knowing the complete state of the game.

The following discussion will present two simple games. As the main topic of this thesis is opponent modeling, there will be particular focus on the importance of identifying your opponent's strategy.

2.1 RoShamBo

RoShamBo, also known as *Rock, Paper, Scissors*, is a game with very simple rules (it is often played by children). Two opponents choose one of three actions (paper, rock, or scissors) simultaneously. Paper beats rock, rock beats scissors, and scissors beats paper. For instance, if player A chooses rock, and player B chooses scissors, then A wins one point, while B loses one point. If both players choose the same thing, they draw and score no points. RoShamBo is zero-sum, and contains hidden information (the simultaneous choice of actions can be viewed as each player choosing their action secretly, before the both players choices are revealed). Its simplicity, however, is deceptive when it comes to playing well against an arbitrary opponent.

Game theory tells us that the optimal solution is trivial. A player who chooses an action uniformly at random (an equal probability of choosing each of the three actions) cannot be exploited in any way. No matter what strategy an opponent employs against this optimal strategy, the expected value of a match is a draw. Of course, while this strategy cannot lose in the long run against the opponent, neither can it win.

In 1999 and 2000, Darse Billings hosted an international RoShamBo programming competition that was modeled after the iterated prisoner's dilemma tournaments (discussed in the following section) [5, 4]. In a tournament setting, each pair of programs played a match of 1000 games against one another. The cumulative score of each opponent dictated its final place in the rankings. Hence, if the field contained at least two non-optimal players that did not draw against one another, an optimal player could not place first.

The tournaments drew many high quality submissions. It was observed that the success of many strategies was significantly dependent on the composition of the other strategies. The authors of the top entries, including some well-known AI researchers, have commented that writing a strong RoShamBo program was much more challenging than they initially expected. The best programs do a sophisticated analysis of the full history of the current match in order to predict the opponent's next action, while avoiding being predictable itself. Programs that used a simple rule-based model for making their decisions consistently finished near the bottom of the standings.

A program named *Iocaine Powder* stood out from the crowd [15]. Although deterministic, it outplayed the other exploitable strategies by out-guessing them. Iocaine Powder does not play an *optimal* strategy. Since it is deterministic, a program could be written to beat it.[1] With the knowledge that the other players are

[1]For instance, one could copy the source code to Iocaine Powder, and play the winning move against what Iocaine Powder would have done. This would play perfectly against Iocaine Powder, but would lose to other strategies.

	Prisoner A	
Prisoner B	Cooperate	Defect
Cooperate	2,2	0,10
Defect	10,0	5,5

Table 2.1: The payoff matrix for a Prisoner's Dilemma game. Each cell contains the penalty for Prisoner A, then Prisoner B.

playing non-optimally, Iocaine Powder attempted to play a strategy that maximized winnings by exploiting these program's predictability.

The example of RoShamBo demonstrates the importance of opponent modeling in imperfect information games. The variety of strategies and the composition of the field determine the success of each non-optimal strategy. The contrast is clear between the optimal strategy and a context-dependent *maximizing* one. The optimal strategy chooses randomly, with no regard for the opponent's strategy, while maximal strategies explicitly observe and outguess their opponents.

2.2 Prisoner's Dilemma

The classic example of an imperfect information game studied by theorists is *Prisoner's Dilemma*.[2] In Prisoner's Dilemma, the two players have just been arrested for a crime and imprisoned in separate cells. The players may not communicate with one another (this gives us the element of missing information). If both players deny their guilt, there will not be sufficient evidence to keep them imprisoned for more than two years. If one player squeals on the other, while the other denies his guilt, the squealer will go free, and the other will be imprisoned for ten years. However, if both squeal on each other, they will both be imprisoned for five years.[3]

The dilemma is obvious. If both prisoners cooperate and stay quiet, they will only serve two years. However, it is tempting to defect and squeal on the other prisoner in an effort to walk free. But if both prisoners decide to defect, the penalty is severe. A typical representation of this kind of game is a *payoff matrix*, shown in Table 2.1.

If the game is played just once, the best strategy is to defect, since it dominates the other options (defecting gets a worst case of five years in prison, while cooperating can lead to ten years in prison). The game can be made even more interesting by playing a series of games, and keeping score. This is called *Iterated Prisoner's Dilemma*. In the iterated case, cooperation becomes a viable option. After a few trials, one can begin to analyze the choices made by the opponent,

[2] It is important to note that this game is not zero-sum, nor is it non-cooperative. Hence, it does not fall into the category of proper game theory. However, it is a widely researched and discussed game of imperfect information, where modeling the opponent is a critical aspect.

[3] These numbers can be arbitrary, provided the relative orderings are preserved.

and develop a specific strategy to thwart it. In 1979, Axelrod conducted an *iterated* Prisoner's Dilemma programming contest. Participants were invited to submit programs, which played one another in a tournament [1].

A wide variety of different strategies competed in the tournament. One of the fundamental lessons learned from these competitions (which is why this is being discussed here) was that the best strategy is dependent on the composition of the field (the other competitors).

For instance, one of the most successful strategies was called *tit-for-tat*. The strategy was extraordinarily simple. It starts by cooperating on the first trial. Afterwards, it does whatever the opponent did in the previous trial. The reason this works so well is not so easy to see. In short, it rewards cooperation with cooperation, and punishes defection with defection, but is quick to forgive. If the opponent cooperates, so will *tit-for-tat*. If the opponent ever defects, *tit-for-tat* will be burned once, but then will defect in the next round. If the opponent defects again, both will take a penalty, and as long as the opponent continues defecting, so will *tit-for-tat*. Once the opponent realizes defecting is not working (it is taking a penalty each time), it may decide to cooperate again. But the instant it switches from defecting to cooperating, *tit-for-tat* will defect since that was the opponent's last choice. In the next trial, *tit-for-tat* will go back to cooperating, and the entire pattern may play out again. In the long run, the entire sequence ends up as (or close to) a draw, because *tit-for-tat* plays so defensively. While *Tit-for-tat* can draw, it also can't win an individual match – a draw is the best it can do. However, it can lose every single match and still win a tournament with a high overall score.

2.3 Nash Equilibria

Nash showed that all zero-sum imperfect information games have an equilibrium (or saddle point) in which every player can ensure the optimal outcome with an appropriate randomized mixed strategy [31]. A game-theoretic *optimal strategy* is a strategy that maximizes gain and minimizes loss against *any* possible opponent strategy. An optimal strategy is the safest strategy one can use when the opponent's strategy is unknown. In other words, if your opponent is using an optimal strategy, the best you can do against them is to play optimally as well.

However, an *optimal* strategy is not necessarily a *maximal* strategy. If your opponent is not playing optimally, there may exist a non-optimal strategy which will maximize winnings against their non-optimal strategy. In order to exploit a non-optimal opponent, one must know what kinds of mistakes they are making. The goal of opponent modeling is to learn how an opponent plays so that weaknesses may be exploited, and strengths may be avoided.

Iterated Prisoner's Dilemma is not a zero-sum game, so Nash Equilibria do not necessarily exist. The *tit-for-tat* strategy is not optimal, but it is robust (it will never lose by more than a few points, no matter what the opponent's strategy). While it is robust, it never initiates defection, which means it cannot maximize winnings against weak opponents. For instance, if it were playing an opponent that always cooperated, *tit-for-tat* would never attempt to exploit it by defecting.

2.4 Optimal Poker Strategies

Some of the previous academic research involving poker has been focused on game theory and optimal solutions. In fact, poker was used for many examples in early game theory explorations by John von Neumann and John Nash [52, 32, 31].

While full-scale poker is still much too large to solve, smaller variants of poker have been analyzed and solved. For example, a smaller deck of cards may be used, along with fewer, less complicated betting rounds. Typically, only two-player variants are examined, and not full ring games (usually about ten players). One of the smallest variants is Kuhn's two-player, three-card, poker game [27]. Each player is dealt one card, and there is a single round of betting. This game is simple enough that it can be dissected and solved by hand.

More recently, simplified poker games have been solved computationally. Koller and Pfeffer have worked on *Gala*, a generalized system for finding optimal randomized strategies for two-player imperfect information games [25]. The standard method of computing optimal strategies for small games is to convert the game tree into *normal form*. However, this representation is exponential in the size of the game tree, rendering the technique useless for all but the simplest of games.

Instead, Koller and Pfeffer's approach uses a new method that is linear in the size of the game tree [24]. Using this technique, they have found optimal strategies for several scaled-down poker-like games. However, the size of the game tree for full-scale two-player Texas Hold'em is still many orders of magnitude higher than the largest problem the Gala system can solve.

Ken Takusagawa, a former student of Daphne Koller, has extended this work by combining the method with abstraction, to produce some near-optimal solutions for particular scenarios of Texas Hold'em [47].

A subset of Texas Hold'em, two-player *Pre-flop Hold'em* has been solved optimally by Alex Selby [41]. In Pre-flop Hold'em, once the pre-flop round is finished, all five board cards are dealt and the players proceed directly to the showdown.

Jiefu Shi and Michael Littman have worked on a smaller version of Texas Hold'em, humorously dubbed *Rhode Island Hold'em* [44]. While much smaller than Texas Hold'em, the game is still too large to solve optimally. Littman collects similar groups of cards into equivalence classes in order to reduce the size of the problem to a manageable size. The resulting solution, while not perfect, is *near-optimal*. Since poker has such high variance, a near-optimal strategy is likely to perform extremely close to an optimal strategy against other non-optimal opponents.

2.5 Other Poker Research

Besides Texas Hold'em, there has been work on other Poker variants such as Five-Card Draw. Five-Card Draw poker is not as strategically complex as Texas Hold'em or 7-card Stud.

Nicolas Findler did early work on this game, intermittently over a twenty year period [18]. His main objective was not to achieve a high level of play, but to model human cognitive processes. His program was able to learn, and was moderately successful.

In the 1970's and 1980's some research was done in exploring production systems for Five-Card Draw strategies. Waterman developed a program which was able to learn the production rules through three different methods – advice (expert rules), analysis (post-game), and adaptation (opponent modeling) [53]. Smith later developed a different production system that learned rule sets by using a genetic algorithm [46]. Smith's program was able to win against Waterman's program, which was considered to play at the level of an experienced human player.

Five-Card Draw poker has been recently studied using Bayesian models with mixed results [26, 12].

2.5.1 Evolutionary Methods

Recently, evolutionary algorithms have been applied to game theory and to poker. Genetic algorithms have been suggested as a way to discover local, and possibly global, Nash-equilibria in game-theoretic domains [16]. A population of co-evolving strategies will eventually converge to a local optimum with respect to the population of competing strategies.

Because of poker's game-theoretic underpinnings, a similar approach has been applied to evolve strategies for ten-player Texas Hold'em [33]. Simplified strategies are described by a small array of values. A regular co-evolutionary genetic algorithm (GA) starts with a population of randomly generated strategies. The strategies are then evaluated by having them compete against one another. The highest scoring strategies are reproduced with some random variation (usually by small perturbations of the values in the strategy). Once a new population has been constructed, the process repeats. The selection pressure steers the random search towards higher scoring strategies, until the population converges on a local optimum.

The method of selecting the surviving strategies from a population determines how the population will evolve, and towards which local equilibrium it will converge. Using ideas from game theory, a more robust method of selection in co-evolving populations can be used. Pareto co-evolution [17] is a new method of selection based on the concept of Pareto optimality. A strategy Pareto-dominates another if it is at least as good as the other strategy over all dimensions, provided it is strictly better in at least one dimension. A Pareto-optimal strategy would be a strategy that Pareto-dominates all other existing strategies. This idea is applied to co-evolution as a way to select for Pareto-dominating strategies. The dimensions of evaluation become the strategy's performance against the other strategies in the population itself. The other strategies are, in a sense, the dimensions for evaluation.

This changes the selection pressure substantially. It selects the more robust strategies, that do well against a larger set of the opponents. The early research indicates that the Pareto co-evolution outperforms the standard selection method of choosing the n strategies with the highest gross scores.

Currently, however, no evolutionary computation methods have been applied to generate high-performance betting strategies for poker. It is likely that the complexity of Texas Hold'em is too great for a naive evolutionary algorithm to tackle effectively.

Chapter 3

Computer Poker Basics

Do not expect to arrive at certainty in every subject which you pursue. There are a hundred things wherein we mortals. . . must be content with probability, where our best light and reasoning will reach no farther.

– Isaac Watts (1674-1748)

The fundamentals of poker strategy are determined by the probabilistic nature of the game. The way a hand is played depends on the probability of both playing the hand profitably, and how it affects the profitability of future hands (by changing your table image). Winning an individual hand is not as important as winning in the long run. As mentioned earlier, poker strategy requires both mathematical understanding of the game and psychological understanding of the opponent. To play well, the concrete mathematical foundations must be in place. The psychological or intuitive aspect of poker can be used to enhance the accuracy of this foundation.

There are several skills required to play strong poker. These will all be required for a world class poker agent:

Hand Evaluation:

Accurate assessment of the current strength of our hand, and the potential strength of our hand when the future board cards are dealt, is crucial. We need the probability that our hand is the best, given the context of the game and the opponents we are up against.

Unpredictability:

Our actions must not give away the strength of our hand. We must play deceptively by *bluffing*, *slow-playing*, and *check-raising*. It is important that our betting strategy be non-deterministic, and mixes strategies in order to conceal information about our hand.

Opponent Modeling:

> We must understand how our opponent plays in order to both exploit their weaknesses and defend against their strengths. We must be able to deduce the strength of hands our opponents hold by interpreting their actions, and we must be able to anticipate how they will play.

This chapter discusses the pre-flop play, post-flop hand evaluation, and betting strategies used by *Poki*. Many of the methods in this chapter are described more thoroughly in [34].

3.1 Poki's High-Level Architecture

A poker game consists of a *dealer* together with multiple *players* that represent either human players or computer players. In our Java implementation, these players are defined as objects. The dealer handles the addition and removal of players from the game, deals the cards to each player at the start of a new hand, prompts each player for an appropriate action when it is their turn, broadcasts player actions to other players, and updates a public game context as the game progresses. The game context contains all of the public information about the game, including the names and relative locations of the players, and the board cards.

Each player extends an abstract *Player* class, that has access to all the public game information, as well as the player's private information (their cards). This class is extended to add the player's personal strategy. A player can be a *bot* (software program), or an interface for a human to play in the game, just by extending this class appropriately.

Likewise, the *Dealer* is a generalization for game controllers. We have implemented several different dealer interfaces: an IRC-Dealer for playing against other players on the IRC poker server, a Tournament-Dealer for self-play experiments, and a TCP/IP-Dealer that allows *Poki* to play against humans using a web browser, and against other programs using a published protocol (see Appendix B).

Each player is responsible for responding to a prompt by choosing an action: fold, call, or raise. In addition to the public game context, each player knows its private hand and the previous actions of all of the players. Each player is defined by an interface, and each version of *Poki* is a different implementation of that player interface.

Figure 3.1 shows the high level architecture of our Java-based poker system. The arrows depict the flow of information. The public game state and history is comprised of a *GameInfo* object, also containing several *PlayerInfo* objects which track public information for each player. The *Player* objects are each given cards by the dealer, and are then prompted for each action in the game. The players may

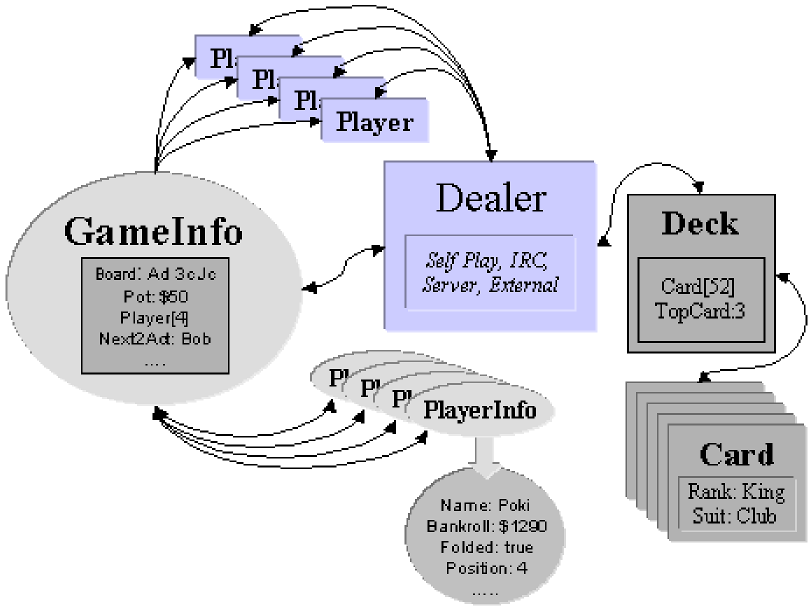

Figure 3.1: High level architecture.

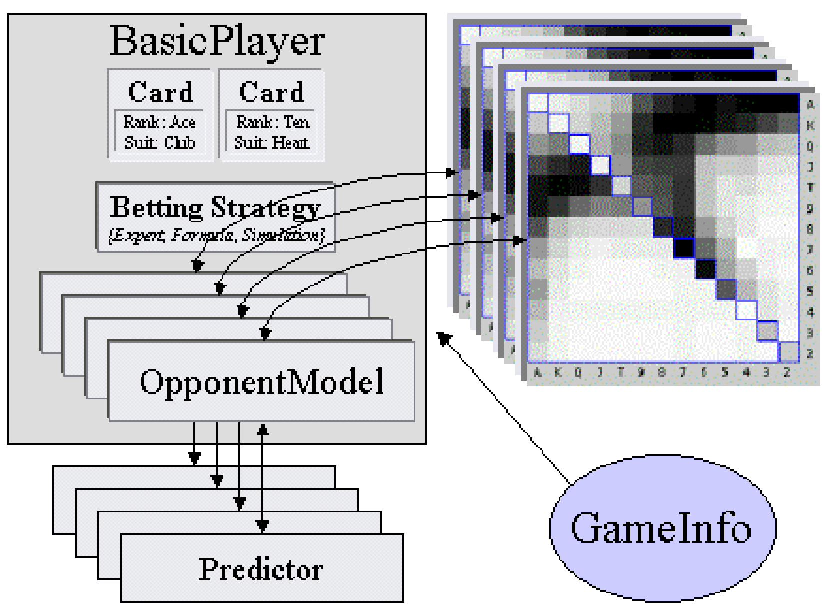

Figure 3.2: High level player architecture.

access the GameInfo and PlayerInfo objects to make their decisions. The dealer then modifes the GameInfo and PlayerInfo objects according to the rules of the game.

Although each version of *Poki* represents and uses the available information in a different way, all versions share a common high-level architecture. In addition to the public game context, *Poki* has two private components: its hand and a collection of persistent opponent models. *Poki* maintains statistical models describing each player participating in the game, including itself. When the dealer broadcasts a player action, the corresponding opponent model is responsible for processing that information.

3.2 Pre-flop Betting Strategy

The strategy of Texas Hold'em in the pre-flop stage (before any board cards are dealt) is different than in the post-flop stages. Because there are no public cards available, the state in the pre-flop is far simpler than later in the game. In total, there are $\binom{52}{2} = 1326$ different hands, but many of these hands are equivalent before the flop. Without board cards, the particular suit of the cards is irrelevant. Both Q$\diamondsuit$-3$\diamondsuit$ and Q$\clubsuit$-3$\clubsuit$ have the same probability of making a flush, and in all other respects they are equal. If the flop contains any diamonds or clubs, then the two hands can no longer be treated identically. Using this knowledge, there are only 169 distinct hand types in pre-flop Hold'em.

Poki's pre-flop strategy is a simple expert system built on top of a set of machine-learned tables containing an expected income rate for each hand. These tables were calculated off-line using a simple technique called *roll-out simulations.* Several million hands were played where each player calls the blind and then checks to the showdown. The cumulative wins and losses for each hand type are kept in the table. Although based on unrealistic play, these values give a good baseline for the relative profitability of each of the 169 hand types.

A refinement to roll-out simulation is to use repeated iterations of the technique, where the previous results govern the betting decision for each player. For instance, in a ten player simulation, a net negative value in the previous iteration would dictate that the hand be folded, rather than calling the big blind. This introduces some hand selection into the roll-outs, which reduces the number of active players in each hand. This generates a more realistic distribution of opponents and probable hands. Many mediocre hands that were profitable to play against players with no hand selection become negative in this more realistic simulation.

After each round of simulations has reached a reasonable degree of stability, another iteration is performed. This process eventually reaches an equilibrium, defining a set of hands that can be played profitably against the blinds and the other unknown hands. The results are most applicable to the "play or don't play"

Hand	IR-10	Iterated	Hand	IR-10	Iterated	Hand	IR-10	Iterated
AA*	+2112	+2920	ATs	+736	+640	KQo	+515	+310
KK*	+1615	+2180	99*	+553	+630	QTs	+655	+280
QQ*	+1224	+1700	KQs	+858	+620	QJs	+720	+270
JJ*	+935	+1270	AQo	+555	+560	A9s	+538	+220
TT*	+714	+920	KJs	+767	+480	ATo	+335	+200
AKs	+1071	+860	88*	+450	+450	KTs	+694	+190
AKo	+718	+850	77*	+364	+390	KJo	+391	+160
AQs	+915	+780	AJo	+430	+380	A8s	+469	+110
AJs	+813	+680	JTs	+657	+360	66*	+304	+40

Table 3.1: Iterated income rate (profitable hands).

decision for each player. Although much better than a simple roll-out simulation, this technique is still far from perfect, because other important considerations such as betting position and known opponent actions have not been accounted for.

In our experiments, each iteration lasted for 50,000 trials. A diminishing noise factor was added to each income rate, analogous to the cooling factor used in simulated annealing. This gives negative expectation hands a chance to recover as the prevailing context changes. After ten generations, the remaining positive expectation hands were played for another 500,000 trials, to ensure stability. The resulting set of profitable hands, shown in Table 3.1, is in strong agreement with expert opinion on this matter [45]. The table shows a comparison of the income rates for 10-player roll-out simulations (IR-10) and the results refined by iterating (Iterated). The values shown are in *milli-bets* (*eg.* a hand with an income rate of +1000 should win an average of one small bet each time it is played). The iterated values are reasonable estimates of actual income rates, unlike the simple roll-out values, which are only used as relative measures.

We can generate *near-optimal* solutions for the two-player game of pre-flop Hold'em with iterated roll-out simulations. The results closely match Selby's results [41] for the optimal two-player pre-flop Hold'em game.

3.3 Hand Evaluation

The post-flop betting strategy is far more difficult than the pre-flop strategy. There are several different strategies for the post-flop that will be described later. Before we do that, we must first describe how hands are evaluated. Later, the betting strategies will show how all of the evaluation information can be tied together into a rational betting decision.

```
HandStrength(ourcards,boardcards) {
   ahead = tied = behind = 0
   ourrank = Rank(ourcards,boardcards)
   /* Consider all two card combinations of the remaining cards.*/
   for each case(oppcards) {
      opprank = Rank(oppcards,boardcards)
      if(ourrank>opprank)        ahead += 1
      else if(ourrank==opprank)   tied += 1
      else /* < */              behind += 1
   }
   handstrength = (ahead+tied/2) / (ahead+tied+behind)
   return(handstrength)
}
```

Figure 3.3: Hand strength calculation.

3.3.1 Hand Strength

A crucial piece of information needed to make good decisions is the strength of our hand. A player must have a good estimate of the probability that they currently have the winning hand. A simple calculation of hand strength (HS) is to enumerate all possible other hands and compare them to ours. For instance, after the flop there are only 47 unknown cards remaining (52 cards minus our hole cards and the three board cards). Thus there are $\binom{47}{2} = 1081$ possible two card combinations we could be up against.

By counting the number of these other hands that win, tie, and lose to our hand, we can compute the probability of having the best hand, against one random hand. Figure 3.3 gives the algorithm for a simple hand strength calculation [34]. Essentially, this is just a calculation of the proportion of possible scenarios in which we have the best hand.

For example, suppose our hand is A♢-Q♣ and the flop is J♡-4♣-3♡. There are 1081 possible hands an opponent could be holding. Of these possible hands there are 444 that are currently better than our hand, 9 that would tie, and 628 that are worse. Counting ties as one half, this corresponds to a percentile ranking, or hand strength, of 0.585. In other words, there is a 58.5% chance that A♢-Q♣ is better than a random hand.[1]

This value, called the *hand rank* (HR) or *raw hand strength* (RHS), is a good first-order approximation of the strength of our hand. However, for a real game of poker, this value is too crude. If our opponent has called one or more bets, then we are most certainly not up against a random hand anymore. Rational opponents will not play most weak hands they are dealt. It is typical for Hold'em players to fold over 50% of their hands before the flop. In most situations, if an opponent

[1]Hand Strength calculations for more than one opponent will be discussed below in Section 3.3.5.

has not folded and has voluntarily put money into the pot, there is a much higher probability they are holding a hand of higher quality than average. Hands that are clear losers would normally be folded. It is too simplistic to count unlikely hands as having the same probability as likely hands.

To make the measurement more realistic, the enumeration can be weighted by the probability of each hand being held. Instead of adding one to each counter for *ahead*, *tied*, and *behind*, a weight representing the relative probability that an opponent has a specific hand can be used instead. Weighting the enumeration will be described in more detail in Section 3.3.3.

3.3.2 Hand Potential

Hand strength is a measure of the current strength of our hand. However, after the flop, the turn and river cards are yet to be dealt. These future community cards can significantly affect our hand strength. For instance, suppose Alice has K♣-K♠, and Bob has 7♡-9♡, with a flop of 8♡-6♣-4♡. At this point, Alice has a high hand strength (RHS = 0.945), while Bob has a low hand strength (RHS = 0.104). However, Bob's hand has several *outs* (cards that improve the hand) that would give him a winning hand (any of 9 remaining hearts to make a flush, plus the T♠, T♢, T♣, 5♠, 5♢, and 5♣ for 6 ways to make a straight). The probability that any one of these cards will come down on the turn or river is high (15/47 unknown cards gives a 32% chance the next card will improve Bob's situation). Despite the weak hand strength, Bob should stay in the hand because of the high potential.

The *Positive Potential*, or *PPot*, is the probability that our hand will *improve* after a new card is dealt. In general, improvement is measured by counting each situation where we are currently *behind* and will end up *ahead*, or *tied*. Likewise, *Negative Potential*, or *NPot*, is the count of hands where we are currently ahead, but will end up behind.

The actual potentials of a hand can be computed by enumeration of all the possibilities. On the flop, there are still two cards to be dealt, so a full calculation requires a two-card look ahead. There are 1,070,190 combinations in a two card look ahead (1081 possible opponent hands, multiplied by the 990 possible turn and river cards). On the turn, there is only one card to come, making for only 43,560 combinations. The algorithm for hand potential is given in Figure 3.4 [34]. A breakdown of cases from the example hand above is shown in Table 3.2.

$$PPot = \mathcal{P}(ahead \mid behind) + \frac{\mathcal{P}(tied \mid behind)}{2} + \frac{\mathcal{P}(ahead \mid tied)}{2} \tag{3.1}$$

Using this equation we can compute the one and two card look ahead potentials for this example from Table 3.2. PPot1, the one-card potential is 0.406, while the two card potential, PPot2, is 0.620.

```
HandPotential(ourcards,boardcards) {
  /* Hand potential array, each index represents ahead, tied, and behind.  */
  integer array HP[3][3]  /* initialize to 0 */
  integer array HPTotal[3] /* initialize to 0 */

  ourrank = Rank(ourcards,boardcards)
  /* Consider all two card combinations of the remaining cards for the opponent.*/
  for each case(oppcards) {
    opprank = Rank(oppcards,boardcards)
    if(ourrank>opprank)      index = ahead
    else if(ourrank=opprank) index = tied
    else /* < */             index = behind
    HPTotal[index] += 1

    /* All possible board cards to come.  */
    for each case(turn) {
      for each case(river) {
        /* Final 5-card board */
        board = [boardcards,turn,river]
        ourbest = Rank(ourcards,board)
        oppbest = Rank(oppcards,board)
        if(ourbest>oppbest)      HP[index][ahead ]+=1
        else if(ourbest==oppbest)HP[index][tied  ]+=1
        else /* < */             HP[index][behind]+=1
      }
    }
  }

  /* PPot:  were behind but moved ahead.  */
  PPot = (HP[behind][ahead] + HP[behind][tied]/2 + HP[tied][ahead]/2)
       / (HPTotal[behind]+HPTotal[tied]/2)
  /* NPot:  were ahead but fell behind.  */
  NPot = (HP[ahead][behind] + HP[tied][behind]/2 + HP[ahead][tied]/2)
       / (HPTotal[ahead]+HPTotal[tied]/2)
  return(PPot,NPot)
}
```

Figure 3.4: Hand potential calculation.

Before	After	Turn Cases	%	River Cases
Ahead	Ahead	4380	9%	89836
Ahead	Tied	0	0%	4449
Ahead	Behind	480	0.9%	12635
Tied	Ahead	81	0.17%	3240
Tied	Tied	324	0.67%	5625
Tied	Behind	0	0%	45
Behind	Ahead	17663	36%	590955
Behind	Tied	0	0%	3010
Behind	Behind	25717	53%	360395

Table 3.2: Breakdown of cases in a hand potential for the hand 7♡-9♡ with a flop of 8♡-6♣-4♡.

Just as hand strength can be computed by enumeration, so can hand potential. Figure 3.4 shows the enumeration algorithm for computing PPot and NPot. One card look ahead is computable on today's hardware (see Appendix A) in roughly 280 milliseconds. Two card look ahead can be done in 2000 milliseconds, which may limit its usage in practical algorithms for real-time play.

3.3.3 Weighting the Enumerations

The calculations of hand strength and hand potential in Figures 3.3 and 3.4 assume that all two card combinations are equally likely. However, the probability of each hand being played to a particular point in the game will vary. For example, the probability that the opponent holds Ace-King is much higher than 7-2 after the flop, because most players will fold 7-2 before the flop.

During the course of each hand, a *weight table* is maintained for each opponent. This is simply an array of values that represent the probability of the opponent holding each possible combination of hole cards. For each of the 1326 entries (one for each two-card combination)[2], we store a weight in the range 0.0 to 1.0. This weighted distribution is updated after each opponent action to reflect the hands that are consistent with the betting decisions observed throughout the current hand. The precise details of this *re-weighting* process depends on our method of modeling each opponent (described in Chapter 4).

The strength of each possible hand is assessed, and a probable distribution of actions is determined by a formula-based betting strategy. These values are then used to update the weight table after each opponent action. The algorithm is shown in Figure 3.5. For each possible opponent hand, the opponent model returns a probability distribution over the likely actions with that hand, given the current

[2]We use a 52 by 52 table for indexing convenience. Weights for all card combinations containing known cards (in our hand and on the board) are set to zero.

```
UpdateWeightTable(Action A, WeightTable WT, GameContext GC, OpponentModel OM) {
  foreach (entry E in WT){
    ProbabilityDistribution PT[FOLD, CALL, RAISE]
    PT = PredictOpponentAction(OM, E, GC)
    WT[E] = WT[E] * PT[A]
  }
}
```

Figure 3.5: The algorithm for updating a Weight Table. The probability of the observed action (A) is taken from an opponent model's (OM) prediction of the opponent (PT) based on the hand (E) and the current game context (GC). This probability (PT[A]) is used to weight the table entry (WT[E]).

game context. From this distribution, the probability of the actual observed action is used to multiply the weight table entry.

For example, suppose an opponent calls before the flop. The updated weight for the hand 7-2 might be 0.01, since it would normally be folded. The probability of Ace-King might be 0.40, since it would seldom be folded before the flop, but is often raised. The relative value for each hand is increased or decreased to be consistent with every opponent action.

Figure 3.6 shows the reweightings of an opponent's weight table. The images are constructed to show the 1326 different hands in a 13 by 13 table. The axes represent the rank of each of the cards. Suited hands are in the upper-right half of the table, unsuited hands are in the lower-left half, and pairs are along the diagonal. The shading in the cell represents the maximum value for the hands grouped in that cell. The darker the cell, the higher its weight. For instance, for the *Ace-King Suited* cell (in the upper-right half of the grid), its value is determined by the maximum weight of the four different Ace-King suited hands (Ace-King of Spades, Ace-King of Hearts, etc...).

From Figure 3.6, we can see the difference in the weights for an opponent that folds, calls or raises in the pre-flop betting round. In the case of folding, the strong and mediocre hands are all shaded lightly or not at all. The weak hands (low cards, unsuited, unconnected) are more likely to be folded and thus have high (dark) weights. For the calling scenario the high weights are in the middle – hands that are consistent with being called. Weak hands would likely have been folded and strong hands would likely have been raised, so their weights are low. Likewise, the raising example shows strong weights only for the very good hands, and low weights for everything else.

To better understand how the weights are updated, consider the weight tables in Figure 3.7, generated from the sample hand in Chapter 1.2.1. The weight tables for Alice on the flop (J♠-8♣-4♠), after she bets $10, might look something like in

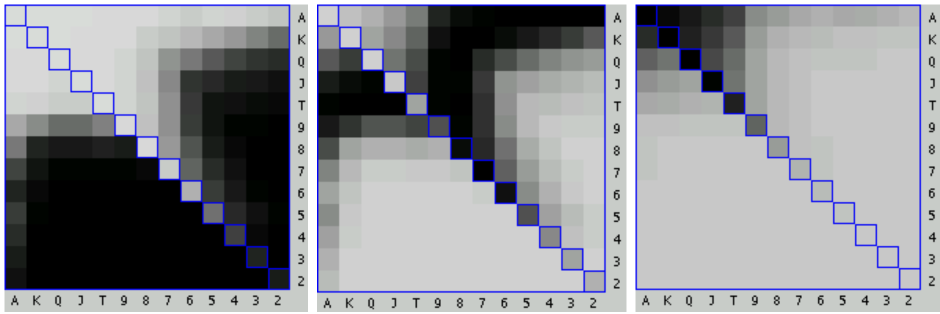

Figure 3.6: An example weight table after a pre-flop fold (left), call (middle), or raise (right).

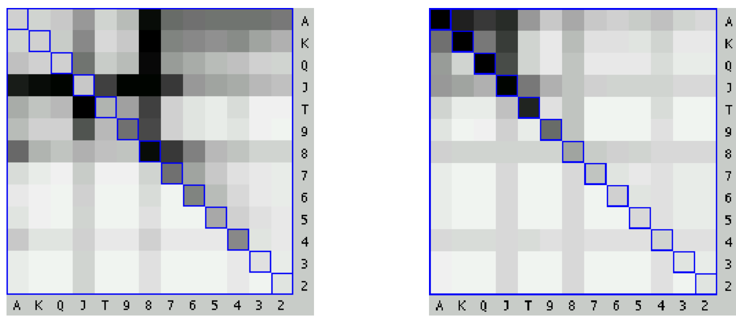

Figure 3.7: Two scenarios for calling pre-flop (left) and raising pre-flop (right), followed by betting after a flop of J♠-8♣-4♠ (See Chapter 1.2.1).

Figure 3.7. On the left is a table for the scenario where she calls pre-flop, and on the right is how the table would appear if she had raised pre-flop (as she did in the example).

The table for the pre-flop call scenario (left) represents hands such as any hand with a Jack (except for a pocket pair of Jacks, because that hand would have been raised pre-flop). Hands that make a pair of Eights are likely, if they have a large *kicker* (for example, King-Eight has a high weight, but Eight-Three does not). Hands that make two-pair (like Jack-Eight) also have relatively high weights. The upper-right triangle of the table represents suited hands, and because there are two spades on the board, the suited half of table is darker than the unsuited half – Alice could be betting a strong flush draw.

In the pre-flop raise scenario (right), There are several differences. Jack-Jack now has a high weight, since it would have been raised pre-flop, then bet on the flop. Other hands like Ace-Ace, and Ace-King are much higher than they were in the other scenario, since they too are hands that are more likely to be raised pre-flop rather than called. Hands like Four-Four are weaker than in the calling scenario, since Four-Four is less likely to be raised in the pre-flop round.

Noise-Factors

Each action a player makes conveys some form of information, however little, about the hand they are holding. The reweighting process attempts to modify the table based on this information. In order to do this accurately, we need to know how much information the observed action gives us, and how much this information should alter the contents of the weight table.

A *noise-factor* is used to dampen the amount of reweighting done to a table with each action. The noise-factor is a value between zero and one, that is extracted from the probability distribution $\{Pr(fold), Pr(call), Pr(raise)\}$ used in reweighting, and then smeared into the distribution equally. For each action a in the probability distribution D, a is updated by $a \Leftarrow a - a * \beta + \frac{\beta}{|D|}$, where β is the given noise factor.

A noise-factor of 0.0 leaves the probability triple untouched, whereas a noise-factor of 1.0 would flatten the triple into a uniform distribution. For instance, suppose we have a triple of {0.0, 0.2, 0.8} and adjust it with a noise-factor of 0.5. The resulting triple would become roughly {0.167, 0.267, 0.567}. That is, half of the distribution has been proportionally smeared toward uniformity (half of one-third is 0.167).

The noise-factor represents the amount of uncertainty we have about the opponent's actions reflecting information about the cards they hold. If an opponent always plays without deception, using a noise-factor would be detrimental. However, if an opponent plays deceptively, some uncertainty must be kept in the weight table.

3.3.4 Pot Odds

> *'A new car built by my company leaves somewhere traveling at 60 mph. The rear differential locks up. The car crashes and burns with everyone trapped inside. Now: should we initiate a recall?*
>
> *You take the number of vehicles in the field (A) and multiply it by the probable rate of failure (B), multiply the result by the average out-of-court settlement (C). A times B times C equals X. If X is less than the cost of a recall, we don't do one.'*
>
> *'Are there a lot of these kinds of accidents?'*
>
> *'You wouldn't believe.'*
>
> *'... Which car company do you work for?'*
>
> *'A major one.'*
>
> – Fight Club (1999)

Returning to our earlier example where Player A has K♣-K♠, and player B has 7♡-9♡, with a flop of 8♡-6♣-4♡, recall that B's hand has high potential, but weak strength. Should B call A's bet? Player B has a 32% chance of his draw coming in on the turn. If the pot size is currently \$50, with \$10 bet by player B, then it is correct for B to call the \$10. B is getting a 32% chance to win with a 5-to-1 bet.

For a one bet investment, we will win 5 bets 32% of the time. The remaining 68% of the time we miss our draw – at worst we can fold and have lost only 1 bet each of these times. The return on our investment is $(\$50 \times 0.32) - (\$10 \times 0.68) = +\$9.20$.

There are several different ways to estimate pot odds. For instance, the *immediate pot-odds* (the current odds for calling a bet), is formulated as:

$$immediate\_pot\_odds = \frac{amount\_to\_call}{pot\_size + amount\_to\_call} \tag{3.2}$$

Using this equation, our pot odds are $(\$10/(\$50 + \$10)) \approx 0.167$. If we call in situations where we have more than a 16.7% chance of making a winning draw, then the expected payoff is positive. For instance, if we only had a 15% chance of winning, we would not have sufficient odds to call. We would win 5 bets only 15% of the time, and lose 1 bet 85% of the time for a net loss of $(\$50 \times 0.15) - (\$10 \times 0.85) = -\$1.00$.

Pot odds give the return on investment for calling a bet. We must have a chance of winning greater than the pot odds in order to win money.

Pot odds calculations should also take future betting rounds into account. If we make our hand on the next round, there is a good chance we will be able to win a pot much larger than the current size. The *implied pot odds* account for the payoff from winning future bets as well as the current pot. There is no simple formula to measure implied pot odds – players rely on judgment to calculate these odds.

Conversely, the *reverse implied odds* are used in situations where we are playing with a made-hand, but against an opponent who may be on a draw or may already have us beat. In the next betting round the opponent may miss their draw and fold, or they may make the draw or already have you beat in which case it will cost you more to see the showdown – increasing the cost of your investment to win the pot.

3.3.5 Multi-player Considerations

The *Hand Strength* computation gives the strength against one random hand. If the enumeration is weighted, it gives the strength against a player with the hand distribution reflected by the weight table. If we are up against more than one opponent then we must, of course, beat all of them. If a weight table is kept for each opponent, then a good approximation is to multiply the hand strengths for each weight table together. This is the probability that we have a hand better than player one, *and* player two, through to player n.

$$HS_n = HS_{p1} * HS_{p2} * \ldots * HS_{pn} \tag{3.3}$$

This is not an exact value, because the hand strengths of the opponents are not independent of each other. For instance, the weight tables do not reflect the fact that two players cannot hold the same cards. However, the full computation is too complicated for a real-time system. Tests comparing the two methods in [34] showed that the error from Equation 3.3 was small (less than 3%).

Another data structure that is used for multi-player reasoning is the *field array*. This is another weight table that represents the average of all opponent weight tables. This allows us to reason about the whole field of opponents at once, and is a further approximation that reduces accuracy in exchange for speed. For instance, a hand strength against the field array, raised to the power of the number of opponents, can be a reasonable estimate of strength.

$$HS_n = (HS_{field})^n \tag{3.4}$$

The field array has been found to be a large source of error in *Loki*. Recent versions of *Poki* have eliminated most of the dependence on the field array, using it only when the regular calculations are too expensive.

A fundamental problem with the field array in *Loki* is that it is an average of the opponent weight tables. This seems to be the wrong abstraction. Suppose we are up against four opponents, and one has a very high probability of holding a strong hand such as a pair of Aces. Further, suppose the three other opponents have very low probabilities of having such a hand, but instead, they all have mediocre hands represented in their weight tables. If the field array is constructed from the average of the four tables, hands like a pair of Aces will have a low value (say, 0.25), while the mediocre hands held by the majority of opponents will have relatively high weights (say, 0.75). This biases the field array in favor of having the weaker hands. What matters most, of course, is whether we can beat the *best* hand in the field. If we cannot beat the pair of aces held by one opponent, it does not matter that we can beat the three other opponents.

Poki has been enhanced by constructing the field array from the maximum of each entry in the table, instead of the average, which is a more realistic approximation. Also, in many places where the field array was used in the past, it has been replaced with the more expensive, but more accurate Hand Strength calculation (3.3).

3.3.6 Effective Hand Strength

For purposes of simplification it can be useful to have a single value for the overall strength of a hand that includes potential. What we want is an estimation of the probability that we will hold the best hand after all the cards have been dealt. We call this the *effective hand strength* (EHS). Equation 3.5 combines the hand strength, negative potential and positive potential, into a single value.

$$EHS = HS \times (1 - NPot) \quad + \quad (1 - HS) \times PPot \tag{3.5}$$

However, in practice, we remove the negative potential from the measure and use Equation 3.6. At first this may seem counter-intuitive. Negative potential is the chance that we are currently ahead, but further board cards will improve the opponent's hand enough to beat ours. When the negative potential is high it is usually necessary to bet anyways since the *free card* danger is high. We must protect our hand and force drawing hands to pay for their draw.

$$EHS = HS \quad + \quad (1 - HS) \times PPot \tag{3.6}$$

3.4 Basic Betting Strategy

The basic betting strategy is a fairly simple formula-based system that uses the hand evaluation described above as input and outputs an appropriate betting action.

First we evaluate our Hand Strength, Potential, and Pot Odds. The formula generates a *probability triple* over the available actions, $\{Pr(fold), Pr(call), Pr(raise)\}$. For instance, a very strong hand might generate a triple such as {0.0, 0.05, 0.95}. This means that in this situation we should call five percent of the time, raise ninety-five percent of the time, and we should never fold. To make a betting decision, a uniform random number is generated between 0 and 1. This value is used to pick an action from the probability distribution.

Randomization is important to make the strategy less predictable [35]. If we always raised with strong hands, folded weak hands, and called drawing hands, we would quickly get taken advantage of by any perceptive opponent. When we raised, opponents would learn to fold, giving us only small pots. When we don't bet, opponents would know to raise, forcing us to fold our weak or drawing hand. By using a properly mixed strategy, we can hide the strength of our hands.

The formula-based strategy makes an attempt to do this, but it is far from perfect. It is also sensitive to the accuracy of the input. If our opponent modeling is poor, the values generated for strength and potential will be bad estimates.

For many hands in Hold'em, the obvious strategy is sufficient. If we have a extremely strong hand ($EHS > 0.85$), we should almost always raise, with some percentage of check-raises. If we have a very weak hand ($EHS < 0.25$), we should almost always fold (unless we have a high potential that gives us correct pot-odds to call, or we think a bluff is in order). Somewhere in the middle, we should mix our strategy to add unpredictability. Some moderate hands should occasionally be raised – this way, the opponent cannot be sure that we have a strong hand when we raise.

The formula is based around two thresholds for calling and raising. For hands that are well above the raise threshold, we raise nearly all of the time, and for hands well below the calling threshold, we almost always fold. Likewise, hands well in the mid-range are almost always called. The hands that are near the thresholds are treated differently. These borderline hands are mixed between calling and raising, or folding and calling, in proportion to their strengths.

This simple betting strategy plays a fairly solid and safe game, but it is too predictable. Any fixed formula will eventually show exploitable weaknesses (unless it is optimal). The formula could be developed further and further, patching weakness and adding more intelligence. This, however, is of minor scientific interest, as it would be little more than an expert system. Not much work has been done to improve the formula based system since *Loki-2* [35]. The formula was only ever intended as a first-cut approximation. It should have been revisited and improved to incorporate more knowledge and information hiding, but it never was. A fully adaptive knowledge-free system is of much more interest as the techniques could be applied more broadly to imperfect information domains.

3.5 Selective Sampling and Simulation Betting Strategy

In perfect information games like chess, minimax game-tree search algorithms such as alpha-beta [38] provide a dynamic method for discovering good moves based on minimal knowledge. The formula based betting strategy described in the previous section can be thought of as the equivalent of a static evaluation function in deterministic perfect information games. Since game-tree search works so well in these domains, the natural step is to test something analogous to game-tree search in the poker domain.

Unfortunately standard game-tree search algorithms like alpha-beta do not work for poker. The nature of the imperfect information game tree makes traditional search of little utility.

In most two-player perfect information games it is effective to search the game-tree several moves ahead, and then call an evaluation function. In these games it is possible to recursively search to find the opponent's best response to our moves. In poker the best response for the opponent is unknown because we do not know what cards they are holding. The minimax principle does not apply because nodes of the game tree are not independent. It is insufficient to call an evaluation function part-way through the tree, and the game-tree is generally too large to search completely

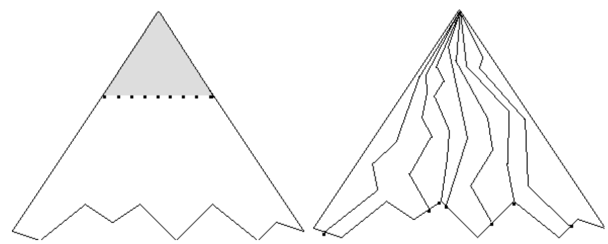

Figure 3.8: The difference between classic game-tree search and simulations.

to the leaves.[3] The leaf nodes of a poker game tree can be evaluated exactly, since the game-state is fully known. However, little can be inferred about the value of internal nodes without searching first to the leaves.

A technique known as *Selective Sampling* can be used to assign probable values to the unknown variables (in poker, the unknown cards). Instead of doing a comprehensive but shallow search, a deep but sparse simulation can probe numerous times to the leaves. Each trial of the simulation selectively samples the search space. With enough trials the cumulative selective sample converges to a statistically confident result.

Selective sampling has been used successfully in other games such as Scrabble [43], backgammon [48], and bridge [20]. *Likelihood weighting*, a similar technique used for Bayesian inferencing algorithms, is another way to bias stochastic simulations [19, 42].

Figure 3.8 shows the difference between standard game-tree search algorithms like alpha-beta (left), compared to the selective sampling simulation method. Alpha-beta will search only an upper region of the tree as deeply as possible, but there is essentially full coverage of the game-tree in the upper region. An evaluation is then applied to the nodes on the frontier. Simulations, however, search a selection of lines to the leaf nodes of the tree. Each line searched samples the game-tree in order to get a statistically relevant expected value for our moves [7].

When *Poki* is faced with a betting decision, it can simulate each of its possible actions, and then simulate the game to the end of the hand. Since we don't know the complete game-state (the cards of our opponents, and the future board cards remain unknown), we must assign probable cards to each player, in each *trial* of the simulation. The weight table of each opponent can be used to selectively bias the chosen cards to match the hands they are likely to hold [35, 8].

After simulating several hundred trials, the average amount won or lost by folding, calling, and raising will typically converge to a stable estimate of the *Expected Values* (EVs) of each action. The action with the largest EV can be chosen as the

[3] For example, Darse Billings has estimated the two-player Texas Hold'em game-tree to be 3×10^{17} in size (counting the number of terminating nodes).

```
simulate() {
   trials = callEV = raiseEV = 0;
   while (trials < MAX_TRIALS) {
      assignCards();
      callEV += simulateHand(call);
      raiseEV += simulateHand(raise);
      trials++;
   }
   callEV = callEV/trials;
   raiseEV = raiseEV/trials;

   return (callEV,raiseEV)
}
```

Figure 3.9: Pseudo-code for poker simulations.

maximal action to take. This, of course, is not a proper mixed-strategy (unless the opponent modeling used in the simulations is sophisticated enough to take the effect of mixed-strategies into account). In practice, the problem of converting the EVs to produce an appropriate mixed strategy remains.

A simulation-based betting strategy for poker proceeds as follows. For each *trial*, probable hands must be assigned to each opponent to hold. These hands can be picked from the distribution of hands in each opponent's weight table. We also choose hands for players that have already folded. This effectively changes the composition of the remaining deck by removing cards that are consistent with folded hands. Once each simulated player has a hand, the unknown board cards are chosen randomly from the remaining cards in the deck.

For each trial, the hand is simulated twice. In the first simulation, *Poki's* first action will be a call (or check if there is no bet). In the second simulation, *Poki* will raise (or bet) for its first action. Once the first action is made, the rest of the hand is simulated to completion. When a simulated opponent must act, their opponent model is asked to predict what the opponent would likely do in the current context, with the hand we have assigned to them. Once the simulated hand is finished, the net amount won or lost is recorded.

After many of these trials the average EV for calling, and raising becomes fairly stable. This must be accomplished within the time constraints of a poker game (usually one or two seconds), limiting the number of trials (typically 200-300 trials for flop decisions).

This technique has several benefits over the formula-based system. It is not a fixed strategy – depending on the opponent models, it can shift dramatically, and adapt to any mix of different opponents. Computing properties like implied pot-odds and draw-odds can be difficult and expensive. Simulations uncover this information naturally, and produce a simple EV for each action. Furthermore, simulations can uncover complex strategies without any specific expert knowledge. For example, tactics such as *check-raising*, *slowplaying*, and *bluffing*, are discovered as an emergent property of the simulations.

The mix of possible cards throughout the trials helps provide realistic estimates of the value of our hand and actions. This process can work effectively, but it is also sensitive to errors and biases both in the simulation, and in the opponent models. Consider the situation where an opponent model incorrectly predicts that the opponent will fold 30% of the time in a particular context, but in actuality the opponent folds only 20% of the time. 10% of the simulations will erronously predict that by raising, *Poki* will immediately win the money in the pot uncontested. This bias will greatly inflate the EV for raising and *Poki's* resulting behaviour will be far too aggressive. Or consider a situation where the opponent model's weight table is representing mostly mediocre hands, when in fact the opponent has *slowplayed* a very good hand. The simulations will consistently pick the wrong set of cards for that opponent, making the EVs unreliable and resulting in a greater loss.

However, given a correct opponent model, the simulation-based betting strategy works well. In general, improvements in the opponent models yield improvements in the accuracy of the simulations. Simulations are sound in principle, but they have not been robust in practice. Chapter 6 will revisit the problems with simulations and present a more robust formulation based on our new *Miximax* algorithm.

3.6 Summary

Poki is a complete reimplementation of the *Loki* AI. The fundamental algorithms for determining hand strength and hand potential involve weighted enumerations of the unknown cards. Weight tables are maintained for each opponent. These tables reflect the hands consistent with their actions throughout the hand. Two different betting strategies have been used to choose actions based on our hand evaluation. The formula based strategy is simple and well tested. The simulation based strategy is a much more adaptive strategy, but it is dependant on quality opponent modeling to get good results. The opponent modeling components of *Poki* will be discussed in the next chapter.

Chapter 4

Opponent Modeling

Knowledge is power, if you know it about the right person.

– Erastus Flavel Beadle (1821-1894)

No poker strategy is complete without a good opponent modeling system. A strong poker player must develop a dynamically changing (adaptive) model of each opponent, to characterize their style and identify potential weaknesses.

In traditional games, such as chess, this aspect of strategy is not essential to achieve a world-class level of play. In perfect information games, it is sufficient to play an objectively best move, without special regard for the opponent. If the opponent plays sub-optimaly, then continuing to play objectively good moves will naturally exploit those errors. Opponent modeling has been studied in the context of two-player perfect information games, but the research has not led to significant performance benefits [11, 21, 22].

The situation with poker is much different. For example, one opponent may bluff too much, the other too little. We adjust by calling more frequently against the former, and less frequently against the latter. To simply call with the optimal frequency would decline an opportunity for increased profit, which is how the game is scored. Even very strong players can employ radically different styles, so it is necessary to deduce each opponent's basic approach to the game, regardless of how well they play.

In poker, opponent modeling is used in at least two different ways. First, we want a general method of deducing the strength of the opponent's hand based on their actions. For example, we would like to know if an opponent will bluff repeatedly on each round. Secondly, we would like to predict their specific action in a given situation. For example, if we are considering a bluff on the river, we will want to know the probability that the opponent will fold or call us down.

At the heart of an opponent modeling system is a *predictor*. The predictor's job is to map any given game context into a probability distribution over the opponent's potential actions. In limit poker, this distribution can be represented by a *probability triple* $\{Pr(fold),\ Pr(call),\ Pr(raise)\}$. Guessing the next action is useful for planning advanced betting strategies, such as a *check-raise*, and is also used in each trial of a full-information simulation.

The difficulty of this task may not be apparent at first. Learning how an opponent plays poker, and how they will play poker in the future, is a formidable challenge for a multitude of reasons:

Uncertainty:

The numerous unknown cards in a game adds a great deal of noise to the game, making it extremely difficult to extract a signal. Each hand of poker can be completely different than the previous ones, simply from the variance of the cards dealt in the hands and on the board. A large number of hands must be played before even the most common situations are encountered several times.

Missing Information:

We only get to see a player's cards if there is a showdown (usually only a small percentage of hands are revealed in a showdown). This means we can only fully verify our model on the few hands that our opponents take to the showdown (which is a heavily biased sample of the total number of hands).

Unknown Dimensions:

There is an unknown number of variables that affect how a player will play a hand. Some factors are much more important than others, and some factors are only relevant to particular players. For one player, their position in the betting round may be a strong criteria for their betting decisions, while position may not be a strong consideration for another player. A particular opponent may favour heart flush draws over flush draws with other suits. The day of the week could even be a factor for some players. One must always be on the lookout for significant correlations.

Intuition:

After a small number of observations, humans form elaborate theories about an opponent, based mostly on intuition and past experience. If a theory holds in future observations, it becomes integrated and used in future betting decisions. A human may even alter their play in order to test a theory. In contrast, machine learning methods typically need a large number of observations, and build a model from statistical inferencing. These statistical methods are passive and slow to adapt.

Multiple Levels:

If we observe an opponent in hands against another player, it may be wrong to predict that they will play similarly towards us. Assuming an opponent models his opponents, they will adjust their play to match each specific opponent. Not only does one need to model the opponent in general, but how the opponent models others (and we can regress further to how the opponent models how we model them, and so on...).

Moving Targets:

We can only learn about an opponent from past observations. A good opponent will change strategies over time. It may be the case that our model is obsolete by the time we have made enough observations to make good inferences about the opponent.

Opponent modeling in poker appears to have many of the characteristics of the most difficult problems in machine learning – noise, uncertainty, an unbounded number of dimensions to explore, and a need to learn quickly and generalize from a relatively small number of training examples, often with missing or misleading data. Furthermore, the real-time nature of poker (a few seconds per betting decision) limits the effectiveness of some popular learning algorithms.

4.1 Methods of Prediction

There are several ways of looking at the opponent modeling problem in poker. From a machine learning point of view, it is convenient to think of an opponent model as a predictor of the opponent. Given the knowledge of the opponent's cards, a perfect model would be able to correctly predict what action the player would choose at each decision. The next best thing would be to predict an accurate probability distribution over the actions they might choose.

4.1.1 Expert Systems

One way to predict an opponent action would be to use our own betting strategy, or some other set of rules, to make a rational choice on behalf of the opponent. When we use this type of fixed strategy as a predictor, we are assuming the player will play in a particular "reasonable" manner, and we refer to it as *generic opponent modeling*. This is not a very effective method of modeling, but it provides a decent baseline. It is a useful starting point for modeling when playing an opponent for the first time.

4.1.2 Statistics

Another obvious method for predicting opponent actions is to expect them to continue to behave as they have done in the past. For example, if an opponent is observed to bet 40% of the time immediately after the flop, we might infer that they will normally bet with the top 40% of their hands in that situation (perhaps including a certain percentage of weak hands that have a good draw). When we use an opponent's personal history of actions to make predictions, we call it *specific opponent modeling*. As one would expect, specific opponent modeling outperforms generic opponent modeling [35].

Our first opponent modeling effort was based on the collection of simple statistical information, primarily on the betting frequencies in a variety of contexts. For example, a basic system distinguishes twelve contexts, based on the betting round (pre-flop, flop, turn, or river), and the betting level (zero, one, or two or more bets).

This history table is essentially a set of conditional action probabilities such as $\mathcal{P}(Call \mid River \ \& \ OneBetToCall)$. However, this is a very limited definition of distinct contexts, since it does not account for many relevant properties, such as the number of active opponents, the relative betting position, or the texture of the board cards (*eg.* whether or not many draws are possible). Establishing a suitable set of conditions for defining the various situations is not an easy task. There are important trade-offs that determine how quickly the algorithm can learn and apply its empirically discovered knowledge. If a context is defined too broadly, it will fail to capture relevant information from very different circumstances. If it is too narrow, it will take too long to experience enough examples for each scenario, and spotting general trends becomes increasingly difficult. Equally important to deciding how many equivalence classes to use is knowing what kinds of contextual information are most relevant in practice.

Furthermore, there are many considerations that are specific to each player. For example, some players will have a strong affinity for flush draws, and will raise or re-raise on the flop with only a draw. Knowing these kinds of personality-specific characteristics can certainly improve the program's performance against typical human players, but this type of modeling has not yet been fully explored.

4.1.3 Neural Networks

To create a more general system for opponent modeling, we implemented a *neural network* for predicting the opponent's next action in any given context. A neural network is a popular machine learning data structure, loosely inspired by biological neural structures.

Neural networks have several nice qualities, but of course there are also many caveats. They are easy to build and train without any domain-specific knowledge.

They typically provide reasonable accuracy, especially in noisy domains. However, they rarely can produce better results than a more formal system built from specific domain knowledge. Furthermore, it can be difficult to extract the knowledge a neural net has learned from a training set.

Neural Networks have been used with success in backgammon programs [49]. The networks in *TD-Gammon* were trained by playing over a million games against itself. The neural network's ability to handle noisy domains was well suited to backgammon, which has stochastic elements. *TD-Gammon* was able to achieve a world-class level of play using this approach.

Structure

A neural network consists of neurons, or nodes, which are connected to each other by weighted directed edges [29, 30]. Input nodes (containing the problem data) are connected to internal (or 'hidden') nodes, and those internal nodes are connected to a set of output nodes. The weighted connections, which can be both negative and positive, transform the signal from the input nodes, through the internal nodes, and to the output nodes. The internal nodes take the sum of all connections entering them, and use a threshold function to decide the output sent to all outgoing connections. For instance, if the sum of inputs is above a threshold, it may output the value '1', otherwise it outputs '0'. More gradual functions can be used such as a sigmoid function which gives a smoother output.

Figure 4.1 shows a simple neural network consisting of four inputs, two internal neurons, and one output neuron. The strength of the inputs are represented by the amount of shading in the input neuron. Connections are shown between each node, where the thickness represents the weight of the connection. Black connections have positive weights, while grey connections have negative weights. In this example, the internal neurons use a threshold function to decide their outputs, and the output neuron uses a sigmoid function to decide its output. Note that in this example, the two input neurons have different thresholds for firing. This is known as a *bias*, and each neuron can have a positive or negative bias affecting it.

Depending on the input values, and the relative values of the weights, each of the internal neurons may or may not fire, which in turn will effect the value of the output neuron. The first internal neuron may only fire when a particular pattern of inputs is detected. The trick is building the correct set of weighted connections that cause the desired behaviour of the network.

Backpropagation

Fortunately, there is a gradient-descent learning algorithm for neural networks called *backpropagation* [37]. Backpropagation is a method of working backwards through

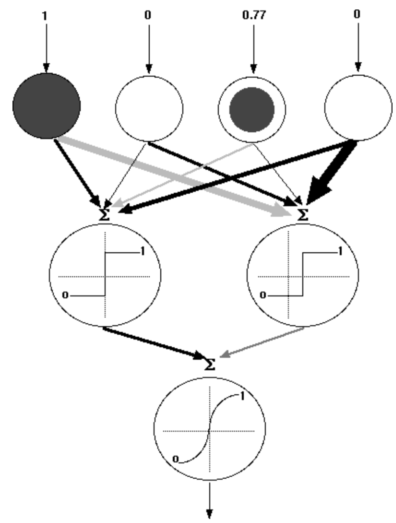

Figure 4.1: A simple neural network.

the neural network, starting from the output and ending back at the input. For a given training example the output on node i, O_i, is compared to the correct answer to determine the the network's error E_i. Starting with the weights at the output nodes, the weights are adjusted in small increments so that the error is reduced. A weight is adjusted proportionally to its contribution to the error.

To assess the blame for a weighted connection ($W_{j,i}$) from an internal node j to an output node i, the error at the output node (E_i) is multiplied by the derivative of the activation function g along the input value (I_i) to the node.

$$\Delta_i = E_i * g'(I_i) \tag{4.1}$$

This determines both the direction of the error, as well as the share of the blame this weight had on the error realized at the output. The weight is updated by adding the Δ_i (*i.e.* its share of the error) times the activation level (O_j). A learning rate α is also used to increase or decrease the amount of change made to the weight for each training example. A high learning rate will allow the network to converge more quickly, but it may overstep the correct configuration.

$$W_{j,i} \Leftarrow W_{j,i} + \alpha * O_j * \Delta_i \tag{4.2}$$

After updating the weights in the output layer, the blame for the errors at the output layer is propagated into the internal layer, since it is likely to share in causing the error. The blame for error at an internal node connection is determined as:

$$\Delta_j = g'(I_j) \sum_i W_{j,i} \Delta_i \tag{4.3}$$

The new delta is computed by propagating the deltas from the output layer. The reweighting rule is essentially the same as the output layer reweighting equation [38].[1]

With each training example, the error in the network is determined and the weights are modified using this backpropagation method. The learning rate α determines the magnitude of the changes made during reweighting. As a network is trained on each sample, the weights are adjusted to minimize the error. All of the weights in the network are modified proportionaly with respects to their relative contribution to the total error. After several training iterations, the network usually converges to a set of weights that minimizes the error over all examples in the training set.

Poker Neural Nets

A standard feed-forward neural net (also known as a *multi-layer perceptron*) was trained on contextual data collected from online games against human opponents. The networks contain a set of eighteen inputs corresponding to properties of the game context, such as the number of active players, *texture* of the board, opponent's position, and so on (Table 4.1). The output layer consists of three nodes corresponding to the fold, call, and raise probabilities. By graphically displaying the relative connection strengths, we are able to determine which input parameters have the largest effects on the output. After observing networks trained on many different opponents, it is clear that certain factors are dominant in predicting the actions of most opponents, while other variables are almost completely irrelevant.

The inputs encode public knowledge of the game context. All inputs range from values of zero to one. For instance the current betting round is encoded using two Boolean inputs. If it is the flop, then both input #5 and #6 are zero.[2] If it is the turn, then input #5 is set to one. If it is the river, input #6 is activated. Inputs #3 and #4 signal if there are bets to call, and inputs #9-12 encode information about the number of players and positions.

Inputs #13 and #14 are the estimated hand strength and potential of the cards the opponent holds. Inputs #15 and #16 indicate what the formula based system would do in this context. In cases were the opponent's cards are unknown, a representative hand is chosen from the opponent's weight table. This hand is used to generate the estimated hand strength and potential.

Figure 4.2 shows a typical neural network after being trained on a few hundred hands played by a particular opponent. The inputs are on the top row, with the activation level ranging from zero (fully white) to one (fully black). The thickness

[1] For a supplementary explanation of backpropagation, see the Appendix B.

[2] Pre-flop actions are not modeled by the neural networks. We are only concerned with post-flop prediction here.

#	Type	Description
0	real	immediate pot odds
1	real	bet ratio: $bets/(bets + calls)$
2	boolean	committed (has put money in the pot this round)
3	boolean	one bet to call
4	boolean	two or more bets to call
5	boolean	betting round = turn
6	boolean	betting round = river
7	boolean	last bets called by player > 0
8	boolean	player's last action was a bet or raise
9	real	$0.1 * numPlayers$
10	boolean	active players is 2 (heads-up)
11	boolean	player is first to act
12	boolean	player is last to act
13	real	estimated Hand Strength for opponent
14	real	estimated Hand Potential for opponent
15	boolean	expert predictor says they would call
16	boolean	expert predictor says they would raise
17	boolean	Poki is in the hand

Table 4.1: Neural network inputs.

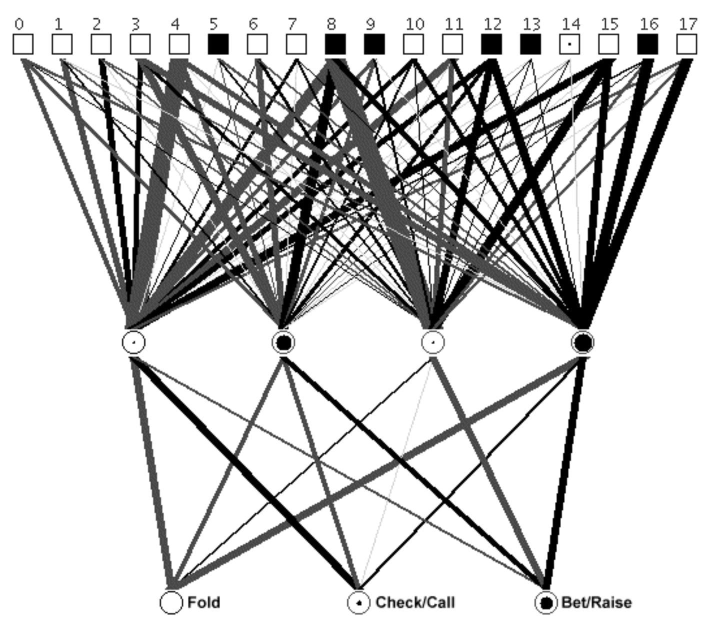

Figure 4.2: A neural network predicting an opponent's future action.

of the lines represent the magnitude of the weights (black being positive, grey being negative). In this example, the connections from input node number eight (true if the opponent's last action was a raise) are very strong, indicating that it is highly correlated with what the opponent will do next. The bottom row shows the network predicting that the opponent will probably bet, with a small chance of checking.

By examining the graphical representations of neural nets trained on an opponent, the public game features that have the highest correlations with the opponent's actions can be spotted. Inputs that are strongly correlated with the actions, will have relatively stronger weights (*i.e.* line thickness). Early experiments helped reveal strong indicators for detecting how an opponent will play. It also became clear that many strong factors could differ significantly between each specific opponent.

The first result of this study was the identification of new features to focus on when modeling common opponents. This produced a relatively small set of context equivalence classes which significantly improved the statistical opponent modeling reported previously [13]. It was discovered that the player's last action is highly correlated with their future actions. This makes intuitive sense – if a player has just bet, there is a very small chance they will fold, and a much larger chance that they will call or raise. Likewise, if their last action was to check, then there is a much higher chance they will fold, rather than raise. As a result, the player's last action was added to the context of the action frequency statistics. This improved the accuracy of the statistics by 10-to-20% on average.

Perhaps the biggest drawback of neural networks, for our purposes, is that they do not output a proper probability distribution over the likely outputs because they are not trained on a probability distribution, but the actual events. This skews the output to represent the most likely action rather than the most accurate distribution over the possible actions.

4.1.4 Decision Trees

Decision trees are another good way to solve classification problems such as the poker action prediction problem. A decision tree is navigated by asking a question at each node, and depending on the answer, traversing down a particular branch, towards a leaf node. At a leaf, there is a count of the frequencies of choices in the training set that would fall under that leaf node.

Figure 4.3 shows an example of a decision tree. Given a set of training data, a tree can be created to classify the data based on some simple rules. A typical method is to start with a single node, then split the node on one of the features such that the feature used results in a split that maximizes the information gain.

In a past feasibility study, we used Paul Utgoff's *Incremental Decision Tree Induction* software [51] to test decision trees on action prediction. Due to the high

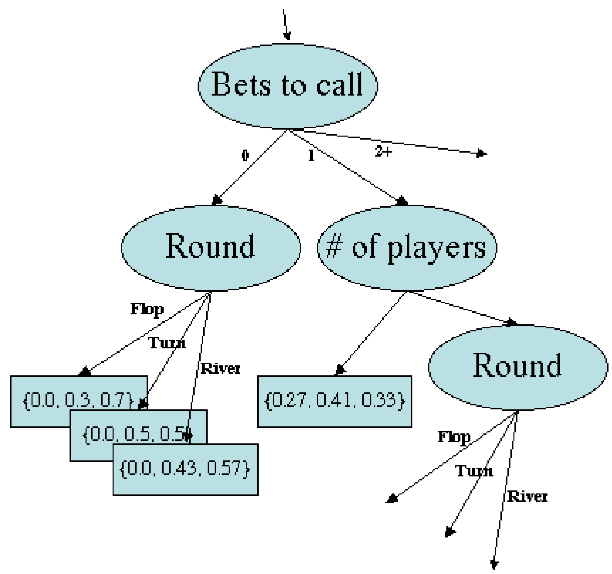

Figure 4.3: Part of a basic decision tree, with probability triples at the leaf nodes.

amount of noise in poker data, all training was done with tree pruning. Pruning after training helps decrease the possibility of over-training and generally improves performance as a result. Preliminary tests on smaller data sets clearly indicated that the decision trees with pruning returned more accurate predictions of the opponent's next move.

While not as robust or noise-tolerant as neural networks, decision trees have several benefits. They can output accurate probability distributions over the available choices, which neural networks cannot do. A decision tree's representation is also a more human-understandable format, so determining what knowledge has been encoded in the tree is much easier.

In preliminary offline testing, decision trees achieved accuracies slightly below the neural networks. While *Poki* does not have any decision-tree software built in, it is a potential avenue for exploration in the future.

4.2 Confusion Matrices

In order to examine what types of classification errors the predictors are making, we construct *confusion matrices*. A confusion matrix displays the distribution of classifications across the combinations of predicted actions and the correct actions. With this matrix, one can see what percentage of test examples were expected to be *folds* when they were actually *calls*, and so on. The diagonal cells are correct classifications and the totals are shown in the margins.

An important use of these matrices is to see what the predictor has learned. For

		Prediction			
		fold	**call**	**raise**	**%**
Actual	**fold**	**11.2**	1.6	0.0	12.8%
	call	0.0	**25.6**	8.0	33.6%
	raise	1.6	3.2	**48.8**	53.6%
	%	12.8%	30.4%	56.8%	**85.6%**

Table 4.2: A typical confusion matrix showing the errors made in predicting an opponent.

instance, by examining a matrix one may find that a predictor getting 67% accuracy is merely choosing call 100% of the time, which the opponent happens to do 67% of the time. In this case we can see that all that has been learned is to pick the most frequent action overall. It has not learned how to distinguish the situations in which the opponent folds or raises from that in which the opponent calls.

We may also observe the tendency to make certain specific types of errors with an opponent. For instance, we may see that when the predictor predicts a fold, it is always correct, but ten percent of the time it incorrectly predicts a raise as a call. Table 4.2 shows an example confusion matrix. Certain types of errors are non-existant – the predictor in question never predicts a fold when the opponent actually calls, and it never predicts a raise when the opponent actually folds. It does make errors in three other cases. For example, it sometimes mistakes a raise for a call – that is, when it predicts a call, $\frac{3.2}{30.4} = 10.5\%$ of those times it is actually a raise. Certain types of errors can be more serious than others. For instance, incorrectly predicting folding can lead to massive errors in EV calculations, while confusing a call with a raise will have a much smaller impact.

Knowing which situations confuse our predictors, a program can make a more informed decision. As mentioned above, it is typical to see an opponent with a high bias towards one of the three actions. Some players have been observed to make a particular action as often as 75% of the time. If we imagine using the simplest predictor imaginable – one that simply chooses the most frequent overall action – it is trivial to get better than 50% accuracy for most opponents. One must keep this in mind when evaluating the more advanced learners. The first 50%-60% accuracy is trivial. The next 10-20% can be far harder to reach.

4.3 Multi-Predictor Voting

With all of these different methods of predicting opponents, it can be difficult to know which to use. An elegant way of solving this problem is to use all of the methods, and let them vote. Each vote can be weighted by the current accuracy of the predictor. When one predictor makes a mistake, there is a good chance that some of the others will pick correctly. This helps to reduce errors, and make the system more robust. It has been observed that the multi-predictor voter system is often slightly more accurate than any of the individual predictors.

In *Poki* the *meta-predictor* system dynamically tracks the accuracy of the predictors. The accuracy of the last n actions of a player are used (where n is typically set to a recent range of the last 25–100 actions) to dynamically evaluate the predictors.

Each predictor votes for an action, and each vote is weighted based on that predictor's accuracy. To estimate accuracy, a confusion matrix is computed over the last n actions. If the predictor is predicting a fold, for instance, we can examine the confusion matrix to see how accurately it predicts folding. If this predictor is often wrong about the opponent folding, its vote will not count for much. For the m predictors, each probability triple Φ_i is multiplied by the accuracies for that predictor:

$$\Phi = \sum_{i=1}^{m} \Phi_i \cdot a_i \tag{4.4}$$

For instance, if $\Phi_i = [0.1, 0.85, 0.05]$ and the accuracies $a_i = [0.9, 0.5, 0.4]$, this means there is a 90% chance that the 10% fold prediction in Φ_i is correct, and so on. We treat the probability triple and accuracies as vectors and take the dot product to get a weighted distribution of $[0.09, 0.425, 0.02]$ put towards the final vote.

The resulting weighted sum of predictions, Φ, is then normalized to a proper probability distribution, and used as the final prediction.

This interesting technique adds a second level of prediction, in that it takes both the predictions and knowledge about the prediction quality into account. The *meta-predictor* is sometimes more accurate (by a few percentage points) than any of the individual predictors. It allows for the seamless addition of any number of different prediction systems, and dynamically adjusts its use of each one to adapt to the current game. The system could perhaps be improved with a better, nonlinear weighting function. For instance, the weights given to a predictor with 90% accuracy and to one with 80% accuracy are quite similar, but it might be beneficial to give a much higher relative weighting to the first one.

4.4 Summary

Opponent modeling is a critical component for a world-class poker program. The problem of opponent modeling can be tackled with numerous methods including statistics, neural networks, decision trees, and other novel approaches. *Poki's* architecture defines opponent modeling as an action prediction task. Any method of action prediction may be implemented and plugged into *Poki*. The *Meta-Predictor* system will automatically evaluate each predictor for each opponent and use each predictor accordingly. Action predictions can be used in reweighting to assess the likelihood that an opponent holds a given hand. The predictions may also be used in simulation based betting strategies in order to anticipate future outcomes of a game.

Chapter 5

Experiments

> *The art of drawing conclusions from experiments and observations consists in evaluating probabilities and in estimating whether they are sufficiently great or numerous enough to constitute proofs. This kind of calculation is more complicated and more difficult than it is commonly thought to be...*
>
> – Antoine Laurent Lavoisier (1743-1794)

Measuring the performance of a poker-playing program is difficult. *Poki* is a complex system of interacting components. Changing a single component often has cascading effects, leading to unpredictable and unforeseen behavior. A variety of methods for assessing the program can be employed, but none of them is completely adequate in isolation.

Poker is a game of high variance. The element of luck strongly effects the outcome of each individual hand. Even two players of equal strength may see many anomalous games and long runs of good or bad luck. Measurements are susceptible to large amounts of noise and many thousands of hands are required to gain confidence in any result. To complicate matters, perceptive players are not static – through the playing of several hundred or thousand hands of poker these players will adapt their play. In addition to the challenge of measuring a noisy result, our target is constantly moving as we measure it.

Self-play experiments are commonly used to compare different features, variations or parameters of a strategy. The games in a self-play experiment can be controlled to minimize some of the variance. For instance to reduce the variance from lucky cards, the same hands can be dealt many times, but to a different player each time. The seating order can also be controlled to reduce the effects of positional advantage.

A major caveat of self-play experiments is that the games lack a wide variety of different styles and conditions that human players provide. Other researchers have previously commented on the "myopia" of self-play games in chess [2]. The problem is far more of a hindrance in poker. Poker styles can vary greatly, and the style of the opponent is a key element in playing correctly.

A self-play experiment that proves one strategy dominates another lends little ability to generalize the results beyond the experiment itself. While a player may be able to beat a specific opponent, it may flounder against another. The results are non-transitive: player A beats player B, player B beats player C, yet player C beats player A. This problem is commonly observed when testing and building computer strategies for poker. It is easy to build a program that takes advantage of a strategic weakness in another program. However, such a program will more often than not prove to play weakly against a strategy for which it has not been tuned to beat (recall the *RoShamBo* example about the ability to easily build an *Iocaine Powder* beater that still cannot win a tournament). A program that does very well against normal opponents may be vulnerable to a particular type of erratic or irrational player, even if their play is objectively worse.

Numerous different experiments will be presented in the following sections. First, two different methods of reweighting shall be examined in Section 5.1. In Section 5.2 several tests are conducted to evaluate the opponent modeling methods. The tests include self-play experiments, online games with humans, and offline tests of opponent models. Section 5.3 will discuss the actual performance of the different programs and strategies from online games.

5.1 Evaluation of Reweighting Methods

The accuracy of a weight table is paramount to the success of simulation-based betting strategies and to obtaining accurate hand evaluations for the formula-based strategy. The following experiment was designed to test two different reweighting methods, as well as to find the best table noise levels for heads-up games.

5.1.1 Reweighting System A

In both *Loki* and older versions of *Poki* reweighting used an error-prone method of opponent modeling to predict the action probability distribution for each entry in the weight table. The model would return the general action frequencies for the current game context – for example, a triple of $\{0.2, 0.5, 0.3\}$ on the flop with one bet to call. This triple would then be used to generate the estimated call and raise thresholds for the player by directly mapping the frequencies onto effective hand strengths. Since the opponent raises 30% of the time, our model would state that the opponent will generally raise with hands that have a EHS of 0.7 or greater. The problem with this

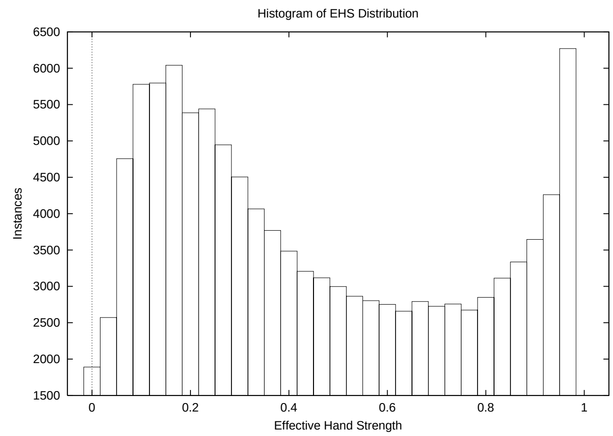

Figure 5.1: A histogram of over 100,000 effective hand strengths (EHS) taken from *Poki* in ten-player online poker games (combined from all post-flop betting rounds).

mapping is that it incorrectly assumes that 30% of the opponent's possible hands have an EHS of 0.7 or greater. Typically, this is not true – the distribution of EHS is not linear, nor is the distribution the same for each board.

EHS Distribution Histograms

Each board has a unique texture determined by the number and types of draws, and the selection of high and low cards. Figure 5.1 shows a histogram of the EHS distribution over a wide variety of boards (the EHS values were collected from thousands of online games). It is clearly a non-uniform distribution. A majority of hands have either low or high strengths, with relatively few hands in between. Intuitively, most hands will either miss-match the board cards, or match to make a strong hand. Figure 5.2 shows a separate histogram for each betting round. Many hands are folded on the flop and turn, so the overall number of instances drops in each progressive round. The overall shape of the distribution remains fairly constant between betting rounds.

5.1.2 Reweighting System B

A new approach was devised to pick more correct thresholds for reweighting. Before reweighting, the EHS of every entry is computed and placed into a sorted list. If the opponent raises 30% of the time, the EHS 30% from the top of the list is chosen as the raise threshold. Likewise, if they fold 20% of the time, the EHS 20% from the bottom of the list is chosen as their folding threshold. This allows the thresholds to be dynamically computed for any EHS distribution.

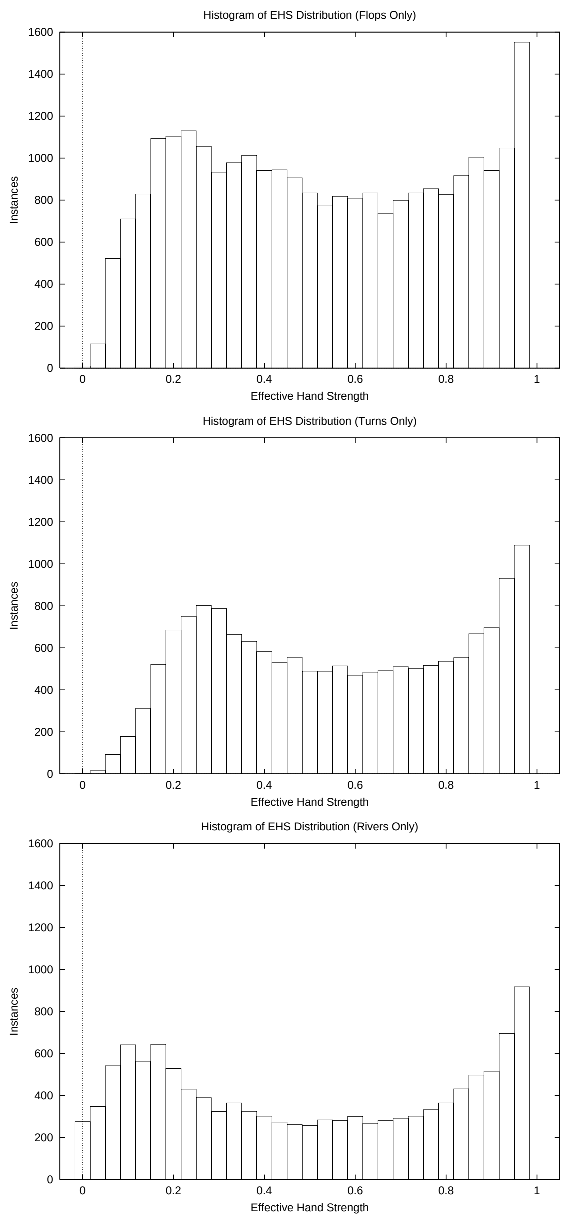

Figure 5.2: Histograms of roughly 55,000 effective hand strengths (EHS) taken from *Poki* in two-player online poker games. Each betting round is shown separately.

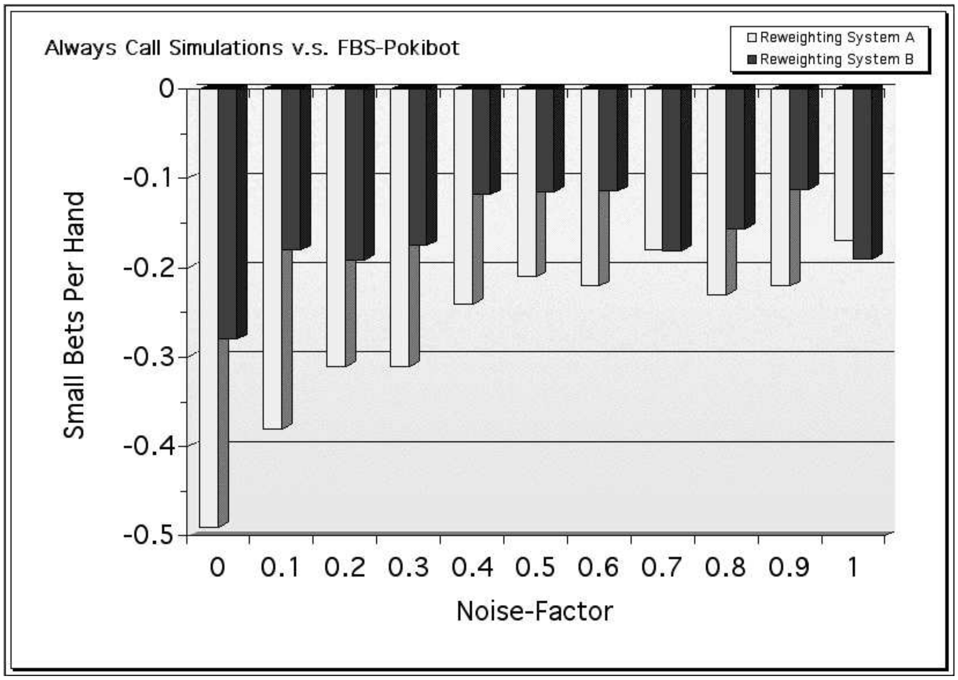

Figure 5.3: Simulation using roll-outs (v.s. FBS-Poki) with different noise factors. All of the SBS programs lose to FBS-Poki, so shorter bars are better (a smaller loss).

5.1.3 Experimental Design and Results

A random number seed was chosen to generate a fixed set of 2000 random hands. A match to evaluate two strategies consists of playing the set of 2000 hands twice. The second time, the two players switch seats so that each player gets the other player's cards from the previous run. This eliminates much of the variance from the experiment, since both players get the same opportunities from any lucky and unlucky cards. For all of the following matches the same set of 2000 hands was used, so that comparisons between different strategies is possible.

Ten levels of reweighting noise-factors were tested in increments of 0.1. A simulation-based strategy (SBS) was tested against a standard formula-based strategy (FBS). The simulation-based player uses no action prediction, but instead always predicts a call, such that the simulation just becomes a roll-out to the showdown. This emphasizes the quality of the weight tables in the simulations as the primary factor in determining expected value. Since it does not simulate future bets and raises it cannot discover tactics like slow-playing or check-raising.

There is a clear disadvantage in using no noise in the reweighting. Without some uncertainty added, the weight table becomes far too committed to a small set of hands. Some noise is needed, otherwise the opponent's actions are given far too

Predictor	Bots	Humans	σ
Statistics I	60.94%	66.76%	0.023
Statistics II	63.94%	69.04%	0.115
Expert Formula	67.87%	55.98%	0.061
Neural Network I	64.67%	79.34%	0.086
Neural Network II	73.49%	79.91%	0.054
Meta-Predictor	75.11%	80.49%	0.023

Table 5.1: Average predictor accuracy in online games.

much credit. A noise-factor of 0.4 seems to be (in this experiment) the best level for heads-up poker, given the current system. The newer reweighting system is also clearly better than the previous version, especially with low noise-factors (with high noise-factors, the reweighting is rendered increasingly irrelevant).

5.2 Evaluating Predictors

In online games played on our poker server, there are both humans and computer programs ('bots') playing in full ten-player games. After each hand, the recent accuracy of each prediction method was logged. The average accuracy of each method is displayed in Table 5.1. Average accuracies (taken from several thousand logged games) for both bots and humans are listed.

The *Statistics I* uses the basic action frequency tracking that was used in *Loki-2* and in older versions of *Poki*, but with an enhanced context [13]. *Statistics II* uses a more sensitive, but similar technique that does not keep statistics for all observed actions, but only the last 40 in each context class. The *Expert Formula* uses *Poki's* own formula based betting strategy to predict the opponent. The only difference between *Neural Network I* and *Neural Network II* is the training data used. In the former, the network is trained on all actions ever observed by the opponent. In the latter, a new network is created and trained from random weights with each new session played with the opponent. That is, if the opponent plays for a few hours one day, and returns to play two days later, NN-I will remember the past sessions, and NN-II is restarted from scratch. The *Meta-Predictor* is using all five of these predictors to form its own predictions.

Note that as expected, *Meta-Predictor* is the best method, faring slightly above the best sub-prediction method that it uses. The standard deviation of the prediction accuracy is also shown for humans. Note that the *Meta-Predictor* has a lower standard deviation than most of the other predictors. This means that it is able to consistently maintain its high accuracy.

The experiment above shows the average, stable prediction accuracies. Another data point is needed to see how quickly the predictors can model a new opponent.

From a set of training data collected from online games, an offline experiment can be done to test the different predictors.

Given a set of training data (containing past game situations, and actions the opponent took) we can use any kind of machine learning technique to try and classify the situations where the opponent took each different kind of action. Once a model has been trained, a second set of test data is used to evaluate the model's accuracy of predicting the actions the opponent will take. This testing data should not be in the training set, otherwise the evaluation may be biased in favour of the model. Testing data should always be on instances of unseen data. This is the classic way of evaluating a machine learning system. However, there are characteristics of our chosen domain that allow us to perform a more realistic evaluation.

Actions in poker are temporally ordered. Opponents may significantly change their methods of play over time, or play particular hands in different ways. In an applied poker system, opponent modeling data is collected one instance at a time. To mimic this chronological ordering, a more precise evaluation is needed. When predicting an opponent's next action, the predictor may draw upon all past data for the opponent. The classic method of dividing the past data into two large sets of training and testing data does not accurately represent the way poker data is available in a real game. Iterating through the data set, we keep count of how many actions we can correctly predict, when all prior items in the set are known, but all future instances are unknown.

This method calculates two types of information. By graphing the cumulative number of correct predictions over time, we can observe how quickly the system learns the opponent's style of play, and we can see how it maintains accuracy as the opponent changes styles. We also have the overall percentage of accurate predictions. This temporal ordering method directly mimics opponent modeling constraints in a live game situation.

In this test, opponent actions from several thousand online games were collected for 24 different opponents. Two of these opponents were bots – one using FBS, the other SBS. The rest were regular human players who had played a sufficient number of hands. In Table 5.2 five methods were used to predict these opponent's actions. In the classic evaluation, the data set for an opponent was split into two equal portions. The first half was used to train the models and the second half was used to test the accuracy. In the incremental test, the predictor was asked to predict each item in the data set one at a time, and told the correct answer after guessing. The difference between the two results was surprisingly similar. On average, the Meta-Predictor performed the best, by a small margin, followed by the Neural Networks.

The 'always call' predictor, as the name implies, always predicts a call. Since this is, by far, the most common action made in the game, it sets a good base-line.

Predictor	Classic	Incremental
Always Call	57.4%	57.4%
Statistics Loki-2	57.5%	57.5%
Statistics I	60.4%	60.6%
Statistics II	60.7%	60.9%
Expert Formula	54.7%	54.9%
Neural Network	68.4%	66.1%
Meta-Predictor	70.4%	69.5%

Table 5.2: Average predictor accuracy.

We can see that it is trivial to attain an average accuracy of 57.4%. It takes much more effort to gain any additional accuracy.

5.2.1 Self-Play Results

A self-play experiment was conducted to assess different predictors for the simulation based strategy. The conditions were exactly the same as the experiment above where two versions of the SBS played against *FBS-Poki*. The SBS players all used the newer reweighting method discussed in Section 5.1. The three different prediction methods were *Always Call*, the *Expert Formula* (*i.e.* the FBS was used as a predictor), and the *Meta-Predictor* which encapsulated several predictors (statistics, expert formula, neural networks).

The results are shown in Figure 5.4. Again, the SBS players all lost to *FBS-Poki* in these self-play matches. Overall, the *Meta-Predictor* SBS lost the least. The *Expert-Formula* SBS was not too far behind, although one would expect this predictor to perform well against *FBS-Poki* since the predictor is nearly identical to *FBS-Poki's* actual strategy. Against other opponents, *Expert Formula* would not perform as well.

5.3 Performance Results from Online Games

Perhaps the best way to test a program is by playing games against a variety of strong human players. A poker program can participate in a real game with willing participants, using a laptop computer on the table. This turns out to be surprisingly difficult, due to the fast pace of a real game and the amount of information to be entered. Even with numerous single-character accelerators, text entry is a bottleneck to the process. A well-designed graphical interface might help considerably, but it is currently more practical to have human players participate in a virtual game, rather than having programs compete in the physical world. Online games with humans have been our best method of empirically assessing *Poki*.

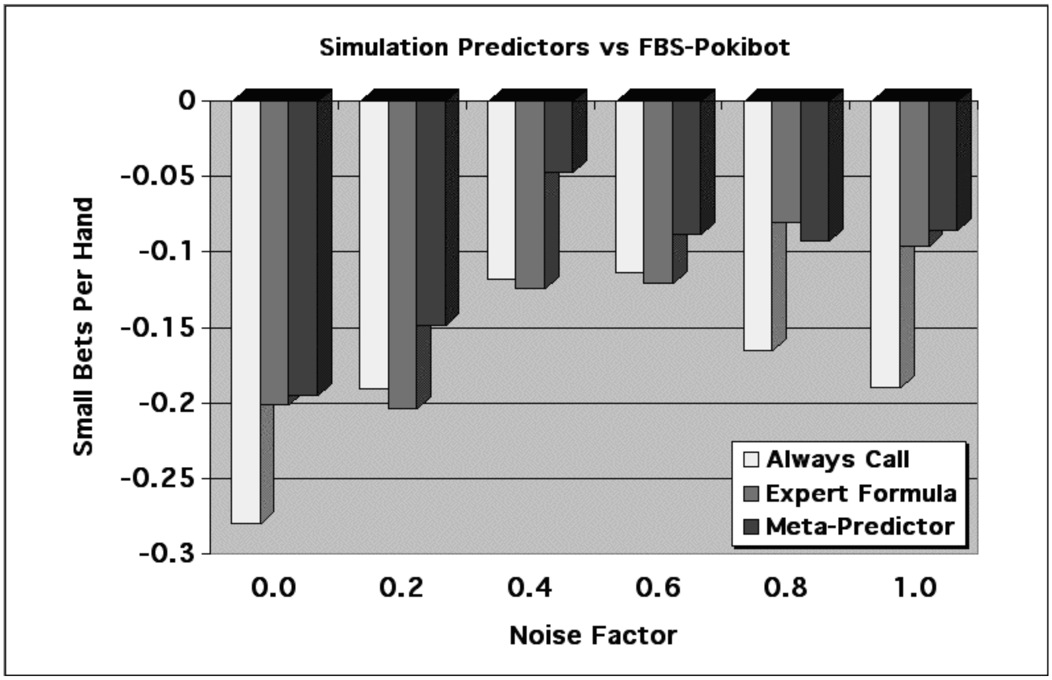

Figure 5.4: Performance of three different simulation predictors against FBS-Poki. Shorter bars are better (a smaller loss).

5.3.1 IRC Poker

Poki played frequently on Internet Relay Chat (IRC) poker against humans and the occasional other 'bots' (as online AI programs are named). A special IRC server allows humans and bots to connect and play poker games against each other online. No real money is at stake, but the accumulated bank-roll for each player is preserved between sessions, and a variety of statistics are maintained. There is a hierarchy of games for limit Hold'em, and a player must win a specified amount in the introductory level games to qualify for the higher tiered games.

The lowest level games (open to everyone) vary from wild to fairly normal, offering a wide variety of game conditions to test the program. The second and third tier games resemble typical games in a casino or card room. Most of these players take the game seriously, and some are very strong (including some professionals). Since *Poki* has been a consistent winner in these higher tiered games (and is in the top 10% of all players on the server), we believe the program plays better than the average player in a low-limit casino game.

Performance is measured in small bets won per hand (sb/hand). This measure is independent of both the speed of the game (hands per hour) and the bet sizes. This enables comparison between many different games. Since no variance reduction methods are available for online games, we generally test new algorithms for a

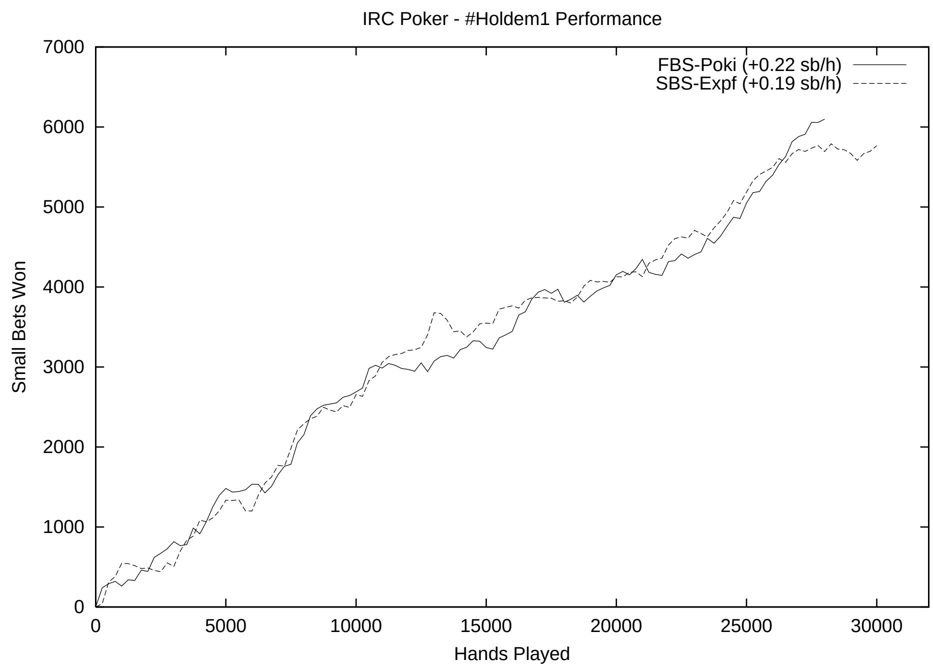

Figure 5.5: Collected from IRC poker #holdem1 in early 2001.

minimum of 20,000 hands before interpreting the results. On this scale, the trends are usually clear and stable amid the noise. Unfortunately, it can take several weeks to accumulate this data, depending on the popularity of the online game in question.

In recent play on the IRC poker server, *Poki* has consistently performed between +0.10 and +0.20 sb/hand in the lowest level games, and between +0.07 and +0.10 sb/hand in the higher tiered games against stronger opposition. In a game of \$10/\$20 Hold'em with 40 hands per hour, an income rate of +0.10 sb/hand translates into \$40 per hour.

The results of simulation-based betting strategies have so far been inconsistent. Despite some programming errors that were discovered later, the earliest (1998) versions of simulation-based *Loki* outperformed the regular formula-based version in both self-play experiments ($+0.10 \pm 0.04$ sb/hand), and in the introductory level games of IRC (+0.13 sb/hand *vs* +0.08 sb/hand). However, it lost slowly in the more advanced IRC games, whereas the regular version would at least break even.

Figure 5.5 shows that both the regular FBS betting strategy (labeled "FBS-Poki") and the simulation-based betting strategy (labeled "SBS-Expf") win at about +0.20 sb/hand in the introductory level games on the IRC poker server. However, there is a clear difference in the more advanced games, where the regular betting strategy routinely wins at about +0.09 sb/hand, but the simulation-based version could only break even (peaking at +0.01 sb/hand after 5000 hands, but returning to zero after 10000 hands).

As the quality of human players on IRC has deteriorated recently and the games have become less frequent, (many of the better players have left the play-money games for the more lucrative real-money online poker rooms) the most recent versions of *Poki* have not been tested on IRC.

5.3.2 Results From the Poki Poker Server

For nearly two years, we have hosted our own online poker server. Although not as popular as IRC Poker, the fact that we can control the server allows for greater flexibility in testing and evaluating different strategies. Using a free Java web applet, users can play at a table with poker programs and other people. Computer programs can support our TCP/IP based poker communication protocol and join in the different hosted games. The mix of humans and different poker programs provides an interesting hybrid between self-play experiments and games against humans.[1]

This game consists of several computer players, and at least one human opponent at all times. Since the artificial players have a style that is quite conservative, this game is quite a bit *tighter* than most IRC games. As a result of the tougher field, the income rates are much lower than in the IRC #holdem1 games. The relative performance of the different players against the same field is more important than the absolute scores.

Figure 5.6 shows some recent results from our server. The top graph is from games played on the server in early-to-mid 2001. The bottom graph is for late 2001 to early 2002. Again, it is important to note that the field grew substantially harder in the second time-period. Some of the weaker bots were replaced with stronger ones, and the site's popularity grew substantially as well, bringing many more strong human players to the game.

The performance of *FBS-Poki* dropped from +0.12 sb/hand to +0.066 sb/hand between 2001 and 2002. Stronger humans find the *FBS-Poki's* play to be fairly predictable after playing a few hundred hands. *SBS-Meta* is a SBS player that uses the meta-predictor in its simulations. *SBS-Expf* uses the FBS as a predictor (i.e the Expert Formula). Not surprisingly, *SBS-Meta* outperforms *SBS-Expf*, earning +0.11 sb/hand, for a 40% overall improvement in this particular game. Although *SBS-Meta's* income rate dropped between the two time periods (+0.11 sb/hand to +0.097 sb/hand), its relative performance to the *FBS-Poki* improved significantly. This is likely due to some small bug fixes and minor enhancements.

The *FBS/SBS* player was designed to be less predictable – for each hand it flipped a coin. If the coin is heads it plays using FBS; tails, it plays as *SBS-Meta*. The *FBS/SBS* program performed, interestingly, in between the FBS and SBS programs with an income rate of +0.075 sb/hand.

[1]See `http://www.cs.ualberta.ca/~games`.

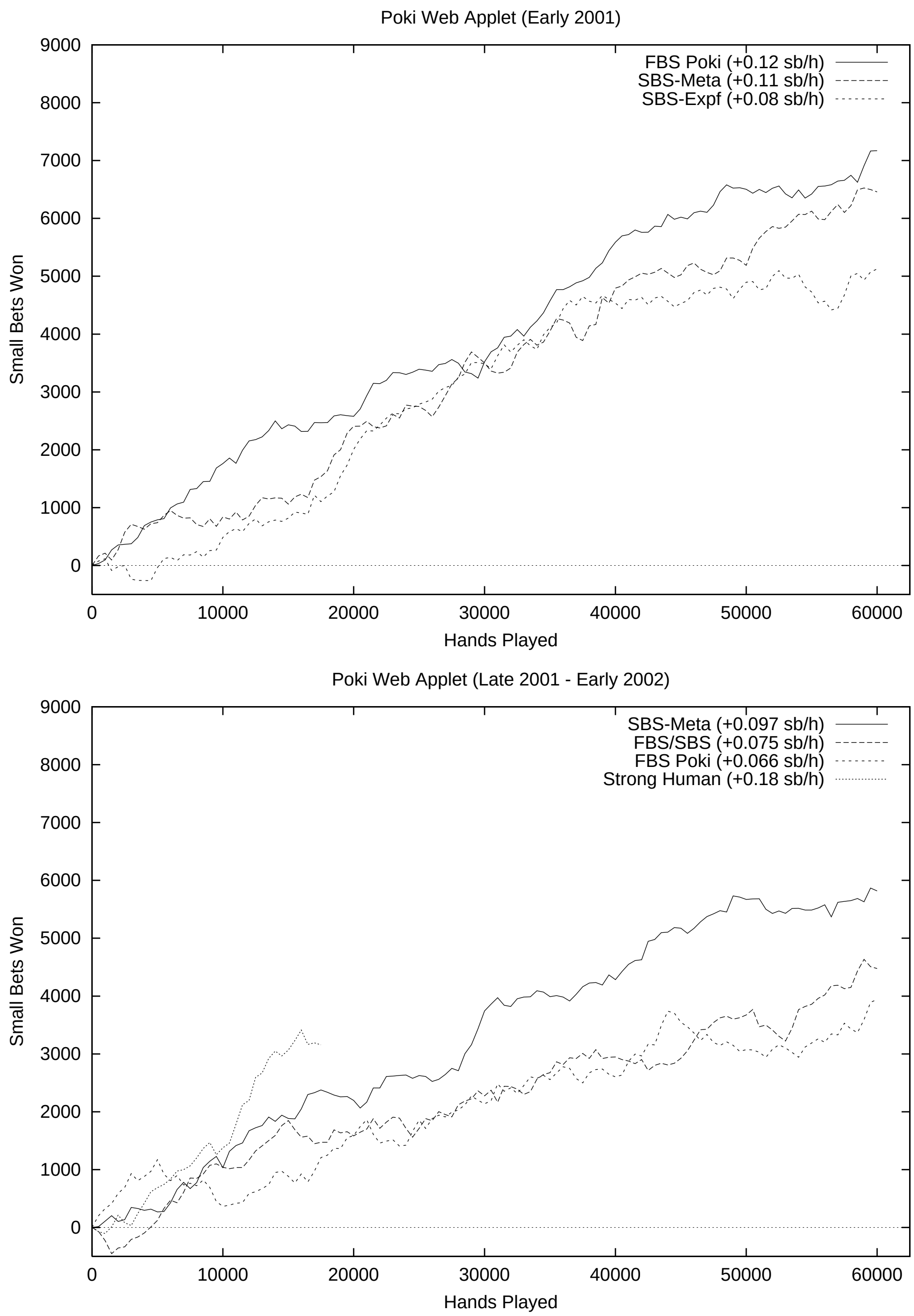

Figure 5.6: Poki poker server results.

One of the strongest human players that plays regularly is also shown in the bottom graph. In 17500 hands, this player has achieved an income rate of +0.18 sb/hand. As one can see, there is much room for improvement in the play of the bots. Very strong human players could likely double or triple this rate.

Poki Poker Heads-up Contest Results

There are many problems with the experiments from IRC and the 10-player room on our Poker Server. For one, the field is always changing. Players come and go, some play terribly, while others are quite strong. Over time, the conditions can change substantially. A more controlled experiment against humans was desired. It was decided that a controlled contest would be a good way to gather results.

In order to test three different versions of *Poki*, a special online contest was conducted. Each contestant played 500 hands of \$10-\$20 Limit Texas Hold'em against each different program. Of the nearly fifty players that attempted to finish the contest, only 11 completed the full 1500 hands required. It is important to note that many of the weaker players became discouraged by their performance and quit before completing the contest. As a result, the eleven players that finished represent some of the better players from our server.

In each room (each version of *Poki* played in its own room), the first 150 hands were random, and the last 350 were the same set of hands. This was done to reduce the variance from one player getting luckier cards than other players. Humans are notoriously good at noticing repeated hands, so the 150 random hands at the start were introduced to make it difficult to notice. Unfortunately, each room had to have a different set of 350 hands, otherwise the participants could unfairly recall hands in the subsequent rooms. This means that while the results between players within a room are comparable, the results between rooms are subject to much higher variance. Designing an experiment in which humans could not cheat or detect repeated cards proved to be quite difficult.

The first bot, *Poki-1*, used the formula based betting strategy, and the new reweighting method. *Poki-2* also used the formula based betting strategy but used the old reweighting system instead. *Poki-3* was our best simulation based betting strategy (at the time). It used the meta-predictor for simulations and also used the newer reweighting system. The overall scores (in small bets won or lost) are shown in Table 5.3. A summary of each bot's performance is shown in Table 5.4. Average winning rates are given for both the total 500 hands and for the last 350 hands.

The results are somewhat surprising, but any conclusions must be taken lightly due to the small sample size and potentially high variance in the cards dealt in each room. The FBS bot with the old reweighting did far better than the other two bots. In fact, it was the only bot to finish with a positive score. There are several

Rank	Poki-1	Poki-2	Poki-3	Total
1	279.0	102.5	321.5	703.0
2	265.0	-42.5	302.0	524.5
3	196.0	-22.5	193.5	367.0
4	12.5	-6.0	115.5	230.0
5	245.0	-103.5	33.0	174.5
6	5.0	-80.0	234.0	159.0
7	-150.0	51.5	180.5	82.0
8	21.0	-62.5	40.5	-1.0
9	-45.5	-138.0	155.0	-28.5
10	-34.0	-151.5	38.5	-147.0
11	-13.0	-267.5	54.5	-226.0

Table 5.3: Poki Poker heads-up contest results (January 2002). The results are for all 500 hands played per room.

Name	Strategy	Reweighting	SB/Hand (500)	SB/Hand (350)
Poki-1	Formula (FBS)	New	-0.162	-0.132
Poki-2	Formula (FBS)	Old	+0.131	+0.110
Poki-3	Simulation (SBS)	New	-0.303	-0.208

Table 5.4: Poki poker heads-up contest results summary.

possible explanations for this. It is quite possible that the set of 350 cards used for the *Poki-2* games may have been significantly biased against the human players. Another possibility is that the formula based system, written several years ago, has been tuned for the old reweighting system. If this is the case, introducing a newer system, despite being more correct, could have a negative effect on its performance. Of course, this is all conjecture – more detailed experiments should be conducted to investigate these possibilities.

The (then) state-of-the-art simulation-based bot did terribly, relative to the FBS bots. In our opinion, the variance in cards was not the cause of this result, but genuinely poor play. It was far too tight, folding too often and only betting aggressively if it had a very strong hand. These problems will be addressed in the following chapter.

5.4 Summary

This chapter evaluated the performance of several program components. Problems with reweighting were investigated and a newer method was shown to improve performance. Numerous opponent modeling techniques were evaluated. The *meta-predictor* successfully encapsulates the other methods and empirically outperforms the other methods in isolation. Neural networks are shown to be the most successful of the individual prediction methods. Finally the performance of our programs in online games is shown. Both FBS and SBS players prove to be successful against humans, however, the relative performance of FBS vs SBS is inconclusive. It was expected that the SBS would be far superior to the FBS system, but to date, this has not been demonstrated. The following chapter will discuss this in detail.

Chapter 6

Simulation Based Betting Strategies Revisited

> *There are many methods for predicting the future. For example, you can read horoscopes, tea leaves, tarot cards, or crystal balls. Collectively, these methods are known as 'nutty methods.' Or you can put well-researched facts into sophisticated computer models, more commonly referred to as 'a complete waste of time.'*
>
> – Scott Adams, 'The Dilbert Future'

There are numerous problems and difficulties with *Poki's* simulation based betting strategy. The results clearly indicate that it does not outperform the simpler formula-based betting strategy. While the simulation based strategy (SBS) does outperform the formula-based strategy (FBS) in certain situations, there are many occasions where it does far worse. The FBS is much more robust, whereas the SBS can vary widely in performance for numerous subtle and not so subtle reasons. The following section will discuss some of these problems in detail.

6.1 Problems with Simulations

Some strange things are observed when watching the SBS programs playing. One of the problems exhibited by SBS in full ten-player games is over-aggressiveness. SBS players will check-raise and re-raise with moderate strength hands. In a two-player game, the opposite problem occurs; SBS becomes extremely pessimistic and folds far too often. Even when evaluating strong hands, the EV's returned from the simulations will often be negative. Situations where bluffing is clearly profitable are not discovered, despite the fact that they should be. There are several reasons for this behavior.

6.1.1 Negative Reinforcement Through Self-Prediction

A major problem in simulations is caused by choosing our own future actions within a simulation. This can result in driving the simulations towards a self-fulfilling prophecy of playing badly. For instance, if we predict ourselves folding within a simulation, this decreases the simulated EV. If we predict ourselves folding too often, the simulated EVs will drop below zero (despite the fact that they often should be positive). With negative EV's *Poki* will fold. Since *Poki* then uses this action to update its own model of itself, this increases the chance that it will predict itself folding in future simulations. This drives the simulations into a downward spiral of predicting future folds, which in turn causes the probability of actually folding to increase.

In a ten-player game, the SBS is too aggressive. It again, falls into a self-fulfilling prophecy of raising in the future, and the opponents calling, more often than reflects reality. With all of the extra bets going into the simulated pot, the pot grows so large that even a moderate-strength hand has the pot-odds to call or raise. The EV's come back unnaturally high in this case.

6.1.2 Biased Weight Tables

Another problem is that the frequency of the simulated actions do not reflect the actual frequencies observed. Hands are chosen from the opponent's weight table which reflects hands that are more consistent with their play – but the weight table does not actually reflect the true probability distribution of cards an opponent holds (ideally it would, but our current approach to reweighting is far away from being this accurate). Instead, the weight table is biased to favour the stronger set of hands that match the board cards. Hands chosen from the weight table distribution are biased towards better hands. Since the simulated actions of opponents are predicted based on the strength of their hand, the actions themselves will be biased to reflect these better cards.

Take for instance, the following situation. We are heads-up and we are first to act on the river. The pot has 6 big bets in it and we have missed our draw. Our model tells us the opponent will fold 20% of the time to a bet on the river. That means that if we bet, we will immediately win 6 big bets 20% of the time, for $(0.2 \times 6) = 1.2$ big bets. If we are called (and we lose), or we are raised (we will fold our hand to a raise) we will lose 1 big bet. This will happen 80% of the time, so we will lose $(0.8 \times 1) = 0.8$ big bets. This gives a net EV of $(1.2 - 0.8) = +0.4$ big bets for betting on the river. Despite having a weak hand, it is profitable to bet.

In a simulation, we may bias the cards given to the opponent enough that we simulate them folding only 10% of the time, instead of 20% of the time. We fail to give them weak enough cards to fold 20% of the time. This changes the odds we are getting from the pot to -0.3 big bets for betting, which is clearly incorrect. If we had a moderately strong hand, despite missing the draw (for instance, a small pair), the biased opponent hands would also win the showdown more often than they should. This all compounds to cause simulated EV's to be far too low.

6.1.3 Patches to Fix Simulations

There are numerous other ways that small biases in the simulation can cause the EVs generated to be problematic. The problems all compound to make the current SBS a very volatile system. There are cases where it performs well, but far more where it swings wildly in the wrong direction. There are many kludgey ways to patch the specific problems. Allowing ourselves to fold or raise too often in the simulation can be artificially dampened. Noise can be added to the weight tables to reduce the bias towards strong hands. However, with these patches, one does not know the right amount of noise to add to the simulations in any given situation. Furthermore, noise merely dampens the bias, but does not correct it. An ideal solution would naturally eliminate bias in order to create a more robust simulation.

6.2 A New Approach – 'Miximax' Search

A new method of game-tree search w/ selective sampling was developed to explore more robust methods of computing the expected values of our actions. For simplicity, only the two player game will be discussed.

This new method performs a full search to the leaf nodes of the imperfect-information game tree. At the leaf nodes where show-downs occur, cards for the opponent are chosen that are consistent with the path to the leaf. The EVs from the leaf nodes are propagated back to the decision point, and weighted by the estimated probability of each branch point. We call this new search *Miximax*. Like minimax search in perfect information domains, we choose the maximal action for ourselves. But instead of choosing the minimal action for the opponent (or 'min' player) we choose the weighted sum of their actions, where the weights are taken from the estimated probability distribution over their actions.

The old simulation method chooses the cards first and then generates the actions. The *miximax* method generates the actions, and then estimates the unknown cards last. This mirrors the actual viewpoint of a poker player in a poker game. The players act in turn, and only until the end does a showdown reveal the cards of the opponent. The majority of hidden-information is placed at the leaves of the search. The assumption is that the action information is more robust than the card information.

For instance, imagine we are on the river with a strong hand, and we are contemplating our next action. Our model tells us that if we check the opponent will

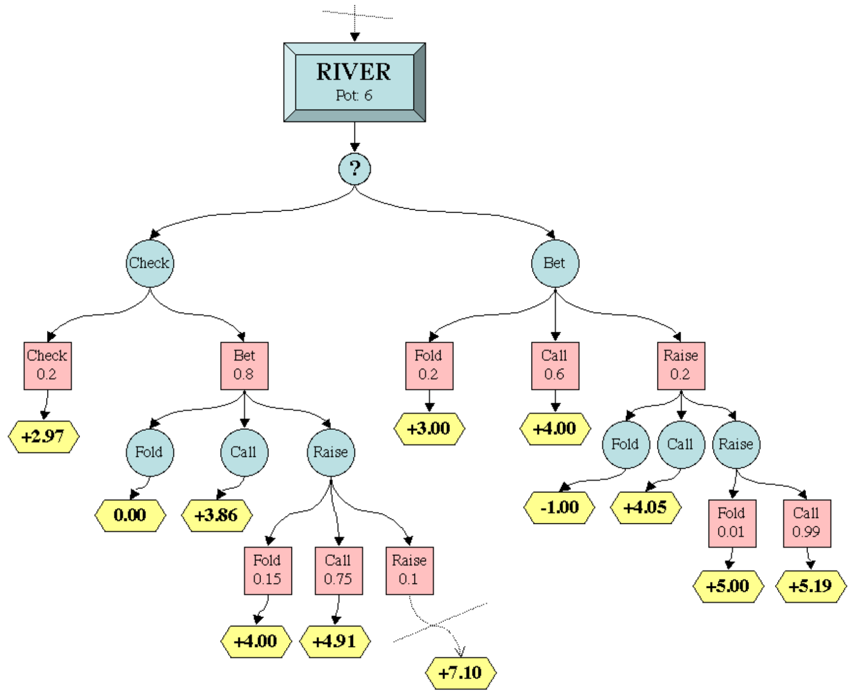

Figure 6.1: A game tree with opponent modeling and hand evaluation for a situation on the river.

check 20% of the time and bet 80% of the time. If we bet, the opponent will fold 20%, call 60%, and raise 20% of the time. We can construct the full betting tree, weighted by our knowledge of the opponent. Figure 6.1 shows the game tree from our decision point and the historical frequencies for the opponent's actions. Circular nodes show our actions, and square nodes represent the opponent's actions. Probabilities for opponent actions are given in each node. The leaves are EV's representing an estimate of the net expected value earned between our original decision point and the leaf node. Since we have a very strong hand in this example, the EV's are large positives. Note that in the betting sequences where our opponent raises, our estimate is less optimistic, since we are more likely to be up against a stronger hand in this case. If we were to recompute the EV's with a weaker hand, the EV's at the leaves would drop substantially, since the probability of winning a showdown would decrease.

By backing up the values at the leaves, we can get good estimates of the EV of the actions available to us. For instance, in Figure 6.2 we can compute that the EV of checking is equal to 0.2 times the EV of the subtree for the situation where we check, and the opponent checks, plus 0.8 times the EV for the subtree for the situation in which we check and the opponent bets.

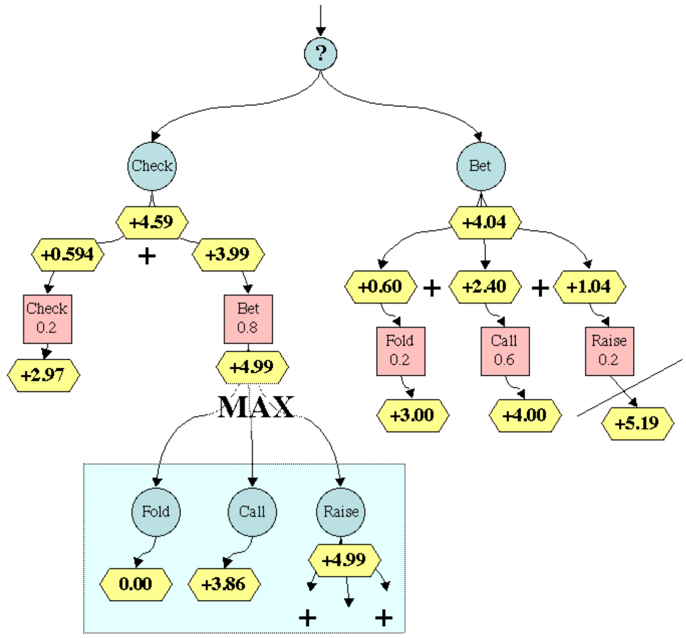

Figure 6.2: A partial calculation of the EV for checking and betting.

To summarize this process:

- To compute the EV for one of our actions we take the weighted sum of its children's EV's, where the weights are the probability of each child's occurance. Let $Pr(A_i)$ be the estimated probability that the opponent will choose child move i of node A, and let n be the number of children. The EV of our action A becomes:

$$EV(A) = \sum_{1 \leq i \leq n} Pr(A_i) \times EV(A_i) \tag{6.1}$$

- To compute the EV of an opponent's action we take the maximum EV of the children. The maximum is used for simplicity, but this can be modified to a more accurate function (*i.e.* a mixed strategy). We assume we would always choose the action with the highest EV.

$$EV(B) = \max(EV(B_{fold}), EV(B_{call}), EV(B_{raise})) \tag{6.2}$$

Or in general, for n choices:

$$EV(B) = \max(EV(B_1), EV(B_2), \ldots, EV(B_n)) \tag{6.3}$$

- To compute the EV of a chance board card (for the turn or river) the subtree below the chance card is expanded multiple times with each of the different possible cards. Since these outcomes occur uniformly at random, the average EV from the expansions is used as the final EV. Let $(\star)$ be a chance node for a board card and n be the number of possible cards left in the deck.

$$EV(\star) = \frac{\sum_{i=1}^{n} EV(\star_i)}{n} \tag{6.4}$$

- The EV at a leaf node is the probability of winning the pot times the size of the pot, minus the cost of reaching the leaf node. If the leaf node is due to a fold then the probability of winning is 1.0 if the opponent folded, and 0.0 if we folded. If the leaf is a showdown, the probability of winning is estimated as best as possible. Methods of estimating this are discussed later. Let L be a leaf node, P_{win} 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.

$$EV(L) = (P_{win} \times L_{\$pot}) - L_{\$cost} \tag{6.5}$$

6.2.1 Miximax and Miximix

Although the algorithm described in the previous section was developed as an improved method for doing poker simulations, it is in fact a fully autonomous algorithm for directly computing an estimate of the value of an imperfect game tree.

The *Miximax* strategy computes the maximal EV's of our actions. However, suppose we are presented with two positive EV's for calling and raising. Typically, one would want to choose the action with the highest EV. However, in poker, we do not necessarily always want to do this. Consistently choosing the maximum may lead to play that is too exploitable. We sometimes want to pick an action with a smaller (but non-negative) EV.

The *Miximax* algorithm predicts a probability distribution for the opponent's actions, and considers the maximum EV made at each of our decision points. The *Miximix* algorithm is more general, considering mixed strategies for both players.

Let us define a function Φ that takes the EV of each action and outputs a randomized mixed strategy over these actions (*i.e.* a probability triple). Instead of using the max function as our strategy within the search, we will instead compute the EV as the weighted sum of our child EV's. Let V be a vector containing the EV of each of our actions: $V = [EV(B_1), EV(B_2), \ldots, EV(B_n)]$

Then the EV can be computed as a dot product of the mixed strategy and the EV's:

$$EV(B) = \Phi(V) \cdot V \tag{6.6}$$

Essentially this models our own mixed strategy within the search itself. Instead of a *Miximax* search, we now have a *Miximix* search. However, defining a reasonable function Φ is still an open problem.

6.2.2 Related Work

Miximax has some high-level similarities to perfect-information game search algorithms such as **-Min* [36] and *ProbiMax* [23, 22]. Both Jansen's *ProbiMax* and Reibman and Ballard's **-Min* algorithm change the min nodes in minimax into chance nodes. Instead of returning the minimum successor of a min node, these algorithms return the weighted sum of the min node's children. In perfect-information games such as Chess, the goal of these predictive searches is to model a fallible opponent within the search – the min nodes become chance nodes that model the opponent choosing sub-optimal moves with a certain probability. This type of search is especially useful when a program is playing a losing position – if the opponent plays optimally, there is nothing that can be done. However, if the opponent is fallible they may make a mistake that can lead to a draw or even a win for the program. Using a predictive opponent model search, the program can select optimal (or even slightly sub-optimal) moves which lead to a position that increases the probability that the opponent will make a mistake.

Much of the similarity of *ProbiMax* to our method is superficial. The high level structure of the algorithm is quite similar, however the application to an imperfect information domain changes many things. The problem being solved, the 'purpose' of the search, is quite different. In Poker, minimax is not possible because we cannot compute the values for an opponent's choice without first knowing their cards. We are not modeling the objective errors of the opponent, but instead we are modeling the hidden information possessed by our opponent.

It is possible to have a good model of the probability with which an opponent will choose each action. Essentially we can model their mixed strategy within our search. Another big difference is seen in the game-tree – in poker there are also real chance nodes (for future board cards) that must be searched. Finally, at the leaf nodes of our tree we must estimate our probability of winning, whereas in perfect information games, there is an objective, correct value to assign a leaf. The estimation at the leaf nodes is loosely equivalent to an evaluation function in a classic search.

6.2.3 Estimating The Probability of Winning at Leaf Nodes

At a leaf node the probability that the pot is won must be determined. There are many ways this could be accomplished. Ideally the weighted set of possible hands that the opponent might take to the showdown, at a particular leaf node, could be used to compute our Hand Strength. In practice, getting an accurate set of weighted hands is difficult.

Ideally, for each leaf node encountered, the hands shown by the opponent could be recorded. The hand's strength and potential at each betting decision could also be computed and stored. When revisiting the leaf node in a simulation, we could then choose hands that match the same distribution of strength and potential through the hand. This, of course, is a prohibitively expensive computation on current hardware. For now, an approximation is desired.

Currently, the *Hand Rank* (HR) of the opponent's hand on the river is stored. When estimating our probability of beating the opponent, we compute our hand's hand rank as well. Recall that a hand rank is the proportion of opponent hands we can beat or tie. A hand with a lower hand rank must be beaten by a hand with a higher rank. Our probability of winning is calculated as the historical percentage of opponent hand ranks that are lower than our current hand rank.

To see, intuitively, why this works, consider the following example. Suppose we are at a leaf node with a betting sequence of `Cc/Kk/Bc/Bc`,[1] and a 0.8 HR. Historically, with this betting sequence the opponent has shown us a hand rank distribution where 90% of the hands were less than our hand rank of 0.8. Thus we can assume that with a 0.8 HR, we have a 90% probability of winning the pot at this leaf. However, when examining the leaf node prefixed by `Cc/KbC/BrC/BrRrC`, only 20% of the hands the opponent has shown have a HR less than 0.8. In the latter sequence the opponent is betting and raising – representing a much stronger hand than in the first sequence. In this case we only estimate our chance of winning the pot with a 0.8 hand rank at 20%. Given enough observations, this turns out be be a decent measure of our actual chances.

6.2.4 Pruning Large Trees

The game tree for Texas Hold'em, particularly during the pre-flop and flop, can be too large to search in real-time. There are several ways to cut down the amount of search needed. The primary explosion in tree size is due to the number of cards that must be dealt on the turn and river. It is possible to get a reasonable estimate of EV without expanding every single unseen card. As long as a smaller, representative sample of turn and river cards are expanded, the EVs should generally be close to that which is generated from the full expansion.

A second consideration is to prune away highly improbable branches of the tree that have a negligible effect on the overall EV of the parent. For instance, if the

[1] This string represents a betting sequence. Capital letters are our actions, while lowercase are the opponent's actions. '/' = the start of a new betting round. 'K' = check, 'C' = call, 'B' = bet, and 'R' = raise.

compounded probability of reaching a leaf-node is so small that its contribution to the final EV is miniscule compared to other leaf nodes, then it can be pruned without fear of radically changing the EVs. There are many common sequences in poker, and many sequences that are extremely rare (such as call-raising, or raise-folding) and will not significantly effect the outcome.

This pruning method will be necessary when using this method on games of more than two players. With added players, the size of the game tree becomes exceptionally large. In this case, it will be intractable to search the full tree in real time.

6.2.5 Current Implementation w/ Context-Trees

This new method is only in the beginning stages of research. Only a preliminary implementation has been done to verify its validity. Currently, the implementation can only play heads-up games. The search is also tied to a pure form of opponent modeling that is extremely slow at learning.

All of the previous different prediction methods attempt to consolidate and generalize from correlations in the data. An alternative is to store and use the data in its complete granularity, without any generalization. Every observed instance is stored with the complete betting sequence as the context of the action. A *prefix-tree* or *trie* can be used to store and group the observed betting sequences. This is a tree where the nodes are labelled by characters and a path from the root to a leaf is concatenated to form a string. If we find our current state in this prefix-tree (or as we call it in this case, a *Context-Tree*) the frequency of the opponent's actions (in the past) can be measured exactly. All public context information is implicitly represented in the Context-Tree.

Currently, no generalizations are made for each observed instance. One observed hand will add only one new instance to the tree. Given sufficient observations, the tree becomes a very accurate model – assuming the opponent plays a fixed strategy. Unfortunately it can easily take 20000 hands to sufficiently fill the tree.

To improve the learning speed of a Context-Tree, we need to generalize observed hands to other, similar paths through the tree. For instance, an observation of `Bc/Bc/BrC/KbC` could also be placed into a nearly identical path of `Ck/Bc/BrC/KbC`, where only the pre-flop actions have changed slightly. However, this problem has been left for future work.

The reason for using a Context-Tree at this stage of development is to have as near a 'perfect' model as possible. Generalizing data is an error prone process. Given an error-free model (*i.e.* the model makes predictions based only on perfect matches from past observations) allows us to test the Miximax search under ideal conditions.

6.2.6 Results

Our miximax-based betting strategy was tested against three different programs – FBS-Poki, SBS-Poki, and a recently developed heads-up player called *ArtBot* which was written by Arthur Choung.

ArtBot is a very loose and fairly passive heads-up player. This style of play makes it difficult to read hands. It also plays quite defensively, which makes it difficult to win large pots. Against FBS-Poki (who is currently a fairly weak heads-up player), ArtBot does very well – winning at a rate of around +0.35 sb/hand.

There is a wide difference in playing styles Between FBS-Poki, SBS-Poki, and ArtBot – a good thing to have when testing a new strategy. The programs were pitted against each other for at least 15000 hands. The Miximax player was either started with no tree at all, or a default tree built from playing a different player.[2]

Technically, the Miximax player tested was actually a Miximix player – but the ad hoc mixing function used was very close to a max function. If the EV's are positive and close (within one small bet of each other), the maximum EV action is chosen 70% of the time, the other 30%. If the two EV's differed more than this the maximum is chosen 95% of the time, and the lesser action 5% of the time. This was done to encourage some exploration – if the maximum action was always chosen, the other alternatives might never be tested.

Miximax's results against these three opponents are shown in Figure 6.3. Typically during the first 5000-10000 hands, Miximax doesn't do to well as it is still lacking a sufficient model of the opponent. However, once the model is sufficient, Miximax is very successful. After learning, the winning rates against FBS and SBS strategies are as high as +0.4 to +0.5 sb/hand. Against ArtBot, Miximax also wins at a rate that fluctuates between +0.1 and +0.2 sb/hand. ArtBot changes its style of play fast enough that the weaknesses of Context-Trees are exposed.

6.2.7 Future Work

As mentioned earlier, this is a recent idea and only preliminary research has been done. There are many ideas for future development. First, the modeling must be made to learn faster. Either the Context-Tree idea must be improved to learn faster, or the past opponent models must be used instead. Another important test would be to extend the Miximax method to handle multi-player games. The size of the game tree grows exponentially with added players. Scalability of the Miximax search might be hard to acheive without substantial pruning. Finally, the method must be tested on more than just other programmed strategies. The acid test will be measuring its performance against strong human players.

[2]The FBS-Poki match was started with an empty tree. The other two matches were started with a tree built from games against FBS-Poki.

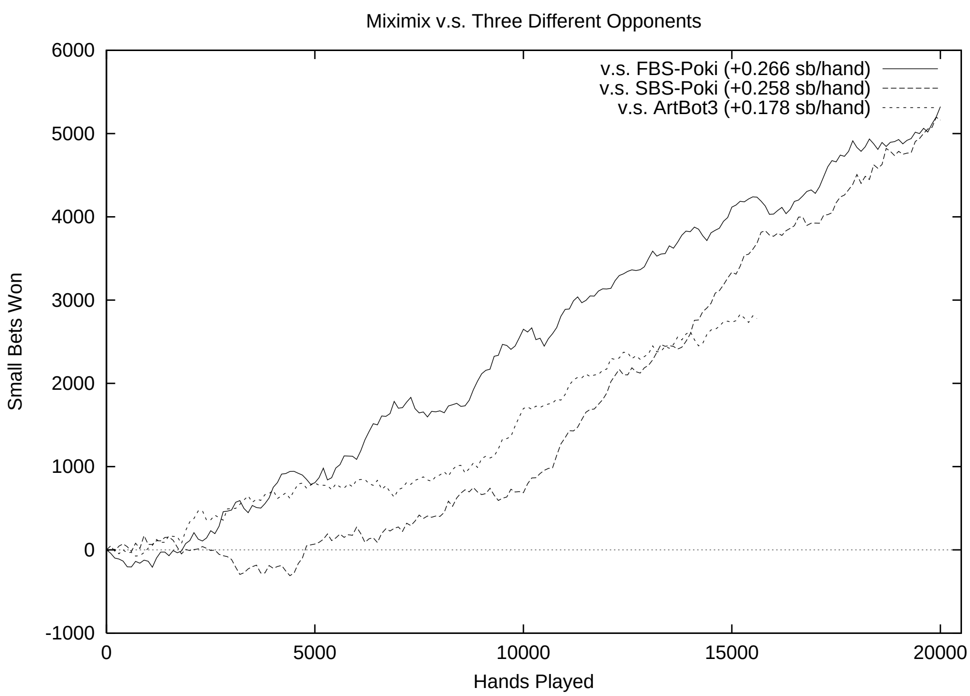

Figure 6.3: Miximax v.s. three different opponents.

Chapter 7

Conclusions and Future Work

Life consists not in holding good cards, but in playing well those you do hold.

– Josh Billings (1818-1885)

The game of poker has been a challenging domain to apply artificial intelligence techniques. It contains a variety of difficult problems in machine learning, search, simulation, and game theory. Solving these problems is a difficult but worthwhile endeavor. The real-world is full of situations that contain both non-determinism and imperfect-information properties. This research may provide a better understanding of how to model an uncertain environment, and how to use that model to make effective decisions.

The *Poki* project has developed over the last five years from its beginning as a C++ program called *Loki* into its current incarnation as a large and complex Java program. *Poki* has shown incremental improvements in performance over past versions of *Loki*. It is a winning player in online poker games against human players of average strength.

7.1 Thesis Contributions

This thesis has contributed many new enhancements to the *Poki* system. A great deal of analysis has been done to discover flaws and bugs within the older programs. As well, numerous opponent modeling enhancements were made. To summarize the major accomplishments:

- *Poki* is a complete object-oriented rewrite of *Loki*.

- *Loki* had a very simplistic 12-bin action frequency collection system for opponent modeling. This was shown to have low accuracy [13].

- The task of opponent modeling was recast as a machine learning classification/prediction problem.

- Neural networks were introduced to improve the opponent modeling abilities of *Poki*. The neural networks were successful at finding useful patterns in the noisy poker data.

- A *Meta-Prediction* system was added to encapsulate any number of prediction methods.

- A way to graphically plot the weight tables improved our ability to visualize the consequences of different reweighting methods.

- The old method of reweighting was shown to be based on the faulty assumption that EHS is uniformly distributed. A new method was designed to address this problem, and was shown to improve performance.

- Analysis of the simulation methods showed that the expected values (EV's) generated are unreliable for numerous reasons.

- *Miximax* and *Miximix*, a new, more robust method of performing a search on an imperfect-information game-tree was introduced and evaluated.

7.2 Future Work

Although *Poki* is consistently a winning player in online games against humans and other programs, *Poki* is still far from being a world-class player. There are numerous possibilities for future work:

- The opponent modeling is an ongoing challenge. There is likely to be much room for improvement. One of the biggest problems is that current machine learning methods of modeling are slow to learn. None of the methods are capable of intuitive generalization after only a few observations.

- A good understanding of game-theoretic optimal mixed strategies is needed. With knowledge of the proper optimal or near-optimal betting frequencies, *Poki's* strength could grow by leaps and bounds. Currently, our randomization is extremely *ad hoc*.

- Research into the power of *Miximax* and *Miximix* search is only just beginning. Much more research is needed in this area.

- While weight tables have been a successful technique, the implementation needs to be rethought. There are currently many flaws and biases in our weight tables – a much more sophisticated implementation is needed.

- Context-Trees are an interesting new method of modeling that stores opponent model data in a pure unadulterated form. Generalization from the data is done at run-time.

God does not play dice with the universe; He plays an ineffable game of his own devising, which might be compared, from the perspective of any of the other players [i.e., everybody], to being involved in an obscure and complex version of poker in a pitch-dark room, with blank cards, for infinite stakes, with a Dealer who won't tell you the rules, and who smiles all the time.

– Terry Pratchett and Neil Gaiman, *Good Omens*, 1991

Appendix A

Miscellaneous

A.1 Hardware for Experiments

All reported experiments were conducted on a computer with dual 933MHz Intel processors, and 512mb of RAM. The operating system used was RedHat Linux 6.2. Most of Poki was implemented in Java, and was tested under the JDK 1.4.0 using Sun's JVM.

A.2 Internet Links

The following links are of sites directly related to this thesis.

The Author's Home Page:
`http://www.cs.ualberta.ca/~davidson/`

The University of Alberta GAMES Group:
`http://www.cs.ualberta.ca/~games/`

The Poker Research Group:
`http://www.cs.ualberta.ca/~games/poker/`

A protocol for online poker games:
`http://www.cs.ualberta.ca/~davidson/poker/opp.html`

Source Code for Poki:
`http://www.cs.ualberta.ca/~davidson/poker/src/`

Appendix B

Backpropagation Example

The following is a supplementary explanation of backpropagation:

Suppose we have the neural network shown in Figure B.1. We are testing the network on the input vector $[1.0, 0.0, 0.77, 0.0]$. In this feed-forward network, evaluation is fairly simple. An input value is weighted by the strength of each connection – so the first input value of 1.0 is connected to the first internal node (left) by a weight of 0.2 (for example), and is connected to the second input neuron by a weight of -0.25. Hence, the value the first internal node sees of the first input feature is $1.0 \times 0.2 = 0.2$.

A neuron's output is computed by passing the sum of its inputs through a threshold function – in the case of our output neuron a sigmoid function. Essentially, if the sum of the inputs is above a threshold – the neuron fires. The neuron now acts as a feature detector and can excite or inhibit other neurons below it, eventually effecting the final output value of the network.

Backpropagation is a way of training the network on examples by assessing the blame of each weight in the network. It starts at the output node(s) and works back towards the inputs.

Suppose, in our example, the correct output was 1.0, but our network's actual output was 0.6. It has made an error of $1.0 - 0.6 = +0.4$ (the sign tells us which direction the error was in. To determine a share of the error for the output node we compute the derivative of the threshold (or activation) function gx on the input. If our activation function is sigmoidal: $g(x) = \frac{1}{1+e^{-x}}$ and the derivative $g'(x) = g(x) \times (1 - g(x))$.

If the input was 0.8, then the node's share of the error is $+0.4 \times [g(0.8) \times (1 - g(0.8))]$ (Equation 4.1).

Now that we know the portion of blame to assign the output node, we can adjust the weights leading into that node (Equation 4.2). A connection weight is adjusted by the blame at the node multiplied by the connection's input, multiplied by α, the learning rate which controls the magnitude of the adjustment.

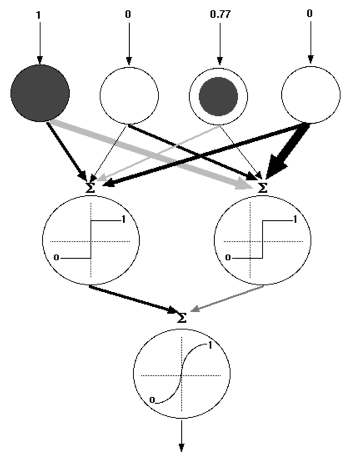

Figure B.1: A simple neural network.

There is a slight difference in computing the blame for nodes further up in the network (Equation 4.3). The activation derivative is multiplied by the sum of all blames for the output connections of that node. That is to say, the blame at output nodes is dependent on the blame at internal nodes.

During backpropagation, each weight's blame is computed and then adjusted accordingly. If a weight is to blame for the error, that weight will be adjusted so that it will contribute slightly to making the error closer to zero.

Appendix C

Sample Confusion Matrices

The following tables are sample confusion matrices for specific opponents for each of the specific modeling methods. The variance from player to player is enormous. Note these are not averages, but single data points intended for supplementary information.

Prediction

Actual	fold	call	raise	%
fold	**12.8**	8.3	0.0	21.0%
call	14.5	**43.8**	0.0	58.3%
raise	4.6	16.1	**0.0**	20.7%
%	31.9%	68.1%	0.0%	**56.51%**

Table C.1: Statistics-Loki2 (Human Opponent)

Prediction

Actual	fold	call	raise	%
fold	**12.5**	8.6	0.0	21.0%
call	12.5	**44.6**	1.1	58.3%
raise	2.9	13.8	**4.0**	20.7%
%	27.9%	67.0%	5.2%	**61.14%**

Table C.2: Statistics-II (Human Opponent)

Prediction

Actual	fold	call	raise	%
fold	**17.6**	3.4	0.1	21.0%
call	5.2	**19.5**	33.5	58.3%
raise	0.4	2.8	**17.5**	20.7%
%	23.2%	25.7%	51.1%	**54.5%**

Table C.3: Expert Formula (Human Opponent)

Prediction

Actual	fold	call	raise	%
fold	**15.7**	5.3	0.0	21.0%
call	3.8	**54.5**	0.0	58.3%
raise	0.1	20.6	**0.0**	20.7%
%	19.6%	80.4%	0.0%	**70.22%**

Table C.4: Neural Network (Human Opponent)

Prediction

Actual	fold	call	raise	%
fold	**14.8**	6.2	0.0	21.0%
call	3.8	**54.1**	0.4	58.3%
raise	0.2	18.6	**1.9**	20.7%
%	18.8%	78.9%	2.4%	**70.83%**

Table C.5: Meta-Predictor (Human Opponent)

Actual \ Prediction	fold	call	raise	%
fold	**2.6**	11.3	0.0	13.8%
call	1.8	**26.8**	8.8	37.3%
raise	0.3	10.5	**38.2**	48.9%
%	4.6%	48.4%	47.0%	**67.5%**

Table C.6: Statistics-Loki2 (FBS-Poki)

Actual \ Prediction	fold	call	raise	%
fold	**3.8**	9.8	0.3	13.8%
call	1.5	**26.8**	9.0	37.3%
raise	0.9	10.1	**38.0**	48.9%
%	6.1%	46.7%	47.3%	**68.55%**

Table C.7: Statistics-II (FBS-Poki)

Actual \ Prediction	fold	call	raise	%
fold	**8.6**	4.9	0.4	13.8%
call	2.0	**17.7**	17.6	37.3%
raise	0.3	1.9	**46.8**	48.9%
%	10.9%	24.4%	64.8%	**73.1%**

Table C.8: Expert Formula (FBS-Poki)

Actual \ Prediction	fold	call	raise	%
fold	**7.4**	2.3	4.2	13.8%
call	1.6	**13.0**	22.8	37.3%
raise	0.3	13.0	**35.7**	48.9%
%	9.2%	28.2%	62.6%	**56.0%**

Table C.9: Neural Network (FBS-Poki)

Actual \ Prediction	fold	call	raise	%
fold	**8.6**	5.0	0.2	13.8%
call	2.0	**22.3**	13.1	37.3%
raise	0.3	4.7	**43.9**	48.9%
%	11.0%	31.9%	57.2%	**74.8%**

Table C.10: Meta-Predictor (FBS-Poki)

Appendix D

Glossary of Poker Terms

This appendix contains definitions of common poker terms used in this thesis.[1] More extensive poker glossaries are available on the world wide web, such as `http://www.kimberg.com/poker/dictionary.html`, or `http://conjelco.com/pokglossary.html`.

- **Aggressive Player** A player who is characterised by frequently betting and raising.
- **All-in** To have one's entire stake committed to the current pot. Action continues toward a **side pot**, with the all-in player being eligible to win only the main pot.
- **All-in Equity** The expected income if the current hand was permitted to go to the **showdown** with no further betting.
- **Ante** A mandatory fee placed into the pot from each player at the start of a game (in some types of Poker).
- **Bet** To make the first wager of a betting round (compare **raise**).
- **Big Bet** The largest bet size in limit poker. $20 in $10-$20 Hold'em.
- **Big Blind** A forced **bet** made before the deal of the cards. $10 in $10-$20 Hold'em, posted by the second player to the left of the **button**.
- **Blind** A forced **bet** made before the deal of the cards

  (see **small blind** and **big blind**).
- **Bluff** To **bet** with the expectation of losing if called.
- **Board** The **community cards** shared by all players.

[1] Adapted from [6].

- **Button** The last player to act in each betting round in Texas Hold'em.
- **Call** To match the current level of betting. If the current level of betting is zero, the term **check** is preferred.
- **Check** To decline to make the first wager of a betting round (compare **call**).
- **Check-Raise** To **check** on the first action and then **raise** in the same betting round after someone else has **bet**.
- **Community Cards** The public cards shared by all players.
- **Connectors** Two cards differing by one in rank, such as 7-6. More likely to make a straight than other combinations.
- **Draw** A hand with good potential to make a strong hand, such as a *straight draw* or a *flush draw* (compare **made hand**).
- **Dry** Lacking possible draws or betting action, as in a *dry board* or a *dry game*.
- **Flop** The first three **community cards** dealt in Hold'em, followed by the second betting round (compare **board**).
- **Fold** To discard a hand instead of matching the outstanding **bet**, thereby losing any chance of winning the pot.
- **Free-Card Raise** To **raise** on the **flop** intending to **check** on the **turn**.
- **Hand** (a) A players private cards (two **hole cards** in Hold'em). (b) One complete game, from the dealing of the cards to the **showdown** (or until one player wins uncontested).
- **Hole Card** A private card in Hold'em.
- **Implied Odds** The **pot odds** based on the probable future size of the pot instead of the current size of the pot.
- **Income rate** The expected amount a hand will win.
- **Kicker** A side card, often deciding the winner when two hands are otherwise tied (*eg.* a player holding Q-J when the **board** is Q-7-4 has **top pair** with a Jack kicker).
- **Loose Game** A game having several **loose players**.
- **Loose Player** A player who does not fold often (*eg.* one who plays most hands at least to the **flop** in Hold'em).

- **Made hand** A hand with a good chance of currently being the best, such as **top pair** on the **flop** in Hold'em (compare **draw**).
- **Mixed strategy** Handling a particular situation in more than one way, such as to sometimes **call**, and sometimes **raise**.
- **Near-optimal** A good approximation of a game-theoretic optimal solution.
- **Offsuit** Two cards of different suits (compare **suited**).
- **Open-Ended Draw** A **draw** to a straight with eight cards to make the straight, such as 6-5 with a **board** of Q-7-4 in Hold'em.
- **Outs** Cards that will improve a hand to a probable winner (compare **draw**).
- **Passive Player** A player who is characterised by frequently checking and calling his hand rather than betting or raising.
- **Pocket Pair** Two cards of the same rank, such as 6-6. More likely to make three of a kind than other combinations (see **set**).
- **Pot Odds** The ratio of the size of the pot to the size of the outstanding bet, used to determine if a **draw** will have a positive expected value.
- **Pre-flop** In Hold'em, the first betting round after the deal of the cards and before the **flop**.
- **Raise** To increase the current level of betting. If the current level of betting is zero, the term **bet** is preferred.
- **River** The fifth **community card** dealt in Hold'em, followed by the fourth (and final) betting round.
- **Second pair** Matching the second highest **community card** in Hold'em, such as having 7-6 with a **board** of Q-7-4.
- **Set** Three of a kind, formed with a **pocket pair** and one card of matching rank on the **Board**. A powerful well-disguised hand (compare **trips**).
- **Showdown** The revealing of cards at the end of a **hand** to determine the winner and award the pot.
- **Side-pot** A second pot for the bets made by active players after another player is **all-in**.
- **Slow-play** To **call** with a strong hand, and then **raise** in a later betting round, for purposes of deception.

- **Small Bet** The smallest bet size in limit poker. $10 in $10-$20 Hold'em.
- **Small Blind** A forced **bet** made before the deal of the cards. $5 in $10-$20 Hold'em, posted by the first player to the left of the **button**.
- **Suited** Two cards of the same suit, such as both Hearts. More likely to make a flush than other combinations (compare **offsuit**).
- **Suited-Connector** A hand that is both **Suited** and a **Connector** (i.e. Q♡-J♡).
- **Table Image** The general perception other players have of one's play.
- **Table Stakes** A poker rule allowing a player who cannot match the outstanding bet to go **all-in** with his remaining money, and procede to the **showdown** (also see **side pot**).
- **Texture of the Board** Classification of the type of **board**, such as having lots of high cards, or not having many draws (see **dry**).
- **Tight Player** A player who usually folds unless the situation is clearly profitable (*eg.* one who folds most hands before the **flop** in Hold'em).
- **Top pair** Matching the highest **community card** in Hold'em, such as having Q-J with a **board** of Q-7-4.
- **Trips** Three of a kind, formed with one **hole card** and two cards of matching rank on the **board**. A strong hand, but not well-disguised (compare **set**).
- **Turn** The fourth **community card** dealt in Hold'em, followed by the third betting round.
- **Under-the-gun** The position of the player who is first to act in the betting round.
- **Wild Game** A game with a lot of raising and re-raising. Also called an **action game**.

Bibliography

[1] R. Axelrod. *The Evolution of Cooperation.* Basic Books, Inc., 1984.

[2] H. Berliner, G. Goetsch, and M. Campbell. Measuring the performance potential of chess programs. *Artificial Intelligence*, 43(1):7–20, 1990.

[3] D. Billings. Computer poker. Master's thesis, University of Alberta, 1995.

[4] D. Billings. The first international RoShamBo programming competition. *International Computer Games Association Journal*, 23(1):42–50, 2000.

[5] D. Billings. Thoughts on RoShamBo. *International Computer Games Association Journal*, 23(1):3–8, 2000.

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

[7] D. Billings, D. Papp, L. Pena, J. Schaeffer, and D. Szafron. Using selective-sampling simulations in poker. In *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, pages 13–18, 1999.

[8] D. Billings, L. Pena, J. Schaeffer, and D. Szafron. Using probabilistic knowledge and simulation to play poker. In *AAAI National Conference*, pages 697–703, 1999.

[9] M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *International Computer Chess Association Journal*, 20(3):189–193, 1997.

[10] M. Campbell, A. J. Hoane, and F h. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[11] D. Carmel and S. Markovitch. Incorporating opponent models into adversary search. In *AAAI National Conference*, pages 120–125, 1995.

[12] F.A. Dahl. A reinforcement learning algorithm applied to simplified two-player Texas Hold'em poker. In *12th European Conference on Machine Learning (ECML'01)*, pages 85–96, 2001.

[13] A. Davidson, D. Billings, J. Schaeffer, and D. Szafron. Improved opponent modeling in poker. In *International Conference on Artificial Intelligence (IC-AI'2000)*, pages 1467–1473, 2000.

[14] H. Dreyfus. *What Computers Still Can't Do: A Critique of Artificial Reason.* MIT Press, 1993.

[15] D. Egnor. Iocaine powder. *International Computer Games Association Journal*, 23(1):33–35, 2000.

[16] S. G. Ficici and J. B. Pollack. A game-theoretic approach to the simple coevolutionary algorithm. In *Parallel Problem Solving from Nature - PPSN VI 6th International Conference*, Paris, France, 16-20 2000. Springer Verlag.

[17] S G. Ficici and J. B. Pollack. Pareto optimality in coevolutionary learning. In *European Conference on Artificial Life*, pages 316–325, 2001.

[18] N. Findler. Studies in machine cognition using the game of poker. *Communications of the ACM*, 20(4):230–245, 1977.

[19] R. Fung and K. Chang. Weighting and integrating evidence for stochastic simulation in bayesian networks. In *Uncertainty in Artificial Intelligence*, pages 209–220. Morgan Kaufmann, 1989.

[20] M. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *International Joint Conference on Artificial Intelligence*, pages 584–589, 1999.

[21] H. Iida, J. Uiterwijk, J. van den Herik, and I. Herschberg. *Thoughts on the Application of Opponent-Model Search*, pages 61–78. Univ. of Limburg, Maastricht, 1994.

[22] P. Jansen. *Using Knowledge About the Opponent in Game-Tree Search*. PhD thesis, Carnegie-Mellon University, 1992.

[23] P. Jansen. KQKR: Speculatively thwarting a human opponent. *International Computer Chess Association Journal*, 16(1):3–17, 1993.

[24] D. Koller, N. Megiddo, and B. von Stengel. Fast algorithms for finding randomized strategies in game trees. In *26th Annual ACM Symposium on the Theory of Computing*, pages 750–759, 1994.

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

[26] K. Korb and A. Nicholson. Bayesian poker. In *Uncertainty in Artificial Intelligence*, pages 343–350, 1999.

[27] H. W. Kuhn. A simplified two-person poker. *Contributions to the Theory of Games*, 1:97–103, 1950.

[28] T. A. Marsland and J. Schaeffer. *Computers, Chess, and Cognition*. Springer-Verlag, 1990.

[29] M. L. Minsky. *Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem*. PhD thesis, Princeton University, 1954.

[30] M. L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, Mass., 1969.

[31] J. F. Nash. Non-cooperative games. *Annals of Mathematics*, 54:286–295, 1951.

[32] J. F. Nash and L. S. Shapley. A simple three-person poker game. *Contributions to the Theory of Games*, 1:105–116, 1950.

[33] J. Noble and R. A. Watson. Pareto coevolution: Using performance against coevolved opponents in a game as dimensions for pareto selection. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 493–500, 2001.

[34] D. Papp. Dealing with imperfect information in poker. Master's thesis, University of Alberta, 1998.

[35] L. Pena. Probabilities and simulations in poker. Master's thesis, University of Alberta, 1999.

[36] A. L. Reibman and B. W. Ballard. Non-minimax search strategies for use against fallible opponents. In *AAAI National Conference*, pages 338–342, 1983.

[37] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back–propagating errors. *Nature*, 323:533–536, 1986.

[38] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[39] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:211–229, 1959.

[40] J. Schaeffer, J. Culberson, N Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–290, 1992.

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

[42] R. Shacter and M. Peot. Simulation approaches to general probabilistic inference on belief networks. In *Uncertainty in Artificial Intelligence*, pages 311–318. Morgan Kaufmann, 1989.

[43] B. Sheppard. Computer Scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

[44] J. Shi and M. Littman. Abstraction models for game theoretic poker. In *Computer Games'00*, pages 333–345. Springer-Verlag, 2001.

[45] D. Sklansky and M. Malmuth. *Hold'em Poker for Advanced Players.* Two Plus Two Publishing, 2nd edition, 1994.

[46] S. Smith. Flexible learning of problem solving heuristics through adaptive search. In *IJCAI*, pages 422–425, 1983.

[47] K. Takusagawa. *Nash Equilibrium of Texas Hold'em Poker.* Undergraduate thesis, Stanford University, 2000.

[48] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[49] G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

[50] A. M. Turing, C. Strachey, M. A. Bates, and B. V. Bowden. Digital computers applied to games. In B. V. Bowden, editor, *Faster Than Thought*, pages 286–310. Pitman, London, 1953.

[51] P. E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161–186, 1989.

[52] J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior.* Princeton University Press, 2nd edition, 1947.

[53] D. Waterman. A generalization learning technique for automating the learning of heuristics. *Artificial Intelligence*, 1:121–170, 1970.