EVOLUTIONARY METHODS FOR LEARNING NO-LIMIT TEXAS HOLD'EM POKER

A Thesis

Submitted to the Faculty of Graduate Studies and Research

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

University of Regina

By

Garrett Joseph Nicolai

Regina, Saskatchewan

November 2008

Abstract

No-Limit Texas Hold'em is a stochastic game of imperfect information. Cards are dealt randomly, and players try to hide which cards they are holding from their opponents. Randomness and imperfect information give Poker, in general, and No-Limit Texas Hold'em, in particular, a very large decision space when it comes to making betting decisions. Evolutionary algorithms and artificial neural networks have been shown to be able to find solutions in large and non-linear decision spaces, respectively. A hybrid method known as evolving neural networks is used to allow No-Limit Texas Hold'em Poker playing agents to make betting decisions. The evolutionary heuristics of halls of fame and co-evolution extend the evolving neural networks to counter evolutionary forgetting and improve the quality of the evolved agents. The appropriateness of a tournament-based fitness function for the evolutionary algorithms is investigated. The results show that the tournament-based fitness function is the most appropriate of the tested fitness functions. Furthermore, the results show that the use of co-evolution and a hall of fame increases the quality of the evolved agents against a number of benchmark agents. The use of co-evolution and a hall of fame separately show that the hall of fame has the greater influence on the evolution of the agents, and the addition of co-evolution to the hall of fame has little added benefit.

Acknowledgments

I would like to thank my supervisor, Dr. Robert Hilderman, without whose encouragement, advice, and support, both moral and financial, this thesis would not have been possible. I would also like to thank the Faculty of Graduate Studies and Research for their financial support. Last, but certainly not least, I need to thank my biggest support group and cheering section, my parents, George and Karen Nicolai, and my siblings, Ed, Dan, and Tori. They may not have always understood what I was doing, but I'll always understand the role they've played in my success.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Statement of the problem

In the field of Artificial Intelligence, games have provided a useful area for research due to a well-defined set of rules and a distinguishable goal state. Game-playing agents have been developed for numerous games such as Chess, Checkers, Backgammon, Go, and Poker. Although the rules of any particular game remain constant, individual games are vastly different, requiring different playing styles and strategies.

Games provide opportunities for evaluation of agents, as agents developed to play a particular game can play other agents of the same game to objectively determine their relative abilities. Game-playing agents touch upon many different areas of Artificial Intelligence, for example:

- Rule-based classification. Many games take as input various variables related to the current state of the game and output the best decision based upon these variables. One example might be a Poker game that takes as input the value of the cards and the cost of a bet, and outputs a decision to fold, call, or raise.

- State-machines. Other games such as Backgammon or Chess take as input the current state of the board, and given the rules of the game, determine the best next-state of the game.

- Machine learning. Just as human players begin playing games as novices, with low skill levels, so too do computer solutions often begin in a less-desirable state. Some agents have playing styles that are adjusted after finishing a game, while other try to improve while the game is being played. Only through many iterations of improvement do the agents become capable of playing a moderate, and eventually an expert-level game.

- Risk minimization. Sometimes, agents must not only maximize their potential of winning, but try to minimize the potential of an opponent winning. For example, in the game of Scrabble, a certain word may score more points than the word that is eventually played, but if it allows the opponent to score even more points, it is not a good solution.

- Event simulation. Whether in the learning phases or in the evaluation phases, agents are run through simulations of the games that they are playing. Agents often have to play many times against each other in order to determine an agent's success, and must search, ideally, the entire state space of a game, even though many states may occur only rarely in human games. Exploration of these unique game states can lead to the development of new strategies to be used by human players.

1.2 Game Theory

In the early years of Computer Science, researchers were already interested in the applications of games, and how to solve them using a computer. John Von Neumann, one of the pioneers of both Computer Science and Game Theory [28, 30, 41] was interested in creating an artificial agent to play Poker [27]. The field of Game Theory itself, while not solely concerned with the "classic" parlour games, does define a standard method of classifying any number of situations.

In Game Theory, a game is any type of relationship between two or more participants with a well-defined goal [41]. Thus, although Chess and Checkers are games according to this definition, more abstract situations are also described as games. A war can be a game, as two or more sides are competing toward a definable goal. A debate can be a game, as can the selling of competing products, both trying to win the customer's money. However, for the sake of this thesis, all references to "game" outside of this chapter shall be strictly referring to the traditional meaning of the word, namely a competition between two or more individuals with the goal of defeating the other players. Thus, in future descriptions, Poker will be considered a game, but a war will not.

In Game Theory, there are at least four methods of classifying a game, none of which are mutually exclusive [30]. Usually, the classes are disjoint, and if a game does not fall into one of the categories, it will fall into the other. All games, whether

they have been around since antiquity, or have been created recently, can be classified according to the categories defined by Game Theory. Games with similar rules are usually classified in the same classes, although it can take only one small change of the rules to move a game to one of the alternate categories.

1.2.1 Competitive and co-operative games

The first category is perhaps the most obvious, and considers how opponents view each other within the context of the game. A game can be said to be either *competitive* or *cooperative*, and can potentially fall somewhere in the middle. The goal of a competitive game is to defeat the opponent and win whatever is at stake in the game. Games such as Chess, Checkers, and Backgammon are competitive in that the only goal of the player is to win the game. The loser may receive some consolation, such as a number of points, but one player generally does not require aid from the other in order to win. A cooperative game, on the other hand, forces players to act together to achieve a goal. Team sports are both cooperative and competitive, in that players cooperate with team-mates, but compete against the opposition. A well known example of a game in which it is more beneficial to cooperate than to compete is the iterated Prisoner's Dilemma. More points are awarded in the long run if players cooperate than if they compete.

1.2.2 Zero-sum and non-zero-sum games

Once it has been decided whether or not the players are competing or cooperating, some reward structure must be defined. A game can be classified as *zero-sum* if and only if a win is worth the same as a loss. In other words, if a player receives a score for winning, the opponent must be punished by the same score. If, after a large number of games, wins are given a positive value, and losses are given a negative value, the sum of wins to losses will add up to zero. A *non-zero-sum* game, on the other hand, has some extra influence, such that the sum of all players' scores is greater than or less than zero. Both the Prisoner's Dilemma and Monopoly are examples of non-zero sum games. The Prisoner's Dilemma is non-zero-sum because cooperating players can both win in the long run. Monopoly introduces new money each time a player passes "Go", meaning that a player's net losses will not equal the opponent's net gains.

1.2.3 Games of perfect and imperfect information

Games often vary by the amount of opponent information that is available to any one player. If all information regarding the current game state is available to all players, then a game is said to contain *perfect information*. Likewise, if any information is private to an individual player, the game is said to contain *imperfect information*. Chess, Backgammon, and Go are examples of games with perfect information. Any player can look at the board and obtain any information that is required. He knows where all of the pieces are in relation to his own, and he knows all of the moves that are available to his opponent. The player can base his strategy on this knowledge. On the other hand, games like Scrabble and Poker are games with imperfect information. In Scrabble, a player does not know which letter tiles his opponent is holding, and thus needs to be careful of providing his opponent with large scoring opportunities. Regardless of the variant of Poker being played, players hide at least some cards from their opponents. Since a player cannot know with 100% certainty what his opponent is holding, he also cannot know with 100% certainty the quality of his own current status. Imperfect information complicates decision making. This is one of the key reasons why agents that can play games of perfect information such as Checkers [37], Chess [12], and Backgammon [43] have had more success than those that play games of imperfect information.

1.2.4 Stochastic and deterministic games

Finally, games are classified according to how the game progresses from one state to the next. If a player's valid choice of moves can be listed with complete certainty before a player moves, a game is said to be deterministic. Whenever a certain state is reached, there are a small number of potential next-states available, as each piece can only be played in a limited number of ways. Chess, Checkers, and Go are examples of deterministic games. In Chess, a queen always moves the same way, as does a bishop, a knight, or a king. As a player decides his move, he can know, with 100% certainty, the exact moves that will be possible for his opponent if he makes a certain move.

Non-deterministic games are called stochastic games. Stochastic games can also have all of their moves plotted, but not immediately after a player makes a move. Some element of chance, be it dice, cards, or some other mechanism, is introduced that makes both players unsure of which moves are possible until after the random

agent has been applied. Backgammon, Scrabble, most card games, and Monopoly are examples of stochastic games. In Backgammon and Monopoly, a player rolls the dice to determine the spaces on the board that are accessible. In Scrabble, letter tiles are selected blindly, leading to a random distribution (although certain letters are more likely to appear). In card games, the deck is shuffled randomly, and cards are distributed in no particular order, allowing what is a good hand in one situation to become a poor hand in another. These elements of chance tend to increase a game's decision space, as it becomes more difficult to predict an opponent's counter-strategy, because of the unknown future state of the game.

1.3 Artificial intelligence in games

Until recently, only in games consisting of either deterministic decisions or perfect information such as Chess [12, 16, 18, 19, 20, 23, 44], Checkers [35, 37], and Backgammon [32, 42, 43] were artificial agents able to succeed against the best human players. The game of Go [26, 33, 39], despite having both deterministic decisions and perfect information, has continued to confuse the best computer agents. Only recently in games with both imperfect information and stochastic decisions, such as Scrabble [40], have agents achieved a level beyond any human players.

1.4 Texas Hold'em playing agents

As computers become more powerful, it is becoming feasible to create agents that can succeed in games with larger decision spaces. Brute-force methods are still too time consuming for games with many potential decisions, such as Backgammon, Poker, and Scrabble, but other methods of problem solving are being applied. Recently, there has been an explosion of interest in the game of Poker, in general, and in the Texas Hold'em variant in particular. In other work [1, 2, 3, 4, 5, 6, 7, 11, 13, 14, 15, 21, 29, 38], efforts have been made to improve the skill-level of computer Poker playing agents.

1.4.1 No-Limit Texas Hold'em

The majority of the research on Texas Hold'em is concentrated on a single variant known as Limit Texas Hold'em. Limit Texas Hold'em limits the amount of money

that can be bet during any round. There is also a variant of Texas Hold'em known as No-Limit Texas Hold'em, with slightly different rules, but very different gameplay. No-Limit Texas Hold'em allows a player to bet any amount whenever it is his turn to bet. This indiscriminate betting has several effects on the gameplay. Players need to be more cautious, as a single mistake can cost a player all of his money. Deception plays a larger role in the game than in Limit Texas Hold'em, as players with a lot of money can make extremely large bets solely for the purpose of scaring away opponents. It becomes harder for a player to accurately analyse his opponents.

There has been a concentrated effort over the last decade to produce skilled Limit agents; the majority of No-Limit agents that exist are adaptations of these Limit agents. Although Limit agents can play the No-Limit variant with minor adaptations, they were developed to play Limit Texas Hold'em, and may be missing some of the intricacies required to play No-Limit Texas Hold'em effectively. An agent that is developed with the goal of playing No-Limit Texas Hold'em could be designed with these intricacies in mind.

1.5 Objective of thesis

In this thesis we develop No-Limit Texas Hold'em Poker playing agents using evolving neural networks. As well as developing these agents, further agents are evolved using the evolutionary heuristics of co-evolution and halls of fame. The goal of this thesis is to develop agents that can play the game of No-Limit Texas Hold'em well, outperforming benchmark agents and previously developed No-Limit Texas Hold'em agents.

No-Limit Texas Hold'em is a game that is very difficult for computer agents to play well:

- The game is competitive, and games often consist of more than two players. Chess, Checkers, Backgammon, and Go are all two-player games. Poker often has up to ten players at a table, and many more in a tournament. Playing styles need to be robust if the agent wants to do well.

- The game is zero-sum. Some casinos take a share of the money that is bet, but for the most part, Poker is a game where gains for one player are balanced by losses for others.

- The game is stochastic, and has a very large decision space. At any decision point, agents must analyse numerous features, some of which are distributed randomly.

- The game contains imperfect information, and the agent knows very little about his opponents. Some information, such as the amount of money available to any agent and the money that is available to be won, is known to all of the agents. An agent needs to be able to determine the playing styles of his opponents, even when he does not know all of the variables of an opponent's betting decisions.

- The game encourages players to misinform their opponents. Players not only have to deal with hidden information, but the information that they are able to discern might not be accurate.

- The game has a larger range of betting than Limit Texas Hold'em. In Limit Texas Hold'em, a bet is always the same, but in No-Limit Texas Hold'em, there is a large range of possible bets. Relatively small differences in bets can lead to large differences in how an opponent plays a hand.

1.5.1 Evolving neural networks, co-evolution, and halls of fame

Evolving neural networks have been used to find solutions in games with large state spaces [26, 35, 43]. Evolving neural networks are a hybrid of artificial neural networks and evolutionary methods, both shown to be able to find non-linear solutions in large state spaces [17, 26, 32, 33, 43]. An artificial neural network performs manipulations upon a vector of input values, eventually returning a vector of output values. Betting decisions in a Poker game involve analysing many various features, and arriving at a vector of potential betting decisions. Artificial neural networks are well-suited to the task of determining betting decisions.

Evolutionary algorithms evaluate competing solutions, discard poor solutions, and make minor changes to generate new solutions. Poker players are evaluated by competing against other players, and small changes in betting decisions between players can result in large differences in skill. Evolutionary algorithms are well-suited to finding good solutions where small differences matter.

Co-evolution and halls of fame have been shown to improve the performance

of evolutionary algorithms [26, 32, 34]. Co-evolution conducts several evolutionary experiments simultaneously, and takes advantage of the shared experience between the separate experiments. A hall of fame is a structure that ensures that agents are able to defeat previously evolved agents, ensuring that the evolutionary algorithm is making progress.

1.6 Outline of thesis

This thesis is organized as follows. Chapter 2 gives an overview of the rules and terms of No-Limit Texas Hold'em. Chapter 3 presents a survey of both work related to the game of Texas Hold'em, as well as the use of evolutionary methods in other games. This survey attempts to present a progression of the state of Texas Hold'em playing agents, as well as reveal several important results obtained through the use of evolutionary methods when applied to games.

Chapter 4 presents our representation of a No-Limit Texas Hold'em Poker playing agent, and our methodology for the evolution of this agent. Many of the aspects of our agents had to be adapted to suit the neural network used in making betting decisions. Feature selection and representation are discussed in detail, and the evolutionary process is also explained. Furthermore, duplicate tables, a method of Poker agent evaluation, is discussed

No-Limit Texas Hold'em Poker playing agents were developed using evolving neural networks by themselves, using co-evolution, and with a hall of fame structure. Three fitness functions: tournament selection, HandsWon, and MeanMoney were used to evolve the agents. Experiments evaluating the skill of the evolved agents are conducted and the results are presented in Chapter 5. The skill levels of the obtained agents are evaluated and evolutionary progress is measured. Discussion follows concerning the objective skill levels of the evaluated agents, and the appropriateness of the additional heuristics is analysed.

Chapter 6 briefly re-states our results, summarises our conclusions, and suggests areas for future work.

Chapter 2

OVERVIEW OF NO-LIMIT TEXAS HOLD'EM

Texas Hold'em is a variant of the card game of Poker. It is considered a *community* variant of the game; although each player holds several cards of his own, a complete hand can only be formed using community cards that are available to all of the players. There are several different variants of Texas Hold'em, such as Limit, No-Limit, and Pot-Limit Texas Hold'em, usually only differing in the amount of money that is allowed to be bet at any one time. In this thesis, we are concerned with No-Limit Texas Hold'em, a variant where a player can bet all of his money at any time in the game, and the relevant rules and other details are introduced in this chapter.

This chapter is organised as follows. Section 2.1 outlines the rules and gameplay of the variant of Poker known as No-Limit Texas Hold'em. Section 2.2 briefly describes the need for players to misrepresent their cards. Finally, section 2.3 outlines some common ways that Poker players attempt to deceive opponents.

2.1 Texas Hold'em Poker

In Texas Hold'em Poker, games are divided into sub-games, known as *hands*, which proceed until either one player has won, or there are no players left playing. Texas Hold'em is a community variant of Poker, where players attempt to make the best ranking hand of five cards from a total of seven cards. Winning hands are shown in Table 2.1, where "T" is used as an abbreviation for 10. Two of these seven cards, called *hole cards*, are private cards that are dealt to one player and can only be seen by that player. The other five cards, known as *the community* or *the table*, are shared public cards that can be seen by all players and used by any of the players to make a hand. A player must use at least three of the community cards to make a hand, but may use all five if necessary.

Table 2.1: Winning Poker hands in descending order

Hand	Consists	Example	Tiebreaker
Straight Flush	Five cards with the same suit, with consecutive rank	A♣ K♣ Q♣ J♣ T♣	High Card
Four of a Kind	Four cards with the same rank, fifth card is of another rank	A♣ A♡ A♠ A♢ K♣	Rank of Four of a Kind; if equal, rank of fifth card
Full House	Three of a Kind of one rank, and a Pair of another	A♣ A♡ A♠ K♢ K♡	Rank of Three of a Kind; if equal, rank of pair
Flush	Five cards with the same suit, but not all with consecutive rank	A♢ K♢ Q♢ J♢ 9♢	High Card; if high cards are equal, continue with next card
Straight	Five cards of consecutive rank	A♣ K♡ Q♡ J♠ T♣	High Card
Three of a Kind	Three cards with the same rank, all other cards are of different ranks	A♡ A♢ A♠ K♠ Q♠	Rank of Three of a Kind; if equal, rank of highest other card
Two Pair	Two pairs, with fifth card of different rank	A♡ A♠ K♠ K♢ Q♣	Rank of High Pair; if of equal rank, rank of low pair; if equal, rank of fifth card
Pair	Two cards with same rank, all other cards are of different ranks	A♣ A♢ K♢ Q♠ J♢	Rank of Pair; if of equal rank, rank of highest other card
High Card	No other hand conditions satisfied	A♠ K♠ Q♠ J♠ 9♢	Rank of highest card; if equal, repeat with next highest card

Play can be best understood if we assume that the players are seated at a round table. Play begins by the *dealing*, or distribution, of each player's hole cards. One player is designated the *dealer*, and cards are dealt in a clockwise direction starting with the player on the dealer's immediate left. The dealer is also used as a reference point for betting, with the player sitting on his immediate left starting the betting in each round. After every hand, a new dealer is designated and is the player on the current dealer's immediate left.

Texas Hold'em rules dictate at least two forced bets. The player on the immediate left of the dealer is designated the *small blind*, and pays a pre-determined amount into the pot. The player on the immediate left of the small blind is designated the *big blind*, and pays a slightly larger pre-determined amount into the pot. The player on the immediate left of the the big blind begins the betting, which proceeds in a clockwise direction. If there are only two players, a situation known as *playing heads-up*, the betting structure is changed. In this case, the dealer is the small blind, and the other player is the big blind. When playing heads-up, the dealer starts the betting in the first betting round, but in all subsequent rounds, the big blind starts the betting. Occasionally, another forced bet, called an *ante*, is paid by all players at the table, in addition to the blinds. Often, in a tournament, the blinds increase as more hands are played, as shown in Table 2.2 [25].

Table 2.2: Selected values of the blinds and ante

Hands Played	Small Blind	Big Blind	Ante
0-14	10	20	0
15-29	15	30	0
30-44	25	50	0
90-104	100	200	25
150-164	600	1200	75
210-224	3000	6000	300
255-269	10000	20000	1000
360-374	150000	300000	15000

Depending on the variant being played, the "winning hand" can have different meanings. Some variants of Poker award a win for the lowest ranking hand, while others have special rules, but they are not considered here. No-Limit Texas Hold'em gives a win to the highest ranking five card hand. Eventually, regardless of the variant of Poker being played, at least two players will need to compare their hands, an event known as a *showdown*. When a showdown occurs, all players show their cards, and the player with the highest ranking hand wins whatever money is in the

pot, a collection of all bets that were previously made. Some Poker variants include a *jackpot*, to which each player contributes every hand. After some criterion is met (i.e., a certain rare hand is achieved, or an overall winner is declared), the jackpot is awarded. Texas Hold'em does not include a jackpot. In Texas Hold'em, suits have no inherent value. The only hands that consider suits are flushes. Since three community cards are required to form a flush, two simultaneous flushes of different suits cannot occur simultaneously.

2.1.1 Betting

Betting is the aspect of Poker that will separate good players from poor players. If there were no betting choices to be made, the game would completely rely upon the luck of the cards, and whichever player received the best cards would win (which would be uniformly distributed amongst all of the players over many hands).

Texas Hold'em allows three different actions when a player makes a bet: *folding*, *calling*, and *raising*. First, when a player *folds* his cards, he declines to make a bet, and removes himself from the playing field for the rest of the hand (i.e., even if cards are dealt later in the hand that improve a player's hand). Second, if another player has made a bet, a player *calls* if he wants to continue in the hand, but does not want to bet any more money. A special case of a call occurs when there is no bet that must be called. A call of cost zero is called a *check*. Last, if there is a bet to call, but a player decides that he wants more money in the pot, he can *raise* the bet (i.e., increase the value of the bet).

A betting round continues until one of three conditions occurs. First, if a player makes a raise, all of the opponents must call his raise. This case is also true for the blinds (i.e., the blinds can be seen as forced raises). If a raise is called, and not *re-raised* (i.e., raised again by another player), and the play returns to the player that made the original raise, then the betting round is considered over. There is no limit placed upon the number of re-raises that are allowed in a single betting round. Second, if all of the players fold their cards, the remaining player wins the hand, and collects all of the money in the pot. A player goes *all-in* by betting all of his money at one time (or by being forced to do so to call another player's raise). If there is only one player left in the hand with any money left to bet, no further raising is allowed since there will be no one who can call his raise, and the hand proceeds to the showdown.

If a player goes all-in, there is the possibility that there are two or more players that will call or raise. After these other players have called the raise, if they still have money remaining, they are allowed to continue betting in a *side-pot*. The *main-pot* consists of the all-in bet from the one player, as well as the money from the other players that was used to call this raise. No more money will be added to this pot. Any money that is bet after the side-pot is created will be inserted in the side-pot. At the showdown, all of the active players can win the main-pot, but only the players that have contributed to the side-pot may win it. If the player with the best hand has not contributed to the side pot, then he will win the main pot, and the side pot will be won by the contributing player with the best hand. If, in the course of betting, there are multiple all-ins, then there can be multiple side-pots.

2.1.2 Gameplay

After the hole cards are dealt, and the blinds are posted, there is a betting round. After the betting round has completed, three community cards, known collectively as *the flop*, are dealt. Another betting round follows the flop. A fourth community card, known as *the turn* or *fourth street* is dealt, followed by another betting round. A fifth and final community card, called *the river* or *fifth street* is dealt, followed by a final betting round. After the final betting round, if there are still at least two players, all cards are revealed, and the player with the best five-card hand made up of any of the seven cards (i.e., the two hole cards and five community cards) wins the hand. If two players have equal hands, then a *split-pot* is awarded, with each winner taking an equal share of the pot. Occasionally, a hand will end without a showdown. If after any betting round, only one player remains (i.e., all of the other players have folded), this player wins and does not have to reveal his cards. Similarly, if a player folds, he does not have to reveal his cards.

2.2 Strategy and deception

No-Limit Texas Hold'em is as much a game of skill as it is a game of luck. Knowing when to fold and when to raise is not as simple as *playing the cards* (i.e., folding with poor cards and raising with good ones). Players who do so rarely do well at No-Limit Texas Hold'em. If a player's opponents can learn that a player plays this strategy, they would only call the player's bet when they themselves have good cards, lessening

the likelihood that the original player will win. Instead of simply playing the cards, good Texas Hold'em players try to disguise the cards that they are holding. The only difference between a player's hand and the hand of an opponent is a player's hole cards, and thus, if a player can represent his hole cards as being much better (i.e., *bluffing*), or much worse (i.e., *sandbagging*) than they really are, it is harder for his opponents to determine what his cards are, and harder for them to exploit his playing style.

2.3 Counter-strategies

The best Texas Hold'em players are the ones that can correctly approximate their opponent's hole cards, and disguise their own. Furthermore, players who can discover tendencies in their opponents' strategies are often able to exploit them. For example, if an opponent rarely calls large bets, a good player will make large bets with weaker cards, hoping that his opponent will fold, but will make a small bet when he wants a call.

As players become better, the cards become less relevant. Instead of *playing the cards*, good players *play their opponents*. Texas Hold'em becomes a game of chicken, where players are constantly analyzing their opponents for weaknesses, while at the same time trying to remove any weaknesses of their own. Strategies meet counter-strategies, and are replaced by counter-counter-strategies, all in an effort to deceive the opponents.

2.3.1 Bluffing

A player will often bluff to try to scare his opponents away from a pot. A *semi-bluff* occurs when a player has a weak hand that could improve with future cards. Although the player does not have a good hand, he raises with the hope that he will get the cards that he needs. A *draw* occurs when a player needs only one or two cards to complete a hand, such as having four cards of one suit (i.e., a fifth card of that suit will complete a flush). If a player receives the cards that he needed to complete his hand, he is said to have *hit* his hand. If the player does not receive the cards, he is said to have *missed* his hand. Even if the opponent calls, there is still a chance that the player will hit his hand, and win the pot. A *full-bluff*, also sometimes called a *stone-cold bluff* or a *pure bluff*, occurs when a player does not have a hand, but

wants to *steal* (i.e., win without earning) the money in the pot. Often, if a player has missed a draw, he will bet as if he hit it, to try to scare his opponent away. Usually, a bluff is a rather large bet, that would be expensive for an opponent to call. If a player *reads* a bluff incorrectly (i.e., assumes an opponent is bluffing when he is not), it can cost him a lot of money, but if he reads it correctly (i.e., calls a large bet by the opponent when he was bluffing), it can pay quite well.

2.3.2 Sandbagging

Sandbagging is the opposite of bluffing, and involves *slow-playing* one's hand (i.e., representing one's cards as worse than they are, such as by calling instead of raising). Usually, sandbagging is used to two ends. The first is to try to get opponents to place more money in the pot. Whereas a large bet may scare away opponents, a small bet may get opponents to call, or even to raise. Secondly, sandbagging can be used to gauge the strength of an opponent's cards. If a player is in *early position* (i.e., betting early in a round), then he has not seen how the other players are betting, and is less likely to make an accurate approximation of his opponents' cards. He may simply call, even with a good hand, to see if the other players are eager to bet, or are cautiously calling. A common sandbagging technique is the *check-raise*, where a player checks when it is his turn to bet, and once his opponent bets, he re-raises his bet. If the opponents fold, the player wins at least the amount of the bets that were called before he raised, and if they call, he gets the same amount as he would have if he had made the bet instead of the check.

Chapter 3

BACKGROUND AND RELATED WORK

Early research into games often avoided those involving chance and imperfect information due to the large decision space that exists in such games [7]. For that reason, the research literature on Poker is not as large as that for games where chance is excluded, such as Chess and Checkers. Furthermore, the imperfect information inherent in Poker further complicates the process of decision-making, and has been a factor in limiting research conducted in the area. In spite of these challenges, research into Poker as a problem in Artificial Intelligence has recently attracted more attention.

In this chapter, we describe relevant work in developing computer Poker agents. Section 3.1 will describe Poker research at the University of Alberta by the largest group studying Texas Hold'em. Section 3.2 will describe recent research in No-Limit Texas Hold'em. Section 3.3 will be concerned with evolutionary methods, with the emphasis upon evolving self-play and its application to games.

3.1 The University of Alberta Computer Poker Research Group

The University of Alberta has the largest computer Poker research group in the world, and their research has culminated in some of the best Limit Texas Hold'em playing agents [21, 22]. Within this group, several approaches to the development of a Poker-playing agent have been investigated. These approaches are described in the next few sections.

3.1.1 Poker by simulation

One of the first agents developed by the University of Alberta was Loki [10, 5, 31], an agent for playing multiple-player Limit Texas Hold'em. The earliest version of

Loki [10, 31] used an estimation of hand strength and potential to make betting decisions. The number of potential hands that were better than, tied with, or worse than Loki's hand were determined, and the probability Loki's hand was the best hand was generated. Opponents were not *modeled*; Loki did not store any information regarding previous betting decisions made by its opponents. Opponent actions were instead estimated assuming that an opponent would always play according to its cards. The different sets of hole cards that an opponent might be holding are weighted according to Loki's estimated likelihood that the opponent would be holding those cards at a certain point in the game. It is assumed that an opponent will always make the best decision. This opponent modeling strategy was eventually abandoned due to its static nature.

A later version of Loki [5] substituted a more robust analysis of the game state for betting decisions. Rather than determine a betting decision by consulting a static rule, this version of Loki attempted to make betting decisions based upon *expected value* obtained through a hand simulation. Opponents are given expected probabilities that they will fold, call, or raise any hand, and the current hand is simulated several times for each decision that Loki can make (i.e., once for a fold, once for a call, and once for a raise).

Opponents in this version of Loki are modeled; that is, as opponents play hands, Loki keeps track of potentially useful information about the decisions made by the opponents, and the state of the game when the opponents made the decisions. When it comes time for the agent to simulate the *rollout*, or completion of the current hand, the agent is able to more realistically predict how its opponent will bet whenever it is its turn to do so. Through multiple simulations of the current hand, the agent is able to determine the average amount of money that it would win or lose, given that it makes a particular decision. This money corresponds to an expected value. The decision with the highest expected value is selected by the agent.

The effectiveness of Loki was evaluated using the amount of money won, using the metric of small bets per hand (sb/h). Sb/h is a metric that is often used to measure the skill level of both human and artificial players. A small bet is the amount of a small blind in a game of Texas Hold'em. The money won by a player is normalized by the size of the small bet and the number of hands played to obtain an objective measure of the player's skill.

In Chapter 4, an alternative metric is described for evaluating an agent's skill.

The sb/h metric is not as useful when evaluating No-Limit Texas Hold'em playing agents. Loki was evaluated using what amounted to an infinite chip stack; whenever Loki ran out of chips, it was given more. While this is acceptable in Limit Texas Hold'em, as bets are structured, and there is a maximum bet, it is unacceptable in No-Limit Texas Hold'em. If an agent knew that it had access to unlimited chips, the meaning of a maximum bet is lost, and there is no real motivation for not betting all of its chips in every hand. Furthermore, Loki uses a static blind (i.e., blinds do not increase as described in Chapter 2). When blinds are increasing, the meaning of a small bet becomes less defined.

The later versions of Loki [5] are more effective than the earlier version [10, 31]. When Loki estimates an opponent's actions according to the probability that they will make a fold, call, or raise (not assuming they will always make the best choice), Loki outperforms the original Loki by 0.023 sb/h. When Loki adds a similar strategy for itself (i.e., not simply making the best decisions, but making stochastic decisions), it outperforms the original Loki by 0.044 sb/h. When Loki then adds hand simulation, it outperforms the original Loki by almost 0.08 sb/h.

Poki [7, 14, 15], is an extension of Loki, using some of the same techniques that were implemented in Loki, yet with a greater concern for opponent modeling. Poki uses weight-tables to predict the probability that an opponent has a combination of two cards, given how it has bet up to the current point in the game. The weights in these tables are adjusted after each action by the opponents. Two methods of opponent modeling, named *general opponent modeling* and *specific opponent modeling* are introduced. General opponent modeling modifies the weights in the table according to some pre-determined formula. Specific opponent modeling takes the opponent's playing style into account, and modifies the weights according to previously observed data regarding betting tendencies.

Originally, Poki simply observed the betting tendencies of the opponents, and developed a *median hand strength*, which it would determine as the cards most likely held by the opponent. This version was deemed to be overly simplistic. Other features of Limit Texas Hold'em's game space were integrated into the re-weighting algorithm, such as the number of active players, the amount of bets to call, and the position of the betting agent.

All versions of Poki use an artificial neural network that is used to predict an opponent's next action, trained using hands from real Poker tournaments. Using this

prediction, along with an estimation of the opponent's cards, Poki makes its betting decisions. A baseline agent with no opponent modeling was tested against a pre-fabricated opponent, and played at a maintenance level, winning an average of 0 sb/h. An improvement of 0.08 sb/h was observed using the original opponent modeling method, and further improvement to 0.22 sb/h was observed using the newer method. This increase in skill was used to demonstrate the importance of opponent modeling in the game of Poker, and it can be reasoned that stronger opponent modeling can lead to better playing agents. As research began to concentrate more on opponent modeling, the focus shifted from multi-opponent Poker to heads-up Poker. For instance, Loki and Poki played ten-player Texas Hold'em, but subsequent agents focused on playing games against a single opponent.

3.1.2 PSOpti

The agents described in [4], named PsOpti, use a game theoretical solution to try to lose as little as possible. PsOpti agents play a similar strategy against any opponent, substituting robustness for exploitation.

The goal of a game theoretic solution is to give an agent a strategy such that he can play no better if he makes alternate decisions. The largest problem with creating a game theoretically optimal solution is that Texas Hold'em Poker has such a large decision space that complete analyses of the space are infeasible. In [4], an abstraction of the game of Limit Texas Hold'em Poker is used instead, to reduce the size of the decision space. After analysing this decision space, several versions of PsOpti were developed. PsOpti0 eliminated pre-flop betting, and ignores prior probabilities for hand distribution (i.e., assumes any hand is equally likely, despite likelihood of weak hands progressing). PsOpti0 also does not distinguish between different post-flop cases (i.e., one raise, two raises, etc). PsOpti1 is the same as PsOpti0, except that it uses a game theoretical solution for the pre-flop case, and considers four post-flop models. PsOpti2 uses a different game-theoretical solution for the pre-flop case, which then gave prior probabilities for seven post-flop models.

PsOpti assumes that an opponent is also trying to minimize loss. Possible cards held by an opponent are grouped together according to the strength of the hands they would form if the opponent is holding them. PsOpti tries to determine the likelihood that any of the particular groups will occur. PsOpti judges the strength of all potential hands, by comparing which hands will win, lose or tie against each other,

and is able to obtain an expected value for its own hand. Similarly, hand simulations are conducted to determine if hands have the potential to improve or decrease in value in later rounds. Decisions are made based on the hand strength of PsOpti's hand, as well as the likelihood that the opponent has a hand of a certain strength. The abstraction of the game space by grouping many hands together allowed a game theoretic solution to be obtained, while maintaining most aspects of the full decision space of Limit Texas Hold'em. The PsOpti agents group similar hands into "buckets" to reduce the size of the decision space. PsOpti0 and PsOpti1 use seven buckets, while PsOpti2 uses six.

PsOpti0 saw little gain against Poki, only winning 0.001 sb/h. PsOpti1 and PsOpti2 saw greater gains, winning an average of 0.045 and 0.047 sb/h against Poki, respectively. The best version of PsOpti, named PsOpti4 (based upon an improved version of PsOpti2) was capable of winning 0.093 sb/h against Poki. Even though PsOpti was only concerned with a smaller abstraction of the game, and did not exploit any of Poki's weaknesses, it was able to win against Poki, and also made gains of up to 0.4 sb/h against other benchmark agents. Unfortunately, while a game theoretic solution is robust against numerous agents, it exploits the weaknesses of none of them, and can perform less than optimally against weaker players (it plays to lose as little as possible, but weak players are less likely to win, and a game theoretic solution does not take risks that might prove profitable).

3.1.3 VexBot and BRPlayer

PsOpti tries to abstract the game of Poker, using smaller models of the game to make decisions in the full-scale version. Vexbot [8], and BRPlayer [38], attempt to further abstract the game, but do so in a different manner than PsOpti. An *expectimax search tree*, an extension of a decision tree for stochastic decisions is implemented and expanded for situations with imperfect information. The nodes of the tree represent both betting decisions and actions that can be attributed to chance. At each of these nodes, a branch is created for each possible outcome. Chance nodes can branch very quickly, and as each opponent can be holding any combination of two randomly assigned cards, the tree can become very large. An agent, however, does not know the cards being held by its opponent, and thus all nodes at a particular level corresponding to different cards can be merged into one decision node (i.e., all the nodes will appear the same to an agent, so there is no need to distinguish them). The tree then only

branches upon decisions, which are observable. At any leaf node of the tree, an agent can receive an expected payout, determined using an estimated strength of the opponent's cards. An agent can do a search for the highest expected payout, and disregard betting actions that will lead to less profitable areas of the tree. This method has been named Miximix, with a special case named Miximax (where an agent must choose the solution with the highest expected value, rather than choosing stochastically).

A leaf node must always be a fold or a showdown (due to the fact that if either player folds, no further search is necessary). The expected value at a node where the opponent folds is always equal to all of the money in the pot. If the node is a showdown, the expected value is weighted by the likelihood that the agent's hand will win. Each potential hand has an expected value associated with it (i.e., if the opponent's hand loses to the agent's hand, the expected value is all of the money in the pot, whereas if the opponent's hand wins, the expected value is equal to -1 times the money in the pot). Opponent modeling is used to gain an estimation over which hands are likely for the opponent, assuming that it uses the same strategy that it has been using. An estimated hand value can be obtained, and the expectimax search can find the highest expected value.

BRPlayer further extends Miximix by investigating two forms of opponent modeling. one considering the actions of the opponents and one considering the reaction of the opponent (called the *strategy model* and the *observation model*, respectively). The strategy model attempts to determine what decision an opponent is likely to make, given that the game is in a current state, based upon prior observations of the opponent. For example, if the hand is in the post-flop phase, and there are three bets to call, and the opponent has an estimated hand value of 0.5, then it will fold 25% of the time, call 50% of the time, and raise 25% of the time. The observation model, on the other hand, makes no assumptions about the current cards being held by the opponent. The observation model instead observes the reactions that the opponent makes whenever the agent plays. For example, if the game is pre-flop, and the agent is holding a queen and a ten of the same suit, and raises, it is satisfactory to know that the opponent has folded once and called twice in this situation in the past. At showdown, if cards are revealed, the observation model is updated, but the cards are not necessary to model the opponent. The strategy model is very similar to the one used for opponent modeling in Poki, while the observation model is a different

approach that tries to place less emphasis on the hidden information of Limit Texas Hold'em Poker.

Vexbot was evaluated against both Poki and PsOpti4. Against Poki, Vexbot saw gains of 0.601 sb/h, and against PsOpti4, Vexbot won an average of 0.052 sb/h. BRPlayer, was implemented using an observation model for opponent modeling, and was evaluated in heads-up games against Poki and PsOpti4. In an attempt to overcome any luck attributed to chance, each game consists of 40,000 hands. Across three matches with Poki, BRPlayer won an average of 0.743 sb/h. In short game evaluations (i.e., the first 5,000 hands of the 40,000 hand games), BRPlayer won an average of 0.511 sb/h. Against PsOpti4, BRPlayer won an average of 0.110 sb/h. Miximix trees and observation models are able to defeat Poki and PsOpti, which had previously been the strongest Limit Texas Hold'em Poker playing agents.

3.1.4 Hyperborean and Polaris

The game theoretic Limit Texas Hold'em Poker model is extended in [21]. Larger abstractions are possible (i.e., more betting rounds and less bucketing), and an attempt is made to correct the non-exploitable nature of PsOpti. An improvement is made to how PsOpti handles hand potential, trying to give more weight to hands that improve later in the game. Several types of strategies are developed, to be used in different game situations.

Counterfactual regret minimization penalizes players for making decisions that result in a loss, and as such, attempts to arrive at a solution very similar to PsOpti's where an agent tries to lose as little as possible. As agents play, they determine the expected value of making any number of betting decisions, and weight these expectations by the probability that they will make the decision. They compare it to the expected values if the betting decisions had 100% probability, and denote the difference. This difference is likened to the regret that a certain action was not taken more often. The probabilities of the betting decisions are then re-assigned to better reflect this regret for the next time that the game enters this type of situation. As the algorithm progresses, the agents learn which percentages are best for given game situations. Evaluated against the best of the PsOpti agents, this agent, known as CFR, achieved an average of 0.042 sb/h.

To counter the lack of exploitability of PsOpti and CFR, a *best response* strategy

was developed. A best response strategy can be considered the opposite of a game theoretic strategy, as it is implemented with the sole purpose of exploiting an opponent. Agents are trained using results from previously observed matches. Unfortunately, a best response needs to know the exact strategy of the opponent. Furthermore, if there are more than two players, the best response agent may only be playing a strategy against one opponent, and will lose to the other opponents. In the best cases, where the agent uses the correct counter strategies, gains of up to 0.33 sb/h are observed, although the agents generally lose when they choose the incorrect strategy.

Two strategies are then presented which take the best of both situations: they allow players to play robustly against a number of opponents, while exploiting opponents that are weak. The first strategy (known as *Restricted Nash Response*) weights the probability that the strategy will use a game theoretic solution against the probability that it will use an exploitable solution. The weights are adjusted to maximize an agent's winnings, and gains of 0.085 sb/h are observed against PsOpti4. Furthermore, gains of 0.026 sb/h are observed against PsOpti4, when the agent was trained against another agent, PsOpti6 (which, despite the higher version number, performs worse than PsOpti4).

The second strategy uses a *meta-agent* to determine the best strategy to play. A team of strategies is combined, and a strategy is chosen as the active agent. If, after a few hands, the agent is performing poorly, the strategy is switched with another one, and so on, until an appropriate exploiting strategy is found. If no such strategy exists on the team, a game theoretic agent can play, and the loss can be minimal. Although this strategy seems more robust, the cost of using inappropriate strategies for even a small number of hands results in a loss to PsOpti. However, if the exploiting strategy is not on the team, the best that the agent can hope for is to play to lose as little as possible.

This team strategy was used to created two agents, named Hyperborean and Polaris. Hyperborean comprises a team of PsOpti4 and PsOpti6, and was submitted in the 2007 AAAI Computer Poker Competition in Limit Texas Hold'em, where it placed first. Polaris was comprised of a set of different teams of agents, including several game-theoretic agents, as well as several Restricted Nash response agents, and competed against two professional Poker players in the first Man-Machine Poker Competition in 2007. Polaris played four matches against the professionals, and lost by a margin of one win, two losses, and one tie (a tie was awarded if after a set

number of hands, the difference in sb/h was insignificant). Polaris had a rematch in the Second Man-Machine Poker Competition in July 2008, and won the match with three wins to two losses and a tie.

3.2 No-Limit Texas Hold'em

Although the agents developed by the University of Alberta Computer Poker Research Group were intended to play Limit Texas Hold'em, they could also be adapted to play No-Limit Texas Hold'em. Since the only difference between the two variants is the betting structure, any Limit Texas Hold'em agent can become a No-Limit Texas Hold'em agent if it is changed to include a No-Limit betting algorithm. However, when it comes to research for agents developed solely for the purpose of No-Limit Texas Hold'em, the literature is significantly scarcer than that for the limit variation. Texas Hold'em is by itself a very complex game, regardless of the betting structure. Many of the attempts to previously create agents for Texas Hold'em have focused on an abstraction of the entire game, or a reduction of the search space.

When betting is changed from limited to unlimited, as in no-limit Poker games, the game space increases in size and complexity. Thus, very few efforts exist where the main research goal was to develop an agent capable of playing No-Limit Texas Hold'em. Furthermore, adapting Limit Texas Hold'em playing agents to play No-Limit Texas Hold'em might not be the best approach. No-Limit Texas Hold'em introduces game situations that cannot occur in the limit variant, and an adapted agent may not play well in these situations. For example, a strategy that was effective in Limit Texas Hold'em might treat all raises equally, while in No-Limit Texas Hold'em, a very large raise should be handled differently than a small raise.

3.2.1 Rule-based No-Limit Texas Hold'em

The only known previous AI system for playing No-Limit Texas Hold'em is a rule-based system [11], which we will refer to as NLBot. Four versions of No-Limit Texas Hold'em agents are presented (NLBot1, NLBot2, NLBot3, and NLBot4). NLBot1 uses no forms of opponent modeling, but only relies on the strength of its cards. Using expert-derived rules, the agent uses these card values to make betting decisions. The NLBot agents were not developed at the University of Alberta, and as such, access to the code of such agents as Poki, PsOpti and BRPlayer was restricted, and

NLBot was not evaluated against these agents. This agent plays heads-up No-Limit Texas Hold'em against an agent that calls, folds and raises with equal probability (RandomBot), as well as an agent that always calls (CallBot), and one that always raises (RaiseBot). It plays in tournaments against the opponents until one of the two agents does not have any chips remaining. Against RandomBot, NLBot1 wins approximately 85% of its games across 500 tournaments. Against CallBot, NLBot1 wins 90% of its games, and against RaiseBot, NLBot wins approximately 60% of its games.

NLBot2 extends the implementation of NLBot1 by incorporating opponent modeling. NLBot2 stores information regarding the cards that the opponent went to the showdown with. Against static opponents, this opponent modeling will have little use (as the opponents make decisions based on information other than their cards). The results against RandomBot, CallBot, and RaiseBot show little difference from the results of NLBot1.

NLBot3 further weights the likelihood of an opponent having a better hand, given that a particular type of flop, turn, and river are dealt. For example, a flop of three different suits does not allow flushes, and thus the likelihood of an opponent playing to the showdown with certain hole cards needs to be re-weighted. Again, the results against the static opponents show little improvement of the agent.

NLBot4 tries to estimate the likely hole cards of the opponent based upon the amount of money that the opponent raised in certain situations. Average hand values are calculated for situations involving folds, calls, and raises. Estimations of the opponents hands can be made by observing how much money is called or raised. The results against the static opponents are again unchanged.

The four implementations were then inserted into a tournament structure, and played multiple games against each other. The agents played a total of 175 tournaments, with NLBot1 placing first, and NLBot4 placing second. The two best agents were then tested against an amateur human player. NLBot1 begins well, winning 7 of the first 10 tournaments. However, the human player learns that it is only playing its cards, and takes advantage of it, winning 9 of the last ten tournaments. NLBot4 plays better, winning almost 50% of 50 tournaments, and it never has a streak of poor play similar to the final ten tournaments of NLBot1.

3.3 Evolutionary methods and self-play in games

Evolutionary methods try to model the biological process of evolution. Agents are evaluated, and the agents best suited to their task are selected as representatives of their generation. New agents are created that are virtually identical to the last generation's best agents, albeit with several small changes. These new agents, as well as the best agents from the previous generation, are re-evaluated, and the process repeats. Gradually, the new agents replace the old agents, and a global increase in performance is achieved.

Agents are often evaluated by a *fitness function*, a formula designed to determine the quality of the agents. When evolving agents to play games, often, the best indication that an agent is improving is by denoting how many games it can win against a static opponent. This can prove problematic when good opponents do not exist, or when the opponent is much better than the evolving agent (the agent might be improving, but it is impossible to determine, as it wins no more games against the superior opponent). These restrictions have inspired the concept of *self-play*. In self-play, agents play against versions of themselves from earlier generations. The champion versus challenger format sees a new agent (the challenger) play against the best agent from a previous generation (the champion). If the champion wins, a new challenger is produced, and plays against the same champion. If the challenger wins, it replaces the champion, and a new challenger is produced. Evolutionary methods and self-play have seen numerous successes in various games, and are described in the following sections.

3.3.1 Checkers

In the 1950's and early 1960's, Samuel was applying the concepts of self-play to the game of Checkers [35]. By using a *scoring polynomial*, and altering the terms after each play made by a Checkers agent against another static agent, better playing styles were promoted through positive reinforcement. If a move led to a better position on the board, the function was altered to promote similar moves in the future, and if it led to a worse position, the function was altered to discourage similar moves in the future. Later work [36] introduced *signature tables*, which grouped various moves according to board positions. Decisions made by the signature table agent were compared to expert knowledge, and were determined to be better than the scoring polynomial.

3.3.2 Chess

Although Chess programs developed through other methods are capable of defeating the best human players [12, 16], attempts at evolving a Chess playing agent have recently been made. A large part of the skill of Chess-playing agents may be attributed to power, rather than skill [18]. Heuristic searches and looking ahead many moves into the future may be sufficient to defeat human players, but it has not prevented the search into alternative and possibly more efficient methods, including evolutionary techniques.

Temporal difference learning has been applied to the game of Chess [44], implemented with an artificial neural network. Each time that the agent must make a move, all legal moves are presented to a neural network which selects what it considers the best move. After the move is made, the state of the game is compared to the state prior to the move, and a temporal difference algorithm is used to adjust the weights in the neural network. The developed agent was able to defeat a commercially available program roughly one time in eight games. While not as good as the best programs, the temporal difference approach showed improvements over back-propagation, another machine learning technique.

A simplified evaluation function is used to compare the states of the board at any decision node [23]. The evaluation function considers the importance of the various pieces in the game and the number of legal moves available to the agent, and weights the pieces according to their importance in winning the game. After each game, the weights change according to whether the agent won or lost the game. The evolution process improved the playing ability of the agent, which was evaluated against a commercially available program. The agent was matched against successively more difficult agents to determine the skill level of the agent, and it unofficially achieved near expert status.

An alpha-beta search heuristic is combined with genetic programming with the goal of creating a moderate Chess agent without the use of expert data [19]. The best Chess playing programs in the world have access to a large library of opening and endgame strategies. The goal of a self-playing agent would be to discover the strategies covered by such a library, but to find them through experimentation and exploration. The agents analyse the state of the board, and use a fitness function to evaluate future moves. This fitness function sums weighted values of various features of the game of Chess. The weights of different features of the game are allowed to

change after every game, and more games are played with the new fitness function. The resulting agents were evaluated against traditional Chess agents, and performed similarly to agents that can look ahead five moves. The evolutionary process allowed the created agent to reduce the complexity of the required search tree and perform more efficiently than the standard brute-force alpha-beta search.

3.3.3 Backgammon

Evolutionary algorithms were used to evolve an agent that could play the game of Backgammon [42, 43]. Whereas Checkers is, like Chess, a deterministic game, Backgammon is not. The use of dice increases the decision space of Backgammon to approximately 400 legal moves each time a player gets a turn, as opposed to 30 for Chess and 10 for Checkers [43].

Temporal difference learning [42, 43] is used in TD-Gammon. Although the agent starts with random weights for a neural network, and plays only against versions of itself, the agent does learn how to play near world-class level Backgammon [42]. Furthermore, increasing the number of generations, the depth of the search, and optimizing the parameters of the neural network resulted in agents that were better than the very best human players [43].

That being said, interesting results were achieved by systems that did not incorporate temporal difference learning, but rather used a simple hill-climbing and simulated annealing approach [32]. The process begins with two agents, each with a neural network where the weights of the nodes were all zero. The agents play against each other for a pre-determined number of games, and the better agent becomes the *champion*. A new agent, called the *challenger*, is created by applying random noise to the nodes of the champion's neural network. More games are played, and one of two outcomes is observed: either the challenger is better than the champion, or the champion is better than the challenger. In the former case, the weights of the champion are re-calculated as a weighted mean of those of the old champion and the old challenger, and a new challenger is created in the same manner as before. In the latter case, the champion remains unchanged, while a new challenger is generated from the champion. While the results were not as impressive as in [42], the final agent won nearly 40% of games against a public benchmark, while using a much simpler evolutionary algorithm.

3.3.4 Go

Like Poker, the game of Go has a very large potential decision space. Unlike Poker, Go is a game of perfect information, and is deterministic rather than stochastic. The game has a large branching factor, and multiple goals need to be observed to guarantee a win. These factors make the game very hard for agents with artificial intelligence to play well [33].

Using a temporal difference algorithm very similar to [42], in [39] self-play was used to create an agent capable of playing small-board Go. Small-board Go has a lower branching factor, and is somewhat easier for computer agents to learn. Although only the basic rules of the game were given to the agent to begin with, it was able to eventually evolve a player that played a winning strategy. Evaluated against a commercially available Go program, the agent was able to win against some of the lower difficulty levels on small-board Go.

The SANE architecture [33] extended the work performed in [39]. Rather than using temporal difference learning, it uses a stochastic selection procedure that is based upon the standard evolutionary methods of recombination and mutation. Rather than manipulating the weights of the neural networks directly, as is common in neural network algorithms, including temporal difference, SANE maintains two populations relevant to the neural networks: *neurons* and *blueprints*. Neurons maintain information about their connections within the network, as well as the weights of the stated connections. Blueprints are only concerned with which neurons are present within the network, regardless of their literal values.

The architecture of the network is fixed, and nodes are swapped at reproduction and mutation phases. Furthermore, mutation and evolution can occur on the neurons themselves, changing the connections and the weights associated with them, although randomly, as opposed to a directed change. Playing on small Go boards, the SANE technique is able to create a player that can play simple Go, without having been taught any strategy. One outlined problem, however, was that the player was more exploitative than skilled, due to a lack of opponents against which it could learn. Tested against the same commercially available program as the agent developed in [39], the SANE agent was able to win roughly 75% of the time. Unfortunately, the networks had a tendency to overfit to the opponent.

This problem of exploitation versus skill is tackled in [26]. Rather than play against a static opponent, the SANE agents were played against each other. Players

were split into two *populations*, one called the *hosts*, and the other called the *parasites*. The goals of the agents were distinctly different between the two populations, with the overall intention of promoting generality in the playing styles of the host agents. The goal of host agents was to win Go games against a general group of parasites. The goal of the parasites, on the other hand, was simply to exploit the weaknesses of a particular host agent. Furthermore, agents were tested against champions from previous generations that were retained in a hall of fame, to ensure that progress was occurring. Although results are not evaluated against any external player, an improvement is shown for the agents evolved using co-evolution against dynamic opponents, as opposed to agents evolved against a static opponent.

3.3.5 Poker

Evolutionary methods have also been applied to Poker. In [2] agents are evolved that can play a restricted version of Limit Texas Hold'em that has only one betting round. Betting decisions are made by providing features of the game to a betting formula. The formula itself is evolved by adding and removing parameters as necessary, as well as changing weights of the parameters within the formula. Evolution is found to improve the skill level of the agents, allowing them to play better than agents developed through other means.

In [29], some of the pitfalls of evolutionary techniques are investigated. Although evolution allows a steady progression from generation to generation, in certain games of chance, skill is not transitive. If an agent A can defeat another agent B, and B can defeat a third agent, C, there is no guarantee that C can defeat A. Unlike the majority of other Poker research, the created agents are intended to play against several opponents, rather than only against one opponent (i.e., heads up). Thus, adaptive neural networks are created to try to find optimal co-evolutionary strategies. In other words, the agents created are meant to play well against a wide variety of opponents, rather than simply being able to exploit the weaknesses of a single weak opponent. Like the game theoretical agents of the University of Alberta, the agents attempt to minimize their losses, while maximizing their winnings against numerous opponents. Results show that evolutionary methods, particularly co-evolution, can learn to play the game of Poker. The co-evolved agents out-performed several evaluation agents, including one that modeled "reasonably good play" [29].

No-Limit Texas Hold'em Poker agents were developed in [3], and were capable

of playing large-scale games with up to ten players at a table, and in tournaments with hundreds of tables. Evolutionary methods were used to evolve two-dimensional matrices corresponding to the current game state. These matrices represent a mapping of hand strength and cost. When an agent makes a decision, these two features are analysed, and the matrices are consulted to determine the betting decision that should be made. The system begins with some expert knowledge (called a *head-start approach*). Agents were evolved that play well against benchmark agents, and it was shown that agents created using both the evolutionary method and the expert knowledge are more skilled than agents created with either evolutionary methods or expert knowledge alone.

Chapter 4

TEXAS HOLD'EM, EVOLVING NEURAL NETWORKS, AND COUNTERING EVOLUTIONARY FORGETTING

This chapter describes the application of evolving neural networks to the game of No-Limit Texas Hold'em and introduces countermeasures to deal with the problem of evolutionary forgetting. Although the agents developed in [35], [42], and [32] were able to achieve respectable results without such countermeasures, evolutionary forgetting can hinder performance and undermine results [34].

Section 4.1 describes the application of the evolving neural network to the game of No Limit Texas Hold'em. Section 4.2 continues by describing the countermeasures implemented in an attempt to prevent evolutionary forgetting. Section 4.3 describes how agents are evaluated.

4.1 Evolving neural networks

4.1.1 Overview

Although subject to certain variations, the evolutionary methods used to evolve neural networks to play various games follow a general pattern. Agents compete against each other, and a *fitness function* is used to evaluate the skill of the agents. When evolutionary algorithms are applied to competitive games, an agent's fitness can be evaluated by determining which agents can win the most games. Due to the costs of multi-agent populations, these evaluations are usually implemented as a challenger versus champion format as described in Section 3.3.3.

The agents developed to play No-Limit Texas Hold'em make their betting decisions by supplying various features of the game to a multi-layer artificial neural network. The network analyses the input, and output is given in the form of a *probability vector*. This vector contains the probabilities that the agent should fold, call, or make a small,

medium, or large raise, given the current state of the game.

The development of the agents can be broken down into two phases: the evaluation phase and the evolution phase. In the evaluation phase, agents play No-Limit Texas Hold'em Poker in a series of tournaments, and the best ranked agents proceed to the evolution phase. In the evolution phase, new agents are created through an evolutionary algorithm, and refill a population of agents, which are then returned to the evaluation phase. The algorithm alternates between the two phases for a pre-determined number of *generations*.

4.1.2 The evaluation phase

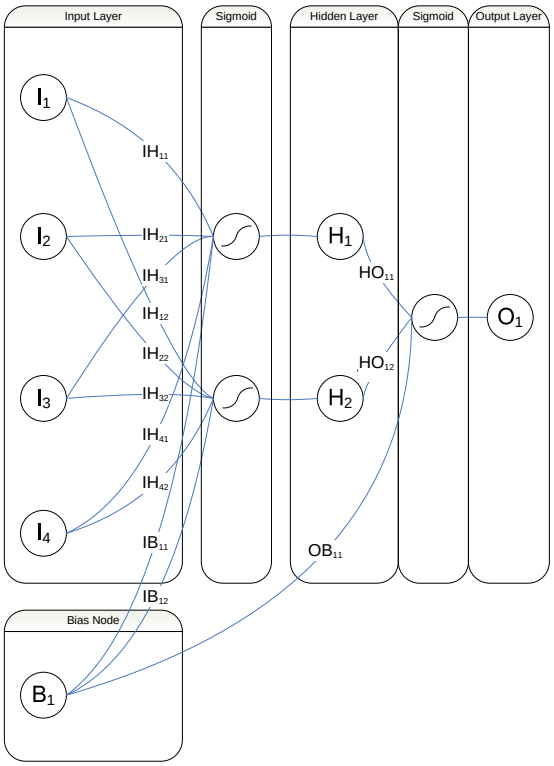

Figure 4.1: A sample 4-2-1 feed-forward neural network

Figure 4.1 shows a sample multi-layer, feed-forward neural network. Each node, or *neuron* of the network belongs to one of three layers: the *input* layer, the *hidden* layer, or the *output* layer. In the sample neural network, there are 4 input neurons, 2 hidden neurons, and 1 output neuron, so it is called a 4-2-1 neural network. The sample neural network is simpler than the one used in this thesis, but is easier to

understand. The neural network shown in Figure 4.1 is said to be fully-connected, since each neuron of the input layer is connected to each neuron of the hidden layer and each neuron of the hidden layer is connected to each neuron of the output layer. In addition to the Input, Hidden, and Output layers, an additional neuron, known as the *bias* is connected to the network, but does not belong to any particular layer. The bias is an extra node that always stores the value of 1.0, and is connected to both the hidden and output layers of the network. In Figure 4.1, the connections are designated IH_{xy} for a connection between a neuron I_x in the input layer and H_y in the hidden layer. Similarly, connections between H_y in the hidden layer and O_z in the output layer are denoted HO_{yz}. Connections between the bias (i.e., B_1) and H_x in the hidden layer are denoted IB_{x1}, and connections between the bias and O_z in the Output layer are denoted OB_{z1}. Each connection is weighted by a real value.

The neurons each store a real value. The values stored in the input level are computed directly from the data that is being analysed. For example, if this neural network were being used to find a solution to the XOR logical operation for two-bit operations, the first two neurons of the input layer, I_1 and I_2, could represent the first operand, and the second two, I_3 and I_4 could represent the second operand. Thus, if the values 10 and 01 were being compared, then I_1 would equal 1, I_2 would equal 0, I_3 would equal 0, and I_4 would equal 1. The output from the input layer is the same as the stored values in the neurons and proceeds to the hidden layer.

Before the input is accepted by the hidden layer, it is manipulated by a *non-linearity* function. This function ensures that the network can provide non-linear mappings between nodes. The function used is the *sigmoid* function, and is defined in equation 4.1.

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{4.1}$$

.

The value that is evaluated with the sigmoid function at any neuron in the hidden layer is a sum of the inputs to that neuron, weighted by their connection weights. For example, in Figure 4.1, the value of the input to node H_1 is represented in equation 4.2, with x between 1 and 4.

$$H_1 = sigmoid(\sum (I_x * IH_{x1}) + (B_1 * IB_{11})) \tag{4.2}$$

The result is a non-linear, weighted sum of the inputs to the system. Each neuron in the hidden layer receives its respective input using a similar formula. After the

values of the hidden layer are computed they are stored in their respective neurons. These values are then transformed using a similar formula, and passed to the output layer. For example, in Figure 4.1, the value of the output neuron O_1 is calculated using equation 4.3.

$$O_1 = sigmoid((H_1 * HO_{11}) + (H_2 * HO_{21}) + (B1 * OB_{11})) \tag{4.3}$$

The value of the output is used to determine the predictive accuracy of the neural network with the current weights. For example, if a neural network is being used to determine if the XOR of two 2-bit values is true, the input would consist of the four bits, which would progress to the hidden and output layers as described above. If the XOR of the two values is true, and the weights of the neural network are near their optimal values, the output gives some value near 1.0 (i.e., values close to 1.0 correspond to true, and values closer to 0.0 or -1.0 to false). For example, a comparison of 01 and 10 should return a value closer to 1.0 than a comparison of 00 and 10 [17].

4.1.3 The input layer

The input of the neural network used in this thesis contains 35 nodes. 30 of these nodes are used for opponent modeling, while the remaining five are concerned with the current status of the table. Table 4.1 gives an overview of the input features.

Table 4.1: Input features of the evolving neural network

Input Node	Feature	Description
1	Pot	Chips available to win
2	Call	Amount needed to call the bet
3	Opponents	Number of opponents
4	Win Percent	The percentage of hands that will win
5	Position	Number of hands until this player is the dealer
6 to 15	Chips	The chip counts for each player
16 to 25	Total	The overall aggressiveness of each player
26 to 35	Current	The recent aggressiveness of each player

4.1.3.1 The pot

The first input value is the current number of chips in the pot. This value is not always the value of the complete pot, but rather the amount that the current agent would win if it wins the hand. Recall the discussion of a side pot in Section 2.1.1. If the player is only eligible for a side pot, then the pot value is the value of the side pot, not the value of the main pot.

4.1.3.2 The bet

The second input value is the amount of the bet to call. Combined with the pot, the bet amount forms what is known as the *pot odds*, which is a ratio of how much a player stands to win, compared with how much he could lose. For example, if the pot is $2000, and a player needs to bet $500 to call the bet, he is said to have 4:1 (4 to 1) pot odds. Although the pot odds are often used by professional Poker players as an off-hand measure of whether a bet is worth calling, its objective importance is unknown. Rather than calculate the pot odds, and input them to the network, the value of the pot and the value of the amount to call are input separately. It is assumed that the network will eventually determine the correct ratio for the pot-odds, as well as its relative importance. If, for example, agents perform better when the weight of the pot is twice the weight of the amount to call, as opposed to 1:1, then a ratio of 2:1 will be evolved.

4.1.3.3 The opponents

The third value of the input layer is the number of agents currently in the hand. As the number of agents in a hand decreases, the likelihood that an opponent has a rarer hand than the agent also decreases. In general, the odds that one opponent has a higher pair in heads-up play, represented as $\mathrm{P}_{single}(\mathrm{x})$, is given in Equation 4.4, where higher corresponds to the number of ranks greater than the rank of the cards held by the player.

$$P_{single}(x) = \frac{(higher) \times (4)}{50} \times \frac{3}{49} \tag{4.4}$$

The odds that an opponent holds one higher hole card are (4 × higher) in 50, as the card must be a higher rank, and can be one of four suits, and is drawn from 50 remaining cards. The odds that the second hole card of the opponent is the same as

the first are 3 in 49, as there are three remaining cards of the same rank, drawn from 49 remaining cards. For example, if an agent is holding a pair of Jacks, there are 3 higher ranks: Queen, King, and Ace. Since there are four suits, there are 12 possible hole cards (out of a remaining 50 cards) that are higher than the agent's hole cards. If one of these cards is dealt to the opponent, there are three remaining cards of the same rank (out of 49 remaining cards) that can be dealt as the second hole card. The probability of both events occuring is 0.0147.

If there are many opponents, however, the odds that at least one opponent is holding a higher pair before the flop increase. The odds that a higher pair is held by any opponent, represented by $P_{multi}(x)$, are shown in Equation 4.5.

$$P_{multi}(x) = P_{single}(x) \times n - P_{multihigh}(x) \tag{4.5}$$

where n is the number of opponents, and $P_{multihigh}(x)$ is represented as

$$P_{multihigh}(x) = \sum_{i=2}^{n} (n-1) \times P_i, \tag{4.6}$$

where P_i is the probability that exactly i opponents will hold a higher pair, and i goes from 2 to n.

In Equation 4.5, $P_{single}(x)$ is multiplied by the number of opponents, as each opponent has $P_{single}(x)$ odds of having a higher pair. However, these odds include the odds that multiple opponents have higher pairs, and the probability of such an event is effectively added multiple times. To counter this, the probability of numerous opponents holding a higher pair, equal to $P_{multihigh}(x)$, must be subtracted.

For example, if an agent is holding a pair of queens, the value of P_{single}(x) is 0.0098. If there are 2 opponents, P_{single} is multiplied by 2, yielding 0.0196. The odds that exactly two opponents are holding a higher pair are equal to Equation 4.7.

$$P_{actual}(x) = (\frac{8}{50} \times \frac{3}{49}) \times (\frac{4}{48} \times \frac{3}{47} + \frac{2}{48} \times \frac{1}{47}) \tag{4.7}$$

The first opponent has the same odds of having a higher pair as in Equation 4.4. For the second opponent to have a higher pair than the agent, one of two events must occur: either the second opponent has a pair that is equal to the pair of the first opponent, or the second opponent has a pair of cards not equal to the first opponent. In the former case, the second opponent must be dealt one of the two remaining cards out of 48 cards remaining in the deck. He must then be dealt the one

remaining card that will make a pair, out of 47 cards remaining in the deck. In the latter case, he must be dealt one of the four cards that are higher than the agent's, but not of the same rank as the first opponent (recall that the agent is holding a pair of queens, and if the first opponent has a higher pair, only one other rank is left). The second opponent must then be dealt one of the remaining three cards that will make his pair. The resulting odds that exactly two opponents will have a higher pair is 0.00006. Returning to Equation 4.5, the odds that at least one opponent will have a higher pair are 0.0195.

4.1.3.4 The cards

The next value into the network is a combination of *hand strength* and *hand potential*. Hand strength is the current strength of a hand, whereas hand potential represents the ability of a hand to improve as more cards are dealt. For example, prior to the flop, with two agents playing, one has two aces, one of spades and one of clubs, and the other has a jack and a queen, both of hearts. The first agent has a stronger hand, as he already has a pair. Whenever a new card is dealt, there is the possibility that each player's hand will gain or lose some strength. For example, if three hearts are dealt on the flop, the second agent will improve to a flush, while the cards will not help the first agent.

Hand strength can be calculated by determining how many hands the player's hand defeats, given the current cards. For example, prior to the flop, if a player is holding an ace of spades and an ace of hearts, all the other two-card hands can be calculated, and these cards can be given a rough ranking (at this point, the highest hands are pairs, but after the flop, a full evaluation can be completed).

Hand potential can be determined by simulating the dealing of future cards, and determining whether the player's hand maintains its ranking. For example, the player with two aces has the best possible hand prior to the flop, but if the flop consists of a three of diamonds, a five of diamonds, and a five of clubs, then the two aces has several hands that can beat it. A three and a four of a kind are possible, and agents can have large portions of a straight or flush (which might be completed on the turn or river). A ratio of the number of potential hands where an agent's rank increases or remains the same to the number of total hands could give an indication of the potential that a hand has to improve.

Although [7] suggests separate values for both hand strength and hand potential,

the value presented here is a combination of both. Although an agent may be concerned with how its hand is improving, in the end, all that matters is if the agent wins the hand. Lookup tables were generated before the agents played even one game of Poker. Rather than assign weights to strength and potential, the lookup tables concentrated on the likelihood that an agent would win at showdown, given that it kept its cards that long. Such a method will penalize a hand that is currently strong, but grows weaker as more cards are dealt. On the other hand, this method will not differentiate between a strong hand that stays strong, and a weaker hand that has a good chance of improving.

Lookup tables were generated for three stages of the game: the pre-flop, the post-flop, and the post-river. For the post-turn stage of the game, rather than build a separate table, the calculations were done at run-time (i.e., the 46 possible river cards were looped, and the post-river table was used to determine the likelihood that the agent would win). All possible hands were generated, and odds were stored in the lookup tables that indicated the percentage of hands that would be won by the given hole cards if they were allowed to go to a showdown. Although the one-time cost is high, the table provides a fast reference when the game is being played.

The pre-flop table is a 2-dimensional matrix. The first dimension represents the number of potential hands available to the agent pre-flop. The second dimension represents the number of active opponents in a hand (recall that the odds of winning a hand change relative to the number of opponents). There are 169 possible hands that an agent can hold prior to the flop (i.e., 78 unique sets of 2 cards of the same suit, 78 unique sets of 2 cards of different suits, and 13 unique pairs). Although there are many more potential combinations of two cards, at this point, suits are irrelevant. That is, a 9 of clubs and a 9 of spades is equivalent to a 9 of spades and a 9 of hearts. There is a maximum of nine opponents at a table, and thus, the pre-flop table has 1,521 total entries.

The post-flop tables are 3-dimensional matrices. Unlike the pre-flop, where one table was sufficient to handle every case, multiple lookup tables were generated for the post-flop phase The flop has six different potential states. The flop can have three cards of the same suit (called ONE_SUIT), two cards of one suit and one of another, or three cards of different suits. When the flop has two suits, two of the cards can be of the same rank, or all three cards can be of different ranks (both cases are in one table called TWO_SUIT). When the flop has three different suits, none, two, or all three

of the cards can be of the same rank (represented by tables called THREE_SUIT, THREE_SUIT_PAIR, and THREE_SUIT_TRIPLE, respectively). Like the pre-flop tables, two of the dimensions of the table represent the number of hole-card combinations and opponents. The third dimension of the table represents the number of possible flops that are of a given type. For example, there are 13 flops where the three cards are of different suits and all have the same rank. Ignoring suits, there are 286 different flops where all three cards are of the same suit (i.e. C_3^{13}). Table 4.2 summarises the number of entries in the post-flop tables.

The post-river tables are 2-dimensional, with the dimensions representing the number of potential flops, potential hole-card combinations. The separate odds for the different number of opponents was not stored in the post-river tables, but calculated after the odds for one opponent were retrieved. After the river, the community cards can be of one of four types: all five cards are of the same suit (represented by a table called FIVE_SUITED), four cards are of the same suit, and one is of another (represented by a table called FOUR_SUITED), three cards are of one suit and the other two are of other suits (represented by a table called THREE_SUITED), or there are no more than two cards of the same suit (represented by a table called NO_SUITED). Table 4.2 summarises the number of entries in the post-river tables.

Table 4.2: Features of the lookup tables

Stage	Name	Hole Cards	Flops	Opponents	Entries
Pre-flop	PRE_FLOP	169	NA	9	1521
Post-flop	ONE_SUIT	344	286	9	885456
Post-flop	TWO_SUIT	721	1014	9	6579846
Post-flop	THREE_SUIT	1176	286	9	3027024
Post-flop	THREE_SUIT_PAIR	744	156	9	1044576
Post-flop	THREE_SUIT_TRIPLE	378	13	9	44226
Post-river	FIVE_SUITED	223	1287	NA	287001
Post-river	FOUR_SUITED	244	9295	NA	2267980
Post-river	THREE_SUITED	266	26026	NA	6922916
Post-river	NO_SUITED	91	6188	NA	563108

The GENERATE_PRE_FLOP_TABLES function

The procedure shown in Figure 4.2 shows the construction of the pre-flop table. Lines 4 through 15 loop through the possible combinations of hole cards. There are

```
1: procedure GENERATE_PRE_FLOP_TABLES()
2: begin
3: HoleID = 0
4: for Hole1 = TWO to ACE do
5:     for Hole2 = TWO to ACE do
6:         if Hole1 > Hole2 then
7:             Holes[HoleID][0] = Card(Hole2, 0)
8:             Holes[HoleID][1] = Card(Hole1, 0)
9:         else
10:            Holes[HoleID][0] = Card(Hole1, 0)
11:            Holes[HoleID][1] = Card(Hole2, 1)
12:        end if
13:        HoleID = HoleID + 1
14:    end for
15: end for
16: for TableID1 = 0 to 51 do
17:    Table1 = Card(TableID1 /4, TableID1 mod 4)
18:    for TableID2 = TableID1 + 1 to 51 do
19:        Table2 = Card(TableID2 /4, TableID2 mod 4)
20:        for TableID3 = TableID2 + 1 to 51 do
21:            Table3 = Card(TableID3 /4, TableID3 mod 4)
22:            for TableID4 = TableID3 + 1 to 51 do
23:                Table4 = Card(TableID4 /4, TableID4 mod 4)
24:                for TableID5 = TableID4 + 1 to 51 do
25:                    Table5 = Card(TableID5 /4, TableID5 mod 4)
26:                    for HoleID = 0 to NUMBER_PRE_FLOPS − 1 do
27:                        if IsValid(Holes[HoleID][0], Holes[HoleID][1], Table1, Table2, Table3, Table4, Table5)
                           then
28:                            HoleScores[HoleID] = Evaluate(Holes[HoleID][0], Holes[HoleID][1], Table1, Table2,
                               Table3, Table4, Table5)
29:                        else
30:                            HoleScores[HoleID] = −1
31:                        end if
32:                    end for
33:                    for TestHoleID1 = 0 to 51 do
34:                        if TestHoleID1 == TableID1∨TestHoleID1 == TableID2∨TestHoleID1 == TableID3∨
                           TestHoleID1 == TableID3 ∨ TestHoleID1 == TableID4 ∨ TestHoleID1 == TableID5
                           then
35:                            continue
36:                        else
37:                            TestHole1 = Card(TestHoleID1 /4, TestHoleID1 mod 4)
38:                        end if
39:                        for TestHoleID2 = TestHoleID1 + 1 to 51 do
40:                            if TestHoleID2 == TableID1 ∨ TestHoleID2 == TableID2 ∨ TestHoleID2 ==
                               TableID3 ∨ TestHoleID2 == TableID3 ∨ TestHoleID2 == TableID4 ∨
                               TestHoleID2 == TableID5 then
41:                                continue
42:                            else
43:                                TestHole2 = Card(TestHoleID2 /4, TestHoleID2 mod 4)
44:                            end if
45:                            TestScore = Evaluate(TestHole1, TestHole2, Table1, Table2, Table3, Table4, Table5)
46:                            for i = 0 to HOLES_PRE_FLOP − 1 do
47:                                if HoleScores[i] == −1 ∨ Holes[i][0] == TestHole1 ∨ Holes[i][1] == TestHole1 ∨
                                   Holes[i][0] == TestHole2 ∨ Holes[i][1] == TestHole2 then
48:                                    continue
49:                                end if
50:                                Determine_Hand_Ranks(TestScore, HoleScores[i], Greater, Lesser, Equal, i)
51:                            end for
52:                        end for
53:                    end for
54:                    Calculate_Odds(HoleOdds[][], Lesser, Equal, Greater)
55:                end for
56:            end for
57:        end for
58:    end for
59: end for
60: Normalise(HoleOdds[][])
61: end GENERATE_PRE_FLOP_TABLES
```

Figure 4.2: The procedure to build the pre-flop tables

```
1:  procedure DETERMINE_HAND_RANKS_(DummyScore, RealScore, Greater[], Lesser[], Equal[], index)
2:  begin
3:  if RealScore < DummyScore then
4:      Greater[index] = Greater[index] + 1
5:  else if RealScore > DummyScore then
6:      Lesser[index] = Lesser[index] + 1
7:  else
8:      Equal[index] = Equal[index] + 1
9:  end if
10: end DETERMINE_HAND_RANKS
```

Figure 4.3: The procedure to determine the rank of the current hand

two different cases for the combinations of hole cards. Either the cards are of the same suit, or they are of different suits. The actual suit of the cards is irrelevant; all that matters is that the suits are the same, or different. Lines 6 through 8 consider the case where the cards are of different suits (designated suit 0). Lines 9 through 11 consider the case where the cards are of the same suit (suits 0 and 1). Each combination of hole cards is given a unique ID in line 13.

Line 16 begins a loop through the possible community cards. The loops allow 52 different card values. The loops allow no duplicates of community cards (as each loop begins on the next value of the previous loop). By dividing the card value by 4, the rank of the card is obtained, and by taking the modulus with base 4, the suit is obtained. The loop from lines 26 to 32 evaluates the score of the hand. If any of the community cards is a duplicate of one of the hole cards, a score of -1 is given to the hand (line 30). Otherwise, hands are scored according to a hand evaluation function based upon the one used by the commercial Poker program Poker Academy [24], and stored according to the ID of their hole cards.

Lines 33 through 52 determine the overall rank of hole card combinations given the current community cards. All possible combinations of hole cards are generated, and the hand with the current hole card combination and community cards is evaluated (line 45), and stored in TestScore. The Determine_Hand_Ranks function compares TestScore to each HoleScore, and is shown in Figure 4.3. If TestScore is greater than the HoleScore at the current HoleID, the number of hands that evaluate greater than the HoleScore is incremented (line 4 of Figure 4.3). Similarly, the number of hands that evaluate less than the HoleScore (line 6), or equal to the HoleScore (line 8), are incremented. In Figure 4.2 line 54 then calculates the odds that a particular hand will win by going to showdown. The function Calculate_Odds simply sums the number of hands that are winning and tied, and divides by the total number of hands to get the odds that the hand will win against one opponent. It then calculates the odds

that the hand will win against multiple opponents. The function Normalise (line 60) normalises the odds by the number of combinations of community cards (the earlier odds were calculated given that a particular set of community cards was dealt),

The GENERATE_FLOP_TABLES function

To build the flop tables, a similar strategy is used after the flop as before the flop, as shown in Figure 4.4. There are five forms that the flop can take when it is dealt. Line 3 represents the case where the flop has three cards, all of the same suit. There are 286 such flops Line 4 represents a flop with cards belonging to two different suits. Line 5 represents the flop with cards of three different suits, and three different ranks. Line 6 shows the flop that has cards of three different suits, but with two cards of the same rank. Line 7 show the flop that has cards of three different suits, but where all the cards have the same rank. The case where the flop has two suits, but also has a pair, is covered in the case in line 4.

```
1: procedure GENERATE_FLOP_TABLES()
2: begin
3: BUILD_1SUIT()
4: BUILD_2SUIT()
5: BUILD_3SUIT()
6: BUILD_3SUITPAIR()
7: BUILD_3SUITTRIPLE()
8: end GENERATE_FLOP_TABLES
```

Figure 4.4: The procedure to build the flop tables

The BUILD_1_SUIT function

The formation of the flop table is similar for the different forms of flops. For simplicity, only procedure BUILD_1_SUIT for building the section of the flop table consisting of flops of a single suit will be demonstrated. The procedure for generating this section of the flop table is shown in Figure 4.5.

The procedure shown in Figure 4.5 assumes that all of the cards in the flop are of the same suit. The suit itself is irrelevant, as suits have no value in Texas Hold'em. Line 3 initialises the FlopId to 0. Each flop will have a unique id. Lines 4 through 9 loop through the possible cards that might be in the flop, beginning with the first card of the flop in line 4, and proceeding to the third card in line 8.

Beginning at line 11, a loop begins that sets up all combinations of hole cards that it is possible for the player to have. There are three situations for the hole cards

```
 1: procedure BUILD_1_SUIT()
 2: begin
 3: FlopID = 0
 4: for i = TWO to ACE do
 5:     Flop[0] = Card(i, 0)
 6:     for j = i + 1 to ACE do
 7:         Flop[1] = Card(j, 0)
 8:         for k = j + 1 to ACE do
 9:             Flop[2] = Card(k, 0)
10:             HoleID = 0
11:             for m = TWO to ACE × 2 do
12:                 if m == Flop[0] ∨ m == Flop[1] ∨ m == Flop[2] then
13:                     continue
14:                 end if
15:                 for n = m + 1 to ACE × 2 do
16:                     if n == Flop[0] ∨ n == Flop[1] ∨ n == Flop[2] then
17:                         continue
18:                     end if
19:                     if m < ACE then
20:                         Hole[HoleID][0] = Card(m, 0)
21:                     else
22:                         Hole[HoleID][0] = Card(m, 1)
23:                     end if
24:                     if n < ACE then
25:                         Hole[HoleID][1] = Card(n, 0)
26:                     else
27:                         Hole[HoleID][1] = Card(n, 1)
28:                     end if
29:                     HoleID = HoleId + 1
30:                 end for
31:             end for
32:             for m = TWO to ACE do
33:                 for n = TWO to ACE do
34:                     Hole[HoleID][0] = Card(m, 0)
35:                     Hole[HoleID][1] = Card(n, 2)
36:                     HoleID = HoleId + 1
37:                 end for
38:             end for
39:             BUILD_ROW(Table1Suit[FlopID], Hole, HoleId, Flop)
40:             FlopID = FlopID + 1
41:         end for
42:     end for
43: end for
44: end BUILD_1_SUIT
```

Figure 4.5: The procedure to build the rows of the flop table for one-suit flops

when there is a flop that is all of one suit. First, the player could have two cards that are of the same suit as the flop. Second, the player could have one card that is the same suit, and one that is different. Finally, the player could have two cards that are different suits, and are also different from the suit of the flop.

Lines 11 through 31 cover the first two cases, while lines 32 to 38 cover the third case. In line 11, the first hole card loops twice the number of cards in a suit. This loop ensures that the first hole card has the opportunity to be every card from either the flop suit, or some second suit. If the card repeats one of the cards in the flop, it is ignored (line 12). The order of the hole cards does not matter (i.e., a two of spades and a four of diamonds is the same as a four of diamonds and a two of spades). The

second hole card begins looping one card after the first hole card (i.e., if the first hole card is the seven of the flop suit, the second hole card begins looping at the eight. If the first hole card is the ace of the flop suit, the second hole card begins looping at the two of a non-flop suit). Similarly to the first hole card, if a flop card is repeated, the card is ignored (line 16).

Lines 19 through 28 assign the values to the hole cards. The loop goes from two to ace twice. The first loop from two to ace considers the first suit, while the second loop considers the second suit. Lines 32 through 38 cover cases where a player has two hole cards of different suits, and the suits are also different from the flop. Lines 32 and 33 loop through the possible hole card combinations, and assign the cards in lines 34 and 35. Line 39 calls the BUILD_ROW function, which is shown in Figure 4.6. The items passed to the function are the appropriate row to construct in the lookup table, denoted by the FlopId, as well as the set of HoleCard combinations, and the number of HoleCard Combinations, denoted by the current HoleID. The cards of the Flop are also passed to the function.

The BUILD_ROW function

Figure 4.6 builds a row of the flop table. Each row in the flop table corresponds to a single combination of cards for the flop, while columns correspond to combinations of hole cards. Our implementation considers the flop of a two, a three, and a four of the same suit to be flop number 1, as denoted in Figure 4.5 lines 4 through 9. The first combination of hole cards would be the smallest combination of hole cards that does not interfere with the flop. The value stored at location (0, 0) of the flop table, for example, corresponds to the likelihood that the player's hand will win, given that the flop is a two, a three, and a four of the same suit, and the player holds a five and a six of the same suit.

Figure 4.6 uses the flop generated in Figure 4.5 to build the appropriate row of the flop table. Lines 3 through 11 loop through the remaining cards in the deck, and generate potential turn and river cards for the flop. Since the cards are looping from 0 to 52, it is necessary to convert the value into a card from *TWO* to *ACE*, and to a suit. Since suits are irrelevant, they will only be considered as 0 through 3. Dividing the current card by four will give the rank of the card, with 0 representing *TWO*, 1 representing *THREE*, and so on, with 12 representing *ACE*. Likewise, taking the modulus 4 of the card will give the suit, with 0 representing the first suit, 1 the second

```
1:  procedure BUILD_ROW(Row[][MAX_OPPONENTS], Holes[][2], HoleCount, Flop[])
2:  begin
3:  for i = 0 to 51 do
4:      Turn = Card(i / 4, i mod 4)
5:      if Turn == Flop[0] ∨ Turn == Flop[1] ∨ Turn == Flop[2] then
6:          continue
7:      end if
8:      for j = i + 1 to 51 do
9:          River = Card(j / 4, j mod 4)
10:         if River == Flop[0] ∨ River == Flop[1] ∨ River == Flop[2] then
11:             continue
12:         end if
13:         for k = 0 to HoleCount − 1 do
14:             if Holes[k][0] == Turn ∨ Holes[k][0] == River ∨ Holes[k][1] == Turn ∨ Holes[k][1] == River then
15:                 HoleScores[k] = −1
16:             else
17:                 HoleScores[k] = EVALUATE(Holes[k][0], Holes[k][1], Flop[0], Flop[1], Flop[2], Turn, River)
18:             end if
19:         end for
20:         for TestHoleID1 = 0 to 51 do
21:             TestHole1 = Card(TestHoleID1 / 4, TestHoleID1 mod 4)
22:             if TestHole1 == Flop[0] ∨ TestHole1 == Flop[1] ∨ TestHole1 == Flop[2] ∨ TestHole1 == Turn ∨
                TestHole1 == River then
23:                 continue
24:             end if
25:             for TestHoleID2 = 0 to 51 do
26:                 TestHole2 = Card(TestHoleID2 / 4, TestHoleID2 mod 4)
27:                 if TestHole2 == Flop[0] ∨ TestHole2 == Flop[1] ∨ TestHole2 == Flop[2] ∨ TestHole2 ==
                    Turn ∨ TestHole2 == River then
28:                     continue
29:                 end if
30:                 TestScore = Evaluate(TestHole1, TestHole2, Flop[0], Flop[1], Flop[2], Turn, River)
31:                 for k = 0 to HoleCount − 1 do
32:                     if HoleScores[k] == −1 ∨ Holes[k][0] == TestHole1 ∨ Holes[k][1] == TestHole1 ∨
                        Holes[k][0] == TestHole2 ∨ Holes[k][1] == TestHole2 then
33:                         continue
34:                     end if
35:                     Determine_Hand_Ranks(TestScore, HoleScores[i], Lesser, Equal, Greater)
36:                 end for
37:             end for
38:         end for
39:         end BUILD_ROW
40:         for k = 0 to HoleCount − 1 do
41:             if HoleScores[k] == −1 then
42:                 continue
43:             end if
44:             Calculate_Odds(Row[k][], Lesser, Equal, Greater)
45:             Normalise(Row[k][])
46:         end for
47:     end for
48: end for
```

Figure 4.6: The procedure to build a row of the flop table

suit, and so on, with 3 representing the fourth suit.

For each potential set of five community cards, each combination of hole cards is evaluated (lines 13 to 19). A Poker evaluation tool, here called EVALUATE, is based upon the evaluator used in the commercial Poker program Poker Academy [24]. EVALUATE generates values corresponding to the quality of a hand (i.e., better hands will have better scores). These scores are stored in HoleScores, indexed by the current combination of HoleCards (line 17). If the current combination is invalid, due to a repetition of cards, the value stored in HoleScores is -1 (line 15).

The quality of a hand is determined by how many other hands it can defeat. If the flop, turn, and river remain constant, the only cards that will affect the likelihood that a hand will win are the hole cards. The loops in line 20 and 25 test all of the other potential combinations of hole cards (i.e., TestHole1 and TestHole2). Line 30 evaluates the hand with the alternate hole cards, and stores the value in TestScore. Lines 31 through 36 compare TestScore to HoleScores[k] using the same Determine_Hand_Ranks function as the pre-flop table. Line 43 through calculates the odds that an agent will win against a given number of opponents. Line 44 Normalises the odds.

The functions to build rows in the flop table for flops of more than one suit are very similar to the BUILD_1_SUIT function. The only difference occurs in lines 5 through 9, where the cards in the flop do not need to be of the same suit. Similarly, lookup tables are constructed for the post-river phase of the hand. For the post-turn phase, rather than build and consult a lookup table, the evaluator simply loops through the possible river cards, similar to lines 8 through 12 of Figure 4.6.

4.1.3.5 The position

After the hand strength, the position of the player is input into the network. The position variable is determined as the number of hands remaining until the current player receives the dealer's button. At a table with ten players, if the button is at seat number seven, and the player is sitting at seat number three, then the position is input as six. On the next hand, the button will move to position eight, and then nine, ten, one, and two, before moving to three.

Position was determined in this manner to take advantage of how late in a hand a player bets. It is desirable to be one of the last players to bet, because the last players have seen the betting decisions of all the previous bettors, and thus have more

information. The value of position will be high if a player is betting late, and low if a player is betting early.

4.1.3.6 The chips

The next 10 inputs to the neural network are the chip counts of the agents playing at the same table as the agent who is making the betting decision. The chip counts are a representation of the number of chips that any agent has, and are stored in a vector of size 10. Chip vectors are always relative to the respective agent's position at the table. For example, at a table, there are five players: Bill, with $500, Judy, with $350, Sam, with $60, Jane, with $720, and Joe, with $220. If Sam is making a betting decision, then his chip vector will look like Table 4.3.

Table 4.3: The vector for Sam's knowledge of opponent chip counts

Player	Distance	Chips
Sam	0	$60
Jane	1	$720
Joe	2	$220
Bill	3	$500
Judy	4	$350
None	5	$0
None	6	$0
None	7	$0
None	8	$0
None	9	$0

This representation is independent of the seat at which the agent is sitting. When an agent is making a betting decision, the input at position 6 of the input vector (i.e., position 0 of the chip vector) will always be its respective chip count. Likewise, regardless of which seat an agent is sitting at, the input at position 7 of the input vector will be the chip count of the agent in the next seat, while the input at position 15 will be the chip count of the player in the previous seat. Furthermore, when a seat is empty, the chip count of that seat is 0. In such a case, the output from that input node will be zero, and the result will be the same as if that node had not been included in the result.

4.1.3.7 Opponent aggressiveness

The final twenty inputs to the neural network are reserved for opponent modeling. The only information that an agent knows about its opponents is what it can directly observe, which in No-Limit Texas Hold'em, is limited to its opponents' betting decisions. As mentioned in Chapter 3, bets are often considered in terms of units called small bets. However, simply considering the size of the bet often ignores the intention of the better. For example, it is worthwhile to distinguish a raise of the bet from \$5 to \$25 and a call of \$25 by another player. A raise is almost always more aggressive than a call.

A call-based aggressiveness factor tries to take this consideration into account. Whenever any agent makes a bet, their action is recorded according to Equation 4.8.

$$AggressivenessWeight = \frac{BetAmount}{CallAmount} \tag{4.8}$$

The call amount is the amount of the bet that must be made by the agent to call the bet, and the bet amount is the amount that is actually bet. Thus, a fold, with a bet of 0, receives an aggressiveness weight of 0, a call receives a value of 1, and a raise receives a value greater than 1. Using the previous example, the player that raises the bet from \$5 to \$25 will receive an aggressiveness weight of 5.0, whereas the player that calls a previous bet of \$25 will receive a weight of 1.0. The unique case of a check, where a call of 0 is required, is given a weight of 0, and is interpreted as approximately as aggressive as a fold.

4.1.3.8 Aggressiveness over the long-term

As agents make betting decisions, all of the other agents at the table observe the decisions and record the actions made. More precisely, a running average of the betting decisions of each agent at the table is kept, and can be accessed by any of the agents when they are making a betting decision. Like the chip counts, the first aggressiveness that is observed is an agent's respective aggressiveness. Second is the opponent immediately after the agent, and so on, until the last aggressiveness observed is that of the opponent immediately before the agent. Returning to the previous example, Sam's aggressiveness vector could look like Table 4.4.

The aggressiveness values are a running average of the decisions made by the various agents. For example, on the next hand, Sam is first to bet. He calls the blind. Jane folds, as does Joe. Bill raises the bet to three times what it was, Judy calls,

Table 4.4: The vector for Sam's knowledge of opponent aggressiveness

Player	Distance	Aggressiveness	Decisions
Sam	0	1.3	55
Jane	1	0.5	38
Joe	2	0.7	35
Bill	3	2.4	67
Judy	4	1.5	64
None	5	0	0
None	6	0	0
None	7	0	0
None	8	0	0
None	9	0	0

and Sam folds. On this betting round, Sam made two decisions, Joe and Jane each made one, Bill made one, and Judy made one. Sam has an average aggressiveness of 1.3, meaning that over 55 decisions, he has a cumulative value of 71.5 (i.e., 1.3×55). Since he called once (a value of 1.0), and folded once (a score of 0.0), his new cumulative value is 72.5, over 57 decisions, for a new average aggressiveness of 1.27. Likewise, the other agents' updated aggressiveness values are shown in Table 4.5.

Table 4.5: The updated vector for Sam's knowledge of opponent aggressiveness

Player	Distance	Aggressiveness	Decisions
Sam	0	1.27	57
Jane	1	0.49	39
Joe	2	0.68	36
Bill	3	2.41	68
Judy	4	1.49	65
None	5	0	0
None	6	0	0
None	7	0	0
None	8	0	0
None	9	0	0

With the exception of Bill, all of the agents' aggressiveness decreased slightly. The goal of the overall aggressiveness is for the agents to try and observe any tendencies that their opponents might have, and be able to take advantage of them. Furthermore, the agents keep track of their own aggressiveness, so that they can occasionally make

decisions that fall outside of their standard playing styles, in an effort to disguise any tendencies that can be exploited.

4.1.3.9 Aggressiveness over the short-term

Short-term aggressiveness, referred to hereafter as current aggressiveness, concerns only the betting decisions made by an agent in the recent past. Instead of keeping a running average of an opponent's aggressiveness values for every hand, only the last ten hands are considered. Ten hands is one complete circuit of the table (for consistency, the number is not reduced when there are fewer agents at the table). Rather than show the current aggressiveness vector for all of Sam's opponents, Table 4.6 only shows Sam's vector for one of his opponents, in this case, Judy.

Table 4.6: The vector for Sam's knowledge of Judy's aggressiveness

Hand Number	Decisions	Cumulative Aggressiveness
0	1	0.0
1	2	1.0
2	4	3.5
3	2	1.0
4	3	2.0
5	1	0.0
6	6	5.0
7	1	0.0
8	3	3.0
9	2	1.0

In Table 4.6, Judy has made 25 betting decisions over the last 10 hands. She has a cumulative aggressiveness value of 16.5 over these 25 betting decisions, for an average aggressiveness value of 0.66 for the last 10 hands. Although her overall aggressiveness is 1.49, as shown in Table 4.5, over the last ten hands, she has been playing differently from her normal style of play.

Rather than consider the current hand to be hand 0, and do a lot of unnecessary shifting of the data in the vector after every hand, a pointer to the current hand is kept. For example, the current hand is hand 7, and it was observed that Judy raised the bet to twice its value, followed by two calls, and another raise to three times the value of the bet. Judy made four betting decisions, with a cumulative aggressiveness value of 7.0. The updated aggressiveness vector for Judy would look like Table 4.7.

Table 4.7: The updated vector for Sam's knowledge of Judy's aggressiveness

Hand Number	Decisions	Cumulative Aggressiveness
0	1	0.0
1	2	1.0
2	4	3.5
3	2	1.0
4	3	2.0
5	1	0.0
6	6	5.0
7	4	7.0
8	3	3.0
9	2	1.0

Judy's updated cumulative current aggressiveness value is 23.5 over 28 betting decisions, for an average current aggressiveness value of 0.84. Sam keeps track of the current aggressiveness of each of the agents at the table in the same manner as the overall aggressiveness. To simplify use of the current aggressiveness information, the total number of decisions and aggressiveness for the last ten hands are kept as well as the individual actions. Table 4.8 shows what Sam knows about his opponents.

Table 4.8: Sam's knowledge of his opponents' aggressiveness

Player	Decisions	Current Aggressiveness	Individual vector
Sam	22	1.32	Sam's vector
Jane	12	0.97	Jane's vector
Joe	27	0.55	Joe's vector
Bill	19	2.67	Bill's vector
Judy	21	0.84	Judy's vector
None	0	0.0	None
None	0	0.0	None
None	0	0.0	None
None	0	0.0	None
None	0	0.0	None

Recall that Bill is at the first seat, followed by Judy, etc., so James is seated after Joe. In Sam's current aggressiveness vector, however, placing is relative to Sam, so James is in position 4. The current aggressiveness vector has only one value: the current aggressiveness value for the previous ten hands. The individual agent vectors

are stored separately, and are only used for the calculation of the aggressiveness value.

4.1.4 The hidden layer

It is assumed that one hidden layer in the neural network will be sufficient to find a non-linear solution [17]. The hidden layer of the neural network consists of 20 nodes fully connected to both the input and output layers, as well as an extra bias node on the input layer and hidden layer. Twenty nodes was chosen arbitrarily early in the implementation and never changed. It is assumed that 20 hidden nodes will be sufficient. The values in the hidden layer are sent to the output layer using Equation 4.3.

4.1.5 The output layer

The output of the neural network consists of a vector of five values. Each value corresponds to a recommended betting decision for the decision-making agent. The first two values correspond to a fold and a call, respectively, while the final three correspond to a raise. Whereas the size of bets in Limit Texas Hold'em is fairly static, No-Limit Texas Hold'em allows bets in a range from a minimum bet to all of an agent's chips.

The raise decision was split into three sub-decisions, namely a small raise, a medium raise, and a large raise. Although small, medium, and large are rather subjective terms, we have defined them identically for all agents. After observing many games of television and online No-Limit Texas Hold'em, it was determined that the biggest decider of *small*, *medium*, and *large* bets was a player's chip count. Bets that were less than 10% of a player's chips were generally considered small. Bets that were more than one third of a player's chips were considered large, and bets somewhere in between were considered medium bets. It was also observed that when a player made what was considered a large bet, the bet consisted of all of his chips nearly 10% of the time.

To encourage bluffing and sandbagging, agents' bets were not restricted to a particular range, but were distributed normally according to a pre-determined standard deviation. It is entirely possible that a player that should make a small bet will bet all of his chips in a bluff, and it is also possible that a player that should make a large bet will make the minimum bet. Watching television and online Poker games,

```
1: procedure GET_BET_AMOUNT(BetType, MaxBet, MinBet)
2: begin
3: FinalBet = 0
4: if MaxBet < MinBet then
5:     returnMaxBet
6: end if
7: Percentile = Random.UniformDouble()
8: if BetType == SMALL then
9:     if Percentile < LoInside then
10:        Percentile = Random.UniformDouble() × LoUpper
11:    else
12:        Percentile = LoUpper + Abs(Random.Gaussian(LoSigma))
13:    end if
14: else if BetType == MEDIUM then
15:    if Percentile < MedInside then
16:        Percentile = MedLower + (Random.Double() × MedWide)
17:    else if Percentile < MedInOrLo then
18:        Percentile = MedLower − Abs(Random.Gaussian(MedSigmaLo))
19:    else
20:        Percentile = MedUpper + Abs(Random.Gaussian(MedSigmaHi))
21:    end if
22: else if BetType == LARGE then
23:    if Percentile < HighAbove then
24:        Percentile = HiPoint + Abs(Random.Gaussian(HiSigmaHi))
25:    else
26:        Percentile = HiPoint − Abs(Random.Gaussian(HiSigmaLo))
27:    end if
28: end if
29: FinalBet = Percentile × MaxBet
30: if FinalBet < MinBet then
31:    FinalBet = MinBet
32: else if FinalBet > MaxBet then
33:    FinalBet = MaxBet
34: end if
35: return FinalBet
36: end GET_BET_AMOUNT
```

Figure 4.7: The procedure to choose the amount to bet

it was determined that generally, the more confident a player is that his cards are high ranking, the more likely he is to stray from his recommended bet level. In other words, a player is more likely to bluff or sandbag when he is committed to making a large bet than when he is committed to making a small bet. The bet selection algorithm is presented Figure 4.7.

Figure 4.7 demonstrates how an agent decides the amount that it will bet. Table 4.9 lists the various constants that are used in Figure 4.7. BetType distinguishes whether the neural network analysis determined the bet should be small, medium or large. MaxBet is the maximum amount that an agent can bet, and is usually equal to the agent's chip count. The only case where the two are not equal is when the agent has more chips than any other active agent, and thus the maximum that can be bet is the chip count of the opponent with the most chips. MinBet is the minimum amount that must be bet in a betting round, and is usually equal to the amount that must be called. When there is no bet to call, the MinBet is equal to the big blind.

Table 4.9: The constants used in GET_BET_AMOUNT

Name	Value	Description
LoUpper	0.06	Upper boundary for SmallBet range
LoInside	0.7	Likelihood that SmallBet will be inside SmallBet range
MedLower	0.1	Lower boundary for MedBet range
MedUpper	0.2	Upper boundary for MedBet range
MedInside	0.6	Likelihood that MedBet will be inside MedBet range
MedOutLo	0.1	Likelihood that MedBet will be less than MedLower
MedOutHi	0.3	Likelihood that MedBet will be greater than MedUpper
HiPoint	0.3	Point which represents "standard" LargeBet
HiAbove	0.95	Likelihood that LargeBet will be greater than HiPoint
HiBelow	0.05	Likelihood that LargeBet will be less than HiPoint
HiAllIn	0.1	Likelihood that a LargeBet will be an all-in

Lines 4 and 5 determine if the agent has enough chips to make the minimum bet. If the agent has fewer chips than the minimum, the MaxBet is returned. Line 7 obtains a uniform, random real number, to compare to the likelihoods that various situations arise. Lines 8 through 13 cover the case of a small bet. Line 9 tests whether the bet should be a small bet. If so, it was decided that it would fall within the standard range 70% of the time. If the percentile is less than 0.7 (Line 9), then the bet is a standard bet. The standard range for the small bet is from 0 to 6% of an agent's chips, and thus a random percentage in that range is chosen from a uniformly random distribution. If the percentile indicates that a non-standard bet is desirable, then a value from in the range of 6% to 100% of an agent's chips is chosen, from a normally random distribution (line 12). The mean of the normal distribution is 6%, and the standard deviation is computed using Equation 4.9, which allows the curve for the standard and non-standard bets to be continuous. In Equation 4.9, LoOutside is the probability of a small bet being non-standard, and is equal to 0.3. LoInside is the probability of a small bet being standard, and equals 0.7.

$$\frac{2.0 \times LoOutside}{LoInside \times \sqrt{\pi} \times \sqrt{2.0}} \tag{4.9}$$

.

Lines 14 through 21 cover the case of a medium bet. The range for a standard medium bet was decided to be from 10% to 20% of an agent's chips. It was decided that a standard bet would occur 60% of the time, with bets being less than 10% of

an agent's chips 10% of the time, and more than 20% of an agent's chips 30% of the time. Line 14 covers the case of a standard medium bet. If the bet is standard, then it is chosen within the standard range from a uniformly random distribution (lines 15 and 16). If it should be a smaller bet, it is chosen from the percentages of an agent's chips that fall below the minimum of the standard range, from a normally random distribution (lines 17 and 18) with a standard deviation calculated with Equation 4.10. Likewise, if it should be a larger bet, it is chosen from a normally random distribution of the percentages above the standard range (lines 19 and 20) with a standard deviation calculated using Equation 4.11. In Equations 4.10 and 4.11, MedWide is the width of the standard range, and is equal to 0.1. MedInside is the probability that a medium bet is in the standard range, and equals 0.6. MedOutsideLo is the probability that if a medium bet is not in the standard range, it is lower than the minimum of the standard range, and is equal to 0.1. MedOutsideHi is the probability that the bet is higher than the standard range, and equals 0.3.

$$\frac{2.0 \times MedWide \times MedOutsideLo}{MedInside \times \sqrt{\pi} \times \sqrt{2.0}} \tag{4.10}$$

$$\frac{2.0 \times MedWide \times MedOutsideHi}{MedInside \times \sqrt{\pi} \times \sqrt{2.0}} \tag{4.11}$$

Lines 22 through 27 cover the case of a large bet. Unlike the small and medium bets, there is no standard range for a large bet. Rather, there is a point, namely 30% of an agent's chips, at which two normal distributions meet. Lines 23 and 24 determine whether the bet should be above 30% of an agent's chips. This will occur 95% of the time. In this case, the bet is chosen from a normally random distribution with a mean of 30%, and a standard deviation that was chosen such that the function would be continuous with the distribution on the other side of the mean. This standard deviation also forces an all-in to occur in 10% of all large bets. Lines 25 and 26 cover the case where a bet of less than 30% of an agent's chips is made. The bet is chosen from a different normally random distribution with a mean of 30% and a standard distribution that would allow the function to be continuous. Lines 30 and 31 ensure that the bet is at least as much as the minimum bet. Lines 32 and 33 force any large bets down to the maximum bet. The entire betting distribution can be seen in Figure 4.8.

The distributions in Figure 4.8 represent the likelihood that a percentage of the agent's chips will be chosen for a bet, given that the bet is small, medium, or large.

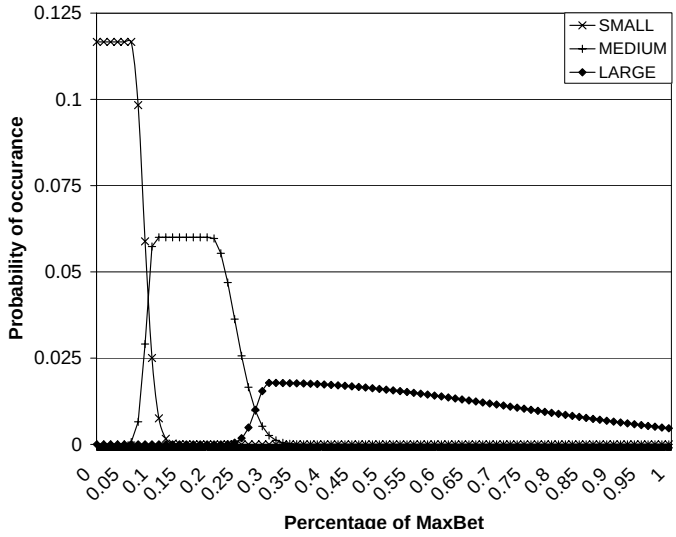

Figure 4.8: The distribution of small, medium, and large bets (original in colour)

The small betting distribution is uniform from 0% of the agent's chips to 6%. It then follows a normal distribution to 100%. A "small" bet of all of an agent's chips is very unlikely, but possible. The medium bet is uniform from 10% to 20% of an agent's chips. At less than 10%, the distribution is normal, with a very high slope. If the bet is higher 20% of an agent's chips, the distribution is also normal, but with a lower slope. For large bets, if the bet is less than 30% of an agent's chips, the distribution is normal with a very high slope, to ensure that most bets are near the 30% value. If the bets are above 30% of an agent's chips, the distribution is still normal, but with a much lower slope, allowing bets up to all of the agent's chips. The standard deviation of this normal distribution was calculated such that all-ins will occur 10% of the time that an agent makes a large bet.

Whenever a betting decision is required, the features in Table 4.1 are input into an agent's neural network, and the vector of betting decisions is output. Agents compete in a number of tournaments, after which they are ranked according to their average standing in the tournament. The agents that played the best in the tournaments are chosen for the evolution phase of the algorithm.

4.1.6 The evolution phase

Evolutionary algorithms attempt to model the biological process of evolution. Agents compete against each other, and the agents that perform their task the best are allowed to continue to compete, while the other agents are removed from the population of agents. Furthermore, new agents are created as slight variations of the successful agents, in an attempt to create increasingly good agents. The entire process is shown in Figures 4.9 and 4.10.

```
 1: procedure EVOLUTION(Generations, NumPlayers, NumPlayersKept, Tournaments)
 2: begin
 3: for i = 0 to NumPlayers − 1 do
 4:     Players[i] = newPlayer(Random)
 5: end for
 6: for i = 0 to Generations − 1 do
 7:     for j = 0 to Tournaments − 1 do
 8:         PlayTournament()
 9:     end for
10:     SortPlayers(Players)
11:     for j = 0 to NumPlayersKept − 1 do
12:         KeptPlayers[j] = Players[j]
13:     end for
14:     EVOLVE_PLAYERS(Players, KeptPlayers, NumPlayers, NumPlayersKept)
15: end for
16: end EVOLUTION
```

Figure 4.9: The procedure to evolve agents

The competition and selection phase is shown in Figure 4.9. This algorithm demonstrates how the best agents are chosen at any given generation. Lines 3 and 4 initialize the population of agents that will be competing in the tournament. The population is initially random. In the context of the neural networks described in Section 4.1.2, random neural networks have all of their weights initialized to random values. The distribution of the random values of the weights is normal, with a mean of 0, and a standard deviation of 0.5.

Lines 6 through 15 loop through the number of generations specified beforehand. A generation can be split into three phases: competition, selection, and evolution. The first phase, selection, is implemented in lines 7 through 9 of Figure 4.9. A number of tournaments are played, during which each agent's rankings are collected. Rankings are determined by how long an agent lasts in a tournament. For example, if there are 500 agents in a tournament, the first agent to be eliminated will receive a ranking of 500th for that tournament. The next agent to be eliminated will receive a ranking of 499th, and so on, until the agent that wins the tournament receives a ranking of 1st. All agents begin a tournament with $1000 in chips, and are considered eliminated when they no longer have any chips. In the case of the winning agent, it is eliminated

in 1st place when there are no longer any opponents left in the tournament. If two or more agents are eliminated at the same time, they are given rankings according to how many chips they had when they were eliminated. For example, if one agent had $500 in chips, and another had $200 in chips, and both were eliminated on the same hand, the former would rank higher than the latter.

As tournaments are played, rankings accumulate. Line 10 sorts the agents according to their cumulative rankings. An agent may win some tournaments, but place terribly in others, while another agent may place highly in all tournaments, without winning. Our algorithm considers the second agent the better agent.

After the agents have been sorted according to their cumulative ranking, Figure 4.9 proceeds to the selection phase of the evolution. In the selection phase, it is decided which agents will be allowed to continue, and which will be discarded. Lines 11 and 12 keep the agents that had the lowest (i.e., the best) cumulative rankings. Line 14 then passes control to Figure 4.10, which will create new agents that are slight modifications of the selected agents.

Figure 4.10 describes the process of creating agents that have new neural networks. The new agents will be similar to the best agents from the previous generation, with slight changes. The population of agents is sorted such that the first agents in the array are the agents that were kept from the previous generation. Line 3 of Figure 4.10 loops through the spots in the array that will hold the new agents. Line 5 determines if only one agent was kept from the previous generation (i.e., a champion format). If so, then all of the new agents will be created from that one agent (line 6). Line 7 determines how many of the previous generation's agents will be used as parents for the new agent. The number of parents is chosen from an exponential distribution, such that two parents is the most likely outcome. One parent is possible, as are three or more, up to the number of agents kept from the previous generation. If only one agent is chosen for a parent, then a random agent is chosen as the parent (line 11). If the number of parents chosen is greater than the available agents, then the number of parents is reduced to the maximum (line 13). As the number of parents increases, it becomes less likely that the child will be similar to any of the individual parents, and thus desirable skills are less likely to be passed on. On the contrary, if only one parent is chosen, the child will be exactly like its parent, and no exploration will occur.

The array of old agents is copied into a dummy array in lines 15 through 17. Manipulations will be made on the agents in the array, and we do not want to alter

```
1:  procedure EVOLVE_PLAYERS(Players[], Elite[], NumPlayers, NumPlayersKept)
2:  begin
3:  ParentCount = 1
4:  for i = NumPlayersKept to NumPlayers do
5:      if numPlayersKept == 1 then
6:          Players[i] = newPlayer(Elite[0])
7:      else
8:          ParentCount = Random.Exponential()
9:          Parents = newNeuralNet[ParentCount]
10:         if ParentCount == 1 then
11:             Players[i] = newPlayer(Random.UniformInt(KeepCount))
12:         else if ParentCount > numKeptPlayers then
13:             ParentCount = numKeptPlayers
14:         else
15:             PlayerCopy = newPlayer * [numPlayersKept]
16:             for j = 0 to numPlayersKept − 1 do
17:                 PlayerCopy[j] = newPlayer(Elite[j])
18:             end for
19:             for j = 0 to ParentCount − 1 do
20:                 Swap(PlayerCopy[j], PlayerCopy[j + Random.UniformInt(KeepCount − j)])
21:                 Parents[j] = PlayerCopy[j]
22:             end for
23:         end if
24:     end if
25:     for j = 0 to ParentCount do
26:         Weights[j] = Random.UniformDouble()
27:     end for
28:     normalise(Weights)
29:     for j = 0 to NumLinks do
30:         Value = 0
31:         for k = 0 to ParentCount do
32:             Value+ = Parents[k].links[j] × weights[k]
33:         end for
34:         Players[i].links[j] = Value
35:         random = Random.UniformDouble()
36:         if random < mutationLikelihood then
37:             Players[i].links[j]+ = (Random.Gaussian(mutMean, mutDev)
38:         end if
39:     end for
40: end for
41: end EVOLVE_PLAYERS
```

Figure 4.10: The procedure to evolve agents

the agents kept from the previous generation. In lines 19 through 21, random parents are chosen from the dummy array, by randomly moving agents to the front of the array, and selecting them (lines 20 and 21, respectively). Lines 25 through 27 choose biases for the parents. The new agent, called the *child* agent, has a neural network composed of weights that are biased averages of the weights of the neural networks of the parents. The biases are chosen randomly in line 26 and are normalised so that they sum to 1 in line 28. Lines 31 through 34 set the values of the weights in the neural network.

The final phase of the evolution is to apply some mutation factor to the neural network. Without mutation, there will be no exploration of the decision space, as all child neural networks are an average of their parents' neural networks. The system will only converge, and never explore areas outside the areas defined by the parents' neural networks. Lines 35 and 36 determine whether or not to apply mutation to the current weight in the neural network. A uniformly random value between 0 and 1 is chosen (line 35), and compared to the mutationLikelihood variable. This variable was arbitrarily set to 0.1, and means that approximately 10% of the weights will be slightly changed. Line 37 applies a normally distributed random value with a mean of 0 and a standard deviation of 0.1. These values were chosen to promote many small changes as opposed to a few large changes. A few large changes might lead to no real change in the network if the nodes changed are functionally deactivated (such as when previous layers in the network result in a 0 reaching the specific node). Many large changes make it too likely that the new network will have nothing in common with the old ones, counter-acting the selection procedure.

The following example shows the evolution of a neural network using the procedure described in Figure 4.9. For simplicity, the neural networks are small, with two nodes on the input level, two on the hidden level, and one on the output level. Furthermore, there are only three agents that were kept from the previous generation. Table 4.10 shows the agents immediately after they have been selected. The numbering of the weights is the same as in Figure 4.1.

Table 4.10: Three agents immediately after selection from the previous generation

Agent	IB11	IB12	OB11	IH11	IH12	IH21	IH22	HO11	HO12
0	0.2	0.1	-0.7	0.5	0.7	0.2	-0.1	-0.3	0.2
1	0.4	-0.2	-0.3	0.2	-0.5	0.3	-0.2	0.4	0.7
2	-0.1	0.2	- 0.2	-0.1	-0.2	0.6	0.8	-0.3	0.5

In the previous generation, agent 0 outperformed all other agents, with agent 1

coming in second, and agent 2 coming in third. The first step when creating a new agent is to select the parents of the new agent. It is decided that there will be two parents. Agents 0 and 2 are chosen as the parents, as shown in Table 4.11.

Table 4.11: The parents selected for the new agent

Agent	IB11	IB12	OB11	IH11	IH12	IH21	IH22	HO11	HO12
0	0.2	0.1	-0.7	0.5	0.7	0.2	-0.1	-0.3	0.2
2	-0.1	0.2	- 0.2	-0.1	-0.2	0.6	0.8	-0.3	0.5

After selecting the parents, weights must be chosen for each of the parents. For this example, the parents are weighted evenly, each with a weight of 0.5. The child is then created through the combination of the neural networks of the parents. Each connection weight of the neural network is calculated using Equation 4.12.

$$ChildLink[i] = (Parents[0] -> Link[i] \times Weights[0]) + (Parents[1] -> Link[i] \times Weights[1]) \quad (4.12)$$

The connection weights of the neural networks of the two parents are combined in such a fashion that the child is a weighted average of the parents. In this case, each of the connection weights in the child's neural network is the weighted mean of the connection weights of the parents' neural networks. Thus, IH00 becomes (0.5 x 0.5) + (0.5 x 0.1), which equals 0.2. This combination is performed for all the weights of the parents' neural networks, to create the child agent. Table 4.12 shows the new agent that is created. After the child agent is created, mutation is applied to the weights in the child's neural network, as shown in Table 4.13.

Table 4.12: Creating a new agent from the parents

Agent	IB11	IB12	OB11	IH11	IH12	IH21	IH22	HO11	HO12	weight
0	0.2	0.1	-0.7	0.5	0.7	0.2	-0.1	-0.3	0.2	0.5
2	-0.1	0.2	- 0.2	-0.1	-0.2	0.6	0.8	-0.3	0.5	0.5
Child	0.05	0.15	-0.45	0.2	0.25	0.4	0.35	-0.3	0.35	NA

Table 4.13: The child agent before and after mutation

Agent	IB11	IB12	OB11	IH11	IH12	IH21	IH22	HO11	HO12
Child	0.05	0.15	-0.45	0.2	0.25	0.4	0.35	-0.3	0.35
New	0.02	0.15	-0.52	0.4	0.17	0.52	0.08	-0.45	0.37

This process is repeated as many times as are necessary, until the population of agents is again full. The system then returns to the evaluation shown in line 8 of Figure 4.10.

4.2 Countermeasures to prevent problems with evolutionary algorithms

Poker is not transitive. If an agent A can defeat another Agent B regularly, and B can defeat a third agent C regularly, there is no guarantee that A can defeat C regularly. It is not possible to guarantee that the evolutionary algorithm is progressing in each generation. Local improvement may coincide with a global decline in fitness. In [34], it is suggested that as long as there exists some set of potential new individuals such that at least one can be found to defeat any sub-optimal existing agent, the system will be able to improve. Agents may be able to defeat opponents that were previously evaluated to be better than they were.

In [34] it is suggested that evolutionary algorithms can occasionally get caught in mediocre loops, with "better" agents simply exploiting the weakness of the opponent, only to be exploited by a similar opponent in the future. In [32], it is suggested that an evolutionary system can lose its learning gradient, or fall prey to a deficiency known as *Evolutionary Forgetting*. Evolutionary forgetting can be defined as the tendency for a population to lose good strategies as they are replaced by seemingly better ones.

For example, an exploitative strategy can evolve that can defeat the current best strategies, but few others. The most successful genetic material from a generation will be passed on to the next generation. If the best strategies from a generation are merely exploitative, the best non-exploitive strategies can be lost.

Evolution can also come to a standstill if it gets stuck in a local maximum. The current strategy will consistently defeat any strategies that are only slightly different, and the system will never progress. When a strategy becomes stuck in a local maximum, all similar strategies appear inferior.

In [34] several potential solutions to counter the problems inherent in evolutionary methods are presented. The suggestions made are for two player games, and thus are either not advisable, or need modification when applied to the game of Poker, if it is to played with more than two competing players. Several of the suggestions were adopted, and the remainder of this section describes the motivation behind the exclusion of certain suggestions, as well as a description of the modifications and implementation of the adapted methods.

4.2.1 Selecting a fitness function

When using evolutionary methods, it is necessary to choose an appropriate fitness function. In large populations of competing agents, it seems logical to reward agents that can defeat the most opponents. In large populations, however, it may be that one agent won many games against easy opponents, while another played against much harder opponents, and won less games. In [34], *fitness sharing* is described. Rather than simply rewarding agent strategies that can defeat many opponent strategies, uniqueness is rewarded. It is not enough that a strategy can defeat x other strategies; all of the other x strategies may be poor strategies that are defeated by a majority of the other strategies. Rather, a strategy is given a bonus to its fitness score if it can defeat a relatively difficult opponent, one that cannot be defeated by many other strategies.

Within the system described in this thesis, fitness is not measured by how many opponents an agent can defeat in a single Poker tournament. Instead of comparing which agents can defeat which others, agents are ranked according to how they perform across many tournaments. After a tournament is played, agents are ranked. The agent that won the tournament is ranked first, the agent that placed second is ranked second, and so on, until the first agent that was eliminated from the tournament is ranked 500th. 500 tournaments are played, and the rankings achieved over the 500 tournaments are averaged. Even if a player gets lucky in one or two tournaments, in the long run, the best agents will be chosen to continue. This fitness function is called the *plain* fitness function.

Two other fitness functions are alternatively used: HandsWon, and MeanMoney. The HandsWon fitness function rewards agents that win many hands. An agent may be doing well in a tournament, and winning hand after hand, but loses to a lucky, but inferior opponent. The HandsWon fitness function is a mixed fitness function. The HandsWon is given a score according to Equation 4.13. The ranking of agents is based upon a weighted average of the plain fitness function, and the HandsWonRanking function. The HandsWonRanking is achieved by ranking the agents according to how many hands they win in a tournament. For example, an agent may place 1st in a tournament, but will have won the 5th most hands in the tournament. Thus, it has a plainRanking of 1, but a HandsWonRanking of 5. If the HandsWon fitness function is sampled at 50%, the HandsWonWeight is 0.5, and the HandsWonScore is 3.0. If another agent place 2nd in the tournament, and won the 2nd most amount of hands,

its HandsWonScore will be 2. According the the HandsWon fitness function, sampled at 50%, the second agent is the better one.

$$HandsWonScore = plainWeight \times plainRanking + HandsWonWeight \times HandsWonRanking \quad (4.13)$$

The MeanMoney fitness function ranks agents based upon how much money they win per hand. In the HandsWon fitness function, an agent may be getting all of its opponents to fold, and may not be winning much money. During the single tournaments, whenever an agent won a hand, the amount of money that that agent had won was incremented. Similarly, when an agent went to showdown and lost, the amount of money lost was incremented. After each generation, the agents were ranked according to Equation 4.14:

$$MeanMoney = \frac{MoneyWon}{HandsWon} - \frac{MoneyLost}{HandsLost} \quad (4.14)$$

4.2.2 Halls of fame

Another measure that is suggested in [34] to counter evolutionary forgetting is a structure called a hall of fame. Figure 4.11 shows how the hall of fame is incorporated into the population of agents.

A hall of fame stores previously successful strategies, and is used as a benchmark against which the current strategies are evaluated. The hall of fame is separate from the population of agents, as shown at the top of Figure 4.11. The agents in the hall of fame and in the population are combined to form the playing population. These agents then play in the tournaments and are ranked. Table 4.14 shows the differences between how an agent of the regular population is ranked, as opposed to an agent of the hall of fame. In Table 4.14, although agent #507 ranks third in the tournament overall, it is the highest ranked agent of the hall of fame. Similarly, agent #615 is second in the hall of fame. Agents that are not part of the hall of fame are given a rank of -1, which is not obtainable otherwise.

Returning to Figure 4.11, after the tournaments are played, the two populations (i.e., the regular population and the hall of fame population are sorted according to their respective rankings. The weakest agents in the hall of fame are replaced according to Figure 4.12.

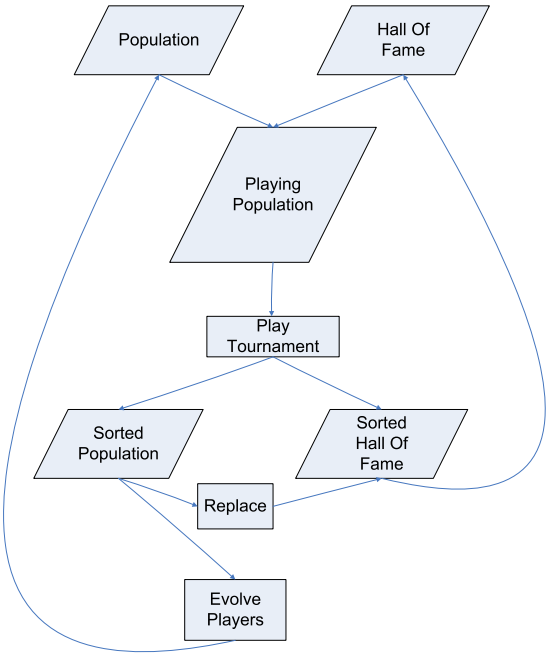

Figure 4.11: Selection and evolution using a hall of fame

Table 4.14: The child agent before and after mutation

Agent	Ranking	HOFAgent	HOFRanking
5	1	false	-1
102	2	false	-1
507	3	true	1
615	4	true	2

Lines 4 through 6 of Figure 4.12 choose which agents should be replaced within the hall of fame. Since the agents are sorted according to their ranks, only the worst agents need to be compared to the best agents in the regular population. If the worst agent in the hall of fame ranked lower than an agent in the regular population, then the agent in the hall of fame is replaced. This is repeated until no such agents exist. In Figure 4.11, the new hall of fame is then used in the next generation, while the population of agents is sent to the EVOLVE_PLAYERS function described in Figure 4.10, before becoming the new population for the next generation. After the next set of tournaments is played (i.e., after line 9 in 4.9), the hall of fame will be sorted according to the rankings obtained in the tournaments.

The hall of fame in [34] is used in an environment that is evaluating a two-player

```
1: procedure REPLACE(HallOfFame[], hallSize, Players[], numPlayersKept)
2: begin
3: j = 0
4: for i = 0 to numPlayersKept do
5:     if HallOfFame[hallSize − numPlayersKept + i].OverallRanking() > Players[j].OverallRanking() then
6:         HallOfFame[hallSize − numPlayersKept + i] = Players[j + +]
7:     else
8:         break
9:     end if
10: end for
```

Figure 4.12: The procedure to replace agents in the hall of fame

game, and at each generation, a single strategy is added to the hall of fame, resulting in a hall of fame of unbounded size. For a large scale game with the potential to add many strategies to the hall of fame, an unbounded hall of fame was determined to be too costly. Furthermore, due to the chance introduced with the cards, the entire hall of fame was included in the playing population, rather than a random selection from an infinite hall of fame. In this manner, agents are evaluated against all members in the hall of fame, and the weakest members of the hall of fame agents are regularly replaced.

4.2.3 The phantom parasite

The concept of a *phantom parasite* is another method suggested to counter some of the drawbacks of evolution [34]. A phantom parasite is a special agent that prevents the best agents from dominating a population, and the worst from being eliminated. A phantom parasite is an agent that does not compete against other evolving agents. When it is time to evaluate the agents, if there is any agent that defeats all of the other agents consistently, the phantom parasite beats this "perfect" agent. Likewise, if an agent loses to all the other agents consistently, the phantom parasite loses to this agent. If parents are selected based upon the percentage of other agents that they can defeat, this ensures that no agent completely replaces the population, and it ensures that no agent is completely lost. Table 4.15 shows an example of the phantom parasite.

Table 4.15: An example of the phantom parasite

Agent	Opponents defeated	Defeated by	Percentage Wins
1	2,3,4	Phantom	75
2	3,4	1	67
3	4	1,2	33
4	Phantom	1, 2, 3	25
Phantom	1	4	NA

In Table 4.15, the Phantom agent does not actually belong to the population. However, since agent 1 can defeat all the other opponents (i.e., 2, 3 and 4), agent 1 is defeated by the phantom parasite. Without the phantom parasite, agent 1 has no losses, and agent 4 has no wins. If parents are chosen based upon the percentage of games that an agent wins, agent 1 will always be chosen, and agent 4 will never be chosen. Eventually, all agents will be similar to agent 1, and the agents could get stuck in a local maximum.

In a game like Poker, it becomes very difficult to give meaning to the phantom parasite. Very rarely does any particular strategy win all of the time. The stochasticity of the game of Poker is enough to guarantee that there will never be a case where an agent wins all of its games against all of its opponents.

4.2.4 Brood selection

In [34] a method called *brood selection* is also discussed. Many more agents than are necessary are created at each generation. These agents compete in smaller competitions to determine the fittest of the new agents, and the fittest agents are inserted into the population of evolving agents. This method attempts to reduce the likelihood that an unfit agent will be introduced into the population.

Unfortunately, brood selection is not a viable option to increase the fitness of Poker playing agents. Poker, although considered a game of skill, still relies heavily upon chance. As described in Section 4.2.1, it is often impossible to determine the better agent in a limited series of games, particularly as the agents become better at the game in question [6]. In order to determine which members of the brood should be promoted to the full population, many thousands of determinating games would have to be staged, which would be similar in scope to a full generation.

4.2.5 Co-evolution

Finally, [26, 32, 34] suggest the use of co-evolutionary methods instead of simple evolution. Co-evolution consists of several populations of agents evolving separately, but competing against each other. By evolving the populations separately, it is hoped that each population will develop its own strategies. When the resulting agents compete against each other, the strongest agents are the ones that can defeat agents from the opposing population. As the co-evolution proceeds, a situation known as

an *arms-race* often develops. In an arms-race, one population evolves strategies that are specially developed to defeat the strategies of the opposing population. The second population will then evolve strategies to defeat the agents original population. The populations take turns having the strongest agents, and they may progressively develop strategies that are not only successful against the opposing population, but in general.

Several changes were made to the method of multiple populations suggested in [26, 32, 34], again due to the nature of the game of Poker. Agent competition could not be strictly restricted to members of opposing populations. Unless tournaments were unreasonably small, or there were a large number of separate populations, it was necessary to include many individuals from each separate population.

As with the hall of fame described in Section 4.2.2, all agents in each of the separate populations were inserted into a larger competing population. Agents competed against agents from both their own sub-population and the other populations. Similarly to Figure 4.11, the sub-populations are combined into a larger playing population. After the tournaments have been conducted, the sub-populations are evaluated and evolved separately. When it is time to create new agents for the next generation, agents in a sub-population are only combined with other agents from their sub-population. Instead of selecting all of the elite agents from the larger population (which might all belong to the same sub-population), an equal number of elite agents are selected from each sub-population. Although one sub-population may be less skilled than another, its agents are allowed to evolve, and attempt to catch up to agents in the more skilled sub-populations.

When halls of fame were included in the process, each sub-population was given its own hall of fame, rather than a hall of fame belonging to the entire population. As mentioned in Section 4.2.2, the goal of the halls of fame was to protect older strategies that might get replaced in the population. In a co-evolutionary environment, it is entirely possible that one sub-population may become dominant for a period of several generations. If a global hall of fame is used, the strategies of the weaker sub-populations would quickly be replaced in the hall of fame by the strategies of the superior sub-population. Each sub-population was given its own hall of fame to preserve strategies that might not be the strongest in the larger population, but could still be useful competitive benchmarks for the evolving agents.

4.3 Duplicate tables

Poker is a game with a high degree of variance. Skill plays a large part in the determination of which players will win regularly, but if a good player receives poor cards, he will most likely not win. In [9], a method for evaluating agents is discussed, which is modeled upon the real-world example of *duplicate tables*. In professional Bridge tournaments, duplicate tables are used to attempt to remove some of the randomness of the cards.

A duplicate table tournament is actually a collection of smaller tournaments called single tournaments. Agents compete at a table of nine agents, which will be described in section 5.1. Agents play hands of No-Limit Texas Hold'em until only one agent remains at the table, at which point their rankings are collected. The agent that won the single tournament is given a ranking of 1, the player that finished second is given a ranking of 2, and so on, until the agent that finished last in the single tournament is given a ranking of 9.

After the first single tournament, the deck is reset to its original state. All of the agents are shifted one seat clockwise along the table, and another single tournament is played. After the single tournament, the rankings are again collected. This process is repeated until each agent has played at every seat at the table. This shifting of seats and repetition of the same cards ensures that if an agent won a single tournament solely due to a lucky set of cards, it will be offset by all of the other agents receiving the same cards. After the agents have each played at each seat, their average ranking from the nine single tournaments is calculated, and this value is considered to be their ranking for the duplicate tables tournament. After the completion of a duplicate tables tournament, another duplicate tables tournament begins. The deck is re-shuffled, so that the cards of this duplicate tables tournament will be different from the cards of the previous duplicate tables tournament.

Agents will see roughly the same cards (i.e., hole cards, the flop, turn and river will be the same) for single tournaments within a duplicate table tournament. A different decision at the same decision point, however, can lead to an agent seeing cards that an opponent would not have seen in the same situation. Likewise, an agent may fold its cards where an opponent did not, and miss seeing certain cards. These differing cards can lead to different results, although agents see roughly the same cards as their opponents.

Chapter 5

EXPERIMENTAL RESULTS

This chapter is organised as follows. Section 5.1 describes the benchmark agents used to evaluate the evolved agents. Section 5.2 presents the experimental results obtained using the duplicate tables method. Section 5.3 introduces a method for evaluating the progress made by the agents during evolution, and presents the results obtained using such a method.

5.1 Opponents

In order to evaluate the evolved agents, a number of benchmarks were used. With little literature and no existing benchmarks related to No-Limit Texas Hold'em, several evaluation agents were devised. In [3, 4, 11, 38], several static agents, which always play the same way regardless of the current state of the game, are used as benchmarks. These agents are admittedly weak benchmarks, but they are supplemented by three agents developed in [3], and are used to gauge the ability of the evolved agents. In the various experiments, the opponents do not change, and any increases or decreases in an agent's ranking can be attributed to an increase or decrease, respectively, in its skill.

5.1.1 Static opponents

The first five opponents are static agents: Random, Folder, Caller, Raiser, and CallOrRaise. Regardless of the state of the game, these agents do not change their playing style. They do not model their opponents and do not respond to their cards. Consequently, aggressive static agents (i.e., Raiser) cannot be bluffed, and passive static agents (i.e., Folder) cannot be drawn into a hand through sandbagging.

5.1.1.1 Random

Random is the simplest static agent. It is an agent that will fold, call, and raise with equal likelihood, regardless of its cards. Against a random agent in heads-up play, it is probably a good idea to raise the bet. Eventually, the agent will fold its cards, and the agent playing against Random will win all of the money in the pot. In a game with multiple opponents, the random agent can be ignored, as the other agents will dictate how the hand should be played.

5.1.1.2 Folder

Folder is the most passive of the static agents. Whenever Folder receives a betting decision, it folds its cards. In the case where it has no bet to call, it will check instead of folding. In heads-up play, it is a good idea to bet against Folder, instead of letting it check and potentially win a hand. In the long run, if an agent does not continually check, it will win in a heads-up confrontation against Folder, although it will only ever win the blinds in a hand. In a game with multiple opponents, there is nothing that can be done to eliminate Folder before the blinds take all of its chips. Like the games including Random, other opponents will dictate how the hand will be played.

5.1.1.3 Caller

Caller is an agent that calls whenever it gets the chance to bet. In heads-up play, an agent should play according to its cards when playing against Caller. If the agent has poor cards, it should probably fold to Caller. If the agent has medium-strength cards, it should call, as it is equally likely that Caller has better or worse cards. If the agent has good cards, it is a good idea to raise the bet, as the caller will never be scared away by a large bet. Against multiple opponents, the agent should play similarly against the caller, although it should also consider the other agents at the table.

5.1.1.4 Raiser

Raiser is an agent that raises the bet every time it gets the chance to bet. It has an equal likelihood of making a small, medium, or large bet. In heads-up play, it can be difficult to play against Raiser, as it will cost a lot of money to see cards. An agent should play against Raiser similar to how it would play against Caller, but

with a higher card threshold (i.e., the agent needs better cards to call against a Raiser than a Caller, and it should only raise with the very best hands). Against multiple opponents, an agent's strategy depends upon the strategies of its other opponents. If there is another aggressive agent at the table, the agent can fold, and hope that one of the agents eliminates the other. The agent could also play similar to heads-up, and only call and raise with very good cards.

5.1.1.5 CallOrRaise

CallOrRaise is a combination of Caller and Raiser. This agent will call and raise the bet with equal likelihood. When CallOrRaise raises, it makes small, medium, and large raises with equal probability. When an agent is playing against CallOrRaise, it may not determine CallOrRaise's strategy as quickly as against Raiser or Caller. The agent should play a strategy somewhat between the strategy for Caller and Raiser. The agent should play according to the value of its cards, and wait for a hand that it is fairly confident it can win.

5.1.2 Dynamic opponents

Three dynamic benchmark agents were developed in previous work [3] that play according to the current state of the game, and attempt to model and exploit opponents.

These agents make betting decisions via *action topographies*, two-dimensional matrices which map the cost of a hand (i.e., how much money an agent must call, and how much it can win) and the strength of a hand (i.e., the likelihood that an agent's cards will win a hand) to a probability vector that determines betting decisions. These agents were also developed using evolutionary methods, via the introduction of random *hills* to the action topographies. A *head-start* (i.e., expert data inserted into the action topographies prior to evolution) was also used to guide the evolution. Opponent modeling is implemented through the skewing of hand strength depending upon the aggressiveness of the opponents.

5.1.2.1 OldStart

OldStart is an agent that uses the head-start data in its action topography, but is not evolved. Unlike the static agents, this agent attempts to model the opponents,

and will change its strategy depending upon the current state of the game. Based upon the results reported in [3], it is expected that this agent will outperform the static agents, but be the weakest of the dynamic agents.

5.1.2.2 OldBest

OldBest is an agent evolved in [3] using the head-start approach. It is the best agent from the final generation of evolution. Based upon the results reported in [3], it is expected that this agent will outperform the static agents, as well as outperform the other dynamic agents. This agent is expected to be the best benchmark opponent.

5.1.2.3 OldScratch

OldScratch is an agent that was evolved in [3], without the head-start approach. The results of [3] suggest that this agent will outperform the static agents, but will perform somewhere in between the other dynamic benchmark agents.

5.2 Duplicate table results

A series of experiments were conducted to evaluate the quality of the agents generated when evolutionary neural networks were used alone, with various fitness functions used for selection of the best agents. Agents were also generated when evolutionary neural networks were used in combination with co-evolution and a hall of fame. Duplicate table tournaments were used in all experiments described in this section.

5.2.1 Agents evolved with a single population and no hall of fame

The objective of the first experiment was to establish a set of baseline rankings for the nine agents, as shown in Figure 5.1. In Figure 5.1 (and in all other figures), the Rank value described along the vertical axis corresponds to the skill level of an agent relative to the other agents, where better agents receive higher (but numerically lower) ranks. The *baseline Control* agent was evolved over 500 generations, with a population of 1,000 agents competing in 500 single tournaments per generation. In the first generation, the population consists of 1,000 randomly created agents. At the

completion of the tournaments for each generation, the 100 most highly ranked agents (i.e., the elite agents) are used as the initial population for the next generation, and the other 900 agents in the population are created through the combination and mutation of these elite agents. At the completion of the tournaments for the 500th generation, the most highly ranked agent was selected as the baseline Control agent. The baseline Control agent then played in an evaluation tournament consisting of 100,000 duplicate tables tournaments against the benchmark agents outlined in section 5.1.

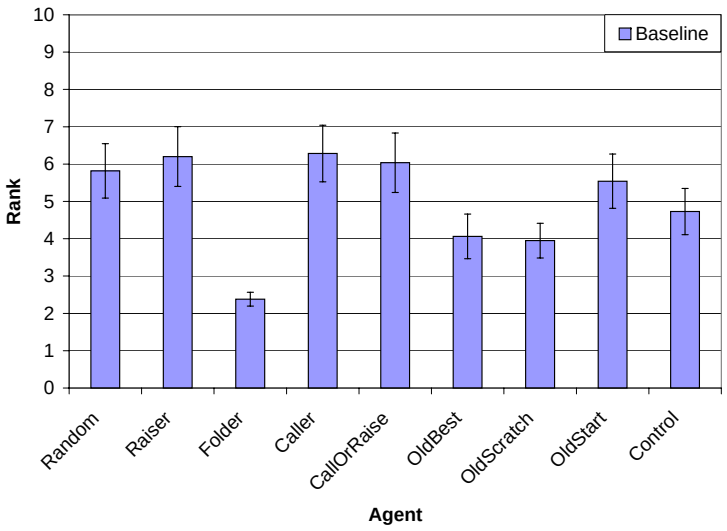

Figure 5.1: Results of duplicate tournament evaluation of baseline

In Figure 5.1, the lowest rank was 6.28, the highest rank was 2.38, the median rank was 5.54, and the standard deviations of the agent ranks varied from 0.19 to 0.80, with a median value of 0.73. Random, Raiser, Caller, and CallOrRaise (i.e, four of the static agents) performed poorest, receiving 6.28, 6.04, and 6.17 as the minimum, maximum, and mean ranks, respectively. Folder performed best with a rank of 2.38. Note however, that the strategy of Folder has an interesting characteristic within the context of No-Limit Texas Hold'em Poker. That is, it would seem that folding every hand and waiting for opponents to eliminate one another is a reasonable strategy, although Folder will never win a hand, will never win a tournament, and, at best, will finish in second place. OldBest and OldScratch (i.e, two of the dynamic agents) performed similarly, receiving a mean rank of 4.01. OldStart received a rank of 5.54, while the baseline Control agent received a rank of 4.73, placing it slightly better than the median rank.

Even though the duplicate tables tournament format contributed to a reduction in variance, the standard deviation is still relatively high, approximately 20% of the ranks obtained. A two-tailed paired t-test was conducted on the null hypothesis that the ranks of any two distinct agents were equal. In all cases, and for all experiments, the null hypothesis was rejected with 99% confidence.

In the graphs that follow in Sections 5.2.2 through 5.2.5, the baseline results are included for the reader's convenience.

5.2.2 Comparison of fitness functions

Three fitness functions were compared. The first fitness function, named *Plain*, is the fitness function used for the evolution of the baseline Control agent. Although the duplicate table tournaments try to remove some of the randomness from the game of Texas Hold'em, the evolution tournaments used to generate the agents do not. It is possible that an agent that does quite well in the tournaments for the most part is eliminated due to lucky cards for the opponent. Two new fitness functions were developed, named *HandsWon*, and *MeanMoney*. Their effectiveness was evaluated by comparing Control agents evolved using these fitness functions to the baseline Control agent.

5.2.2.1 HandsWon

The objective of the second experiment was to evaluate the HandsWon fitness function. The agents were evolved over 500 generations, with a population of 1,000 agents and 500 tournaments per generation. The only difference between these agents and the ones evolved in the baseline experiment are the method of selection of the parents for the subsequent generations. At the completion of each generation, the elite agents are selected using the HandsWon fitness function, sampled at 0%, 20%, 40%, 60%, 80% and 100%. The remaining 900 agents in the new population are created through the combination and mutation of these parent agents. At the completion of the 500th generation, the agent from the final generation that won the most hands is selected as the Control agent. This Control agent is known as the *Handsx* Control agent, where x is the percentage that the HandsWon function was weighted (i.e., the Hands20 Control agent is the agent evolved using the HandsWon fitness function sampled at 20%). The Hands0 Control agent is the same agent as the baseline Control agent. The Control Agent is then played against the eight benchmark opponents in

100,000 duplicate tables tournaments. The agents are evaluated based upon their rankings in the duplicate tables tournaments, and not the number of hands won.

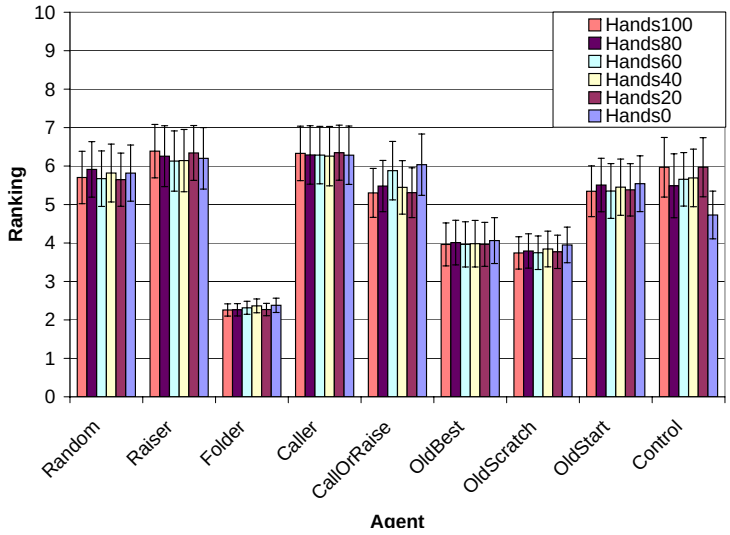

Figure 5.2: Results of duplicate tournament evaluation of HandsWon agents

The Hands100 Control agent's ranking has decreased to 5.97 from the Hands0 Control agent's ranking of 4.73, a decrease of 26.2%. The median rank of all of the Hands100 agents (i.e., the Control agent and all of the opponents) in this sample has increased from 5.54 to 5.77, and the Hands100 Control agent performs worse than the median. The best agents from the baseline experiment, namely Folder, OldScratch, OldBest, and OldStart seem to have gained the most from the decrease in skill of the Control agent, improving by 5.1%, 5.3%, 2.5%, and 3.5%, respectively. The mean ranking of any one tournament remains constant, at 5.0, so any loss of rank of one agent must be balanced such that the sum of the gains in rank of all the other agents equal to the loss for the Control agent. Furthermore, the Hands100 Control agent's rank of 5.97 was defeated by the Random and CallOrRaise static agents' respective ranks of 5.70 and 5.30, as well as all the agents that defeated the Hands0 Control agent.

The Hands80 Control agent also does not perform as well as the Hands0 Control agent, showing a decrease in rank of 16.1%, from 4.73 to 5.49. However, the Hands 80 Control agent does show a slight improvement over the Hand100 Control agent, improving upon the Hand100's Control agent's rank of 5.97, an improvement of 8.1%. This improvement is enough that the Hands80 Control agent is able to defeat Random,

and approach CallOrRaise, both of which were defeated by the Hands0 Control agent, but not by the Hands100 Control agent. The median rank is 5.49, and is equal to the rank of the Hands80 Control agent. Although the Hands80 Control agent is not as skilled as the Hands0 Control agent, the mixture of the two fitness functions performs better than the HandsWon function on its own.

Unfortunately, it would seem that the improvement of the Hand80 Control agent over the Hand100 Control agent has not translated to the Hands60 Control agent. The Hands60 Control agent's rank has decreased to 5.66, and although it still defeats most of the static agents, its ranking is equal to that of Random. It represents an increase of 19.6% over the Hands0 Control agent, and an increase of 3.1% over the Hand80 Control agent. It is more stable than the Hand80 Control agent, with a standard deviation of 0.70, as opposed to 0.83 for the Hand80 Control agent, but it is more unstable than the Hands0 Control agent's standard deviation of 0.62. The best rank achieved was again by Folder, with a rank of 2.31, and the worst rank was Caller's 6.29, with a median rank of 5.66. OldScratch's rank increases by 5.2%, which was the greatest increase in the sample, while OldStart's rank increases by 3.4% and OldBest's rank increases by 2.4%.

The Hands40 Control agent does not improve upon the Control agent of Hand60 or Hand80, achieving a rank of 5.69. The median rank of the agents is 5.57, while the best rank is 2.36 and the worst is 6.36 for Folder and Caller, respectively. Like the Hands100 Control agent, the Hands40 Control agent fails to perform at the level of the median of its sample. OldScratch's rank increases by 2.7%, while OldStart's rank increases by 1.6% and OldBest's rank increases by 2.0%. The best increase in ranking over the Hands0 Control agent is CallOrRaise, which increases by 9.7%. Although the mixed value is more heavily weighted towards the tournament rankings, the Control agent is getting worse.

Although the tournament rankings are only slightly different from the 100% used in Hands0, the Control agent achieves a rank of 5.97, an increase of 26.3% over the Hands0 Control agent. The best rank is 2.27 and the worst is 6.35 for Folder and Caller, respectively, while the median rank is 5.51, and the Hands20 Control agent performs seventh out of the nine agents in its sample. OldScratch's rank increases by 4.5%, while OldStart's rank increases by 2.9% and OldBest's rank increases by 2.4%. Like Hands40, the greatest increase is seen by CallOrRaise, which improves by 12.1%. The Control agent is tied with the Hands100 Control agent for the worst

rank of the HandsWon Control agents. The number of hands won introduces enough randomness into the fitness function that even when the weighting is only 20% for the number of hands won, the Control agents of the HandsWon experiment are worse than the baseline Control agent.

It would seem that simply using the number of hands won by a particular agent is not as good of a fitness function as the tournament ranking function was. As with the previous test, a paired t-test was performed, and the same null hypothesis was rejected with a confidence of 99%.

5.2.2.2 MeanMoney

The objective of the next experiment, named the *MeanMoney* experiment, is to evaluate the effectiveness of the MeanMoney fitness function. The top 100 agents were selected as the elite agents, using the MeanMoney fitness function, and the other 900 agents for the next generation were created through the evolution of these elite agents. The best agent from the 500th generation was chosen as the Control agent. This agent participated in 100,000 duplicate tables tournaments against the eight benchmark agents.

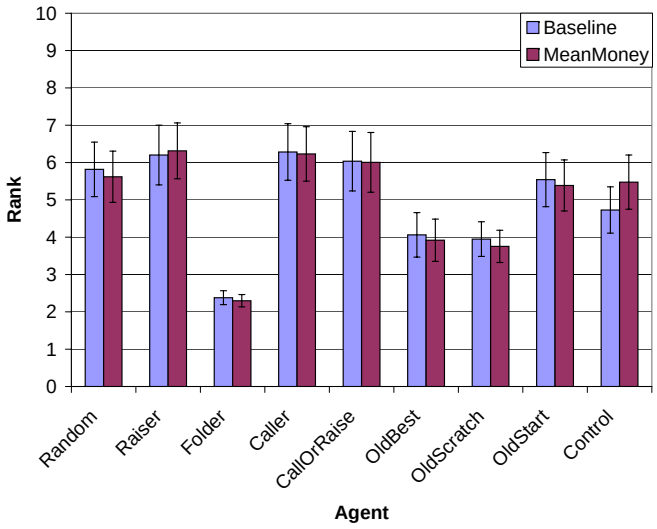

Figure 5.3: Results of evaluating MeanMoney agents

Figure 5.3 shows the results of the agents evolved using the MeanMoney fitness function. The ranking of the Control agent is 5.48, approximately equal to the best HandsWon Control agent. However, it is still 15.8 % worse than the baseline Control

agent. The mean rank of the agents in this experiment is 5.00, while the median is 5.48. Like the baseline Control agent and all of the HandsWon Control agents the MeanMoney Control agent is unable to defeat any of the dynamic opponents, although it does defeat all of the static opponents. The best rank again goes to Folder with a rank of 2.30, an increase of 3.4% over the baseline, and the worst rank is 6.31 for Raiser, which is a decrease of 1.8 % from the baseline. It would appear that the MeanMoney fitness function is ill-suited to be the fitness function.

5.2.3 Agents evolved with co-evolution

The objective of the next experiments is to attempt to measure the effectiveness of the concept of an arms race, achieved through the use of a co-evolutionary environment. Whereas the baseline agents were evolved in a single population of 1,000 individuals, all competing and inter-breeding, the co-evolutionary environment separates the agents. Rather than one population of 1,000 agents, there are multiple populations. The number of agents in the populations sums to 1,000. The agents compete against both members from the other population and their own, but are restricted in the selection of potential breeding partners to members of their respective population.

5.2.3.1 Two-population agents

The first co-evolutionary experiment aims to split the population into two sub-populations called *2Pop-1* and *2Pop-2*. They are evolved in separate populations of 500 agents apiece, each evolved over 500 generations with 500 tournaments per generation. At the end of each generation, the top 50 agents from each sub-population are chosen as their population's respective elite agents. The remaining 450 agents in each population are generated from their own elite agents. For example, the 450 agents from the first sub-population are created from the 50 elite agents from the first sub-population, and have no access to the agents from the second sub-population. After the 500th generation, the best agents from each sub-population are selected as the Control agents for 2Pop-1 and 2Pop-2, respectively. Each of these Control agents participates in 100,000 duplicate table tournaments against the eight benchmark agents.

In Figure 5.4, the rankings for the five static agents and three dynamic agents are similar to those obtained in the baseline results of Figure 5.1. The rankings for the Control agents from 2Pop-1 and 2Pop-2 are higher than the rankings for

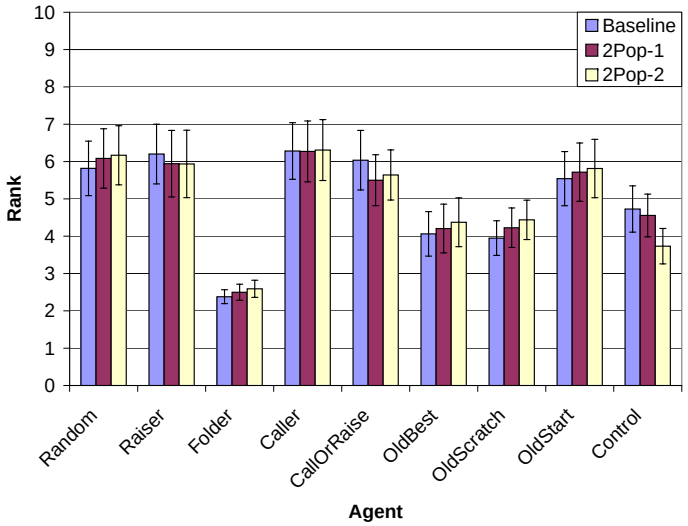

Figure 5.4: Results of using 2 separate populations in the evolutionary process

their respective baseline Control agents, representing an increase of 3.7% and 21.0%, respectively. The 2Pop-1 Control agent was also able to obtain a higher rank than OldBest and OldScratch, higher by 14.6% and 15.9%, respectively. The only difference between the Control agent in Figure 5.1 and the Control agents in Figure 5.4 is that the latter were co-evolved.

The Control agents evolved using co-evolution improved upon the baseline Control agents. Although the populations in which the agents were evolved contained fewer agents, and thus less competition, the agents saw an improvement. Although the 2Pop-1 Control agent was not quite as skilled as the 2Pop-2 Control agent, it still showed an improvement of 3.7% over its respective baseline Control agent. It appears that co-evolution with two sub-populations does lead to an improvement in the No-Limit Texas Hold'em playing agents.

5.2.3.2 Four-population agents

The next experiment was conducted to see if the results from Section 5.2.3.1 could be expanded and improved upon by further splitting the population. The control agents were evolved from four sub-populations of 250 agents apiece, evolved for 500 generations with 500 tournaments per generation. At the end of a generation, the 25 best agents from each sub-population were chosen as the elite agents of their respective sub-population. The remaining 225 agents in each sub-population were

generated from these elite agents, and completed their respective sub-populations. After the 500th generation, the best agents from each population, named *4Pop-1*, *4Pop-2*, *4Pop-3*, and *4Pop-4* were chosen as the Control agents from the first, second, third, and fourth populations, respectively. These Control agents each participated in 100,000 duplicate table tournaments against the benchmark agents.

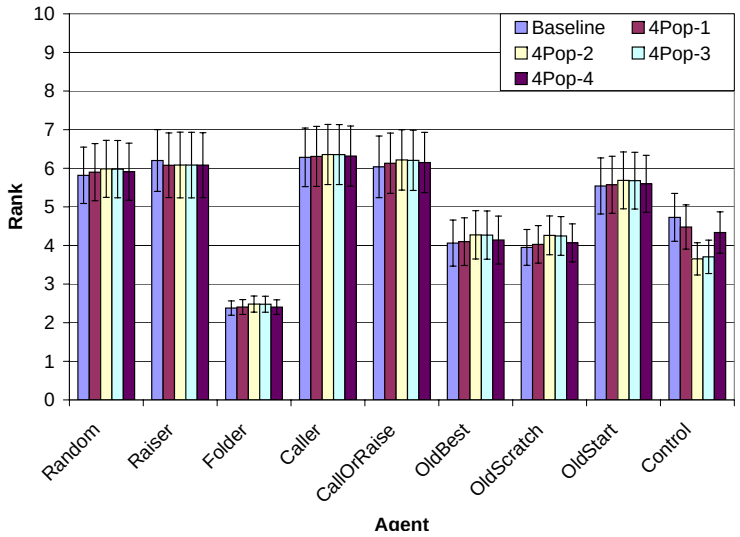

Figure 5.5: Results of using 4 separate populations in the evolutionary process

The results in Figure 5.5 are similar to those in Figure 5.4, with the Control agents of 4Pop-1, 4Pop-2, 4Pop-3, and 4Pop-4 showing an increase in rank over their respective baseline Control agents of 5.3%, 22.7%, 21.6%, and 8.3%, respectively. Furthermore, 4Pop-2 and 4Pop-3 also obtained a higher rank than OldBest by 14.5% and 13.2%, respectively, and OldScratch by 14.2% and 12.7%, respectively. An improvement over the baseline Control agents is seen, as well as a small improvement over the 2-population Control agents. The best of the 4-population agents, 4Pop-2, with its rank of 3.65, is 2.1% better than the best 2-population agent, the 2Pop-2 Control agent, with a rank rank of 3.73.

The 4-population experiment produced a slightly better agent than the 2-population experiment, but the dividing of the population was stopped at 4 sub-populations. While it would be interesting to investigate with more sub-populations, each division decreases the size of the reproducing population of each sub-population. It was decided that any further division would give too small of a reproducing population,

unless the total population size was increased. Increasing the size of the total population would have increased the time necessary to conduct experiments, and thus further sub-division has been left for future work.

5.2.4 Agents evolved with a hall of fame

The objective of the next experiment was to evaluate the effectiveness of the hall of fame structure without co-evolution. The results of this experiment are shown in Figure 5.6, where the Control agents were evolved over 500 generations. The *LargeHOF* Control agent was evolved from a reproducing population of 1,000 agents using a hall of fame of 1,000 agents, for a total playing population of 2,000 agents. When selecting parents for the next generation, only agents in the reproducing population, and not the hall of fame, are considered. At the completion of the tournaments for each generation, the 100 most highly ranked agents from the reproducing population are used as the initial population for the next generation. The other 900 are created through the combination and mutation of the elite agents. In addition, if any of the elite agents from LargeHOF have higher rank higher than the 100 lowest ranked agents from their hall of fame, these elite agents replace the lower ranked agents in the hall of fame. The *SmallHOF* follows the same procedures as the LargeHOF, except that all values for the SmallHOF are half of what they were for the LargeHOF. At the completion of the 500th generation, the most highly ranked agents from LargeHOF and SmallHOF, respectively, were selected as the Control agents. The Control agents then played in their own duplicate table tournaments against the benchmark agents.

In Figure 5.6, the rankings for Random, Raiser, Caller, CallOrRaise, and OldStart for both LargeHOF and SmallHOF are similar to the baseline. For LargeHOF, Folder, OldBest, and OldScratch have a 13.3%, 9.5%, and 12.5% lower rank, respectively, than the corresponding baseline rank, and the median rank of LargeHOF is 5.78. The Large HOF Control agent, with a rank of 2.85, has a rank 39.8% higher than the rank for the baseline Control agent, and has a higher rank than all other agents except Folder, which has a rank of 2.70. For SmallHOF, Folder, OldBest, and OldScratch have a 10.1%, 6.7%, and 8.1% lower rank, respectively, than the corresponding baseline rank, and the median rank is 5.69. The Control agent from SmallHOF, with a rank of 3.48, has a rank 26.4% higher than the rank for the baseline Control agent.

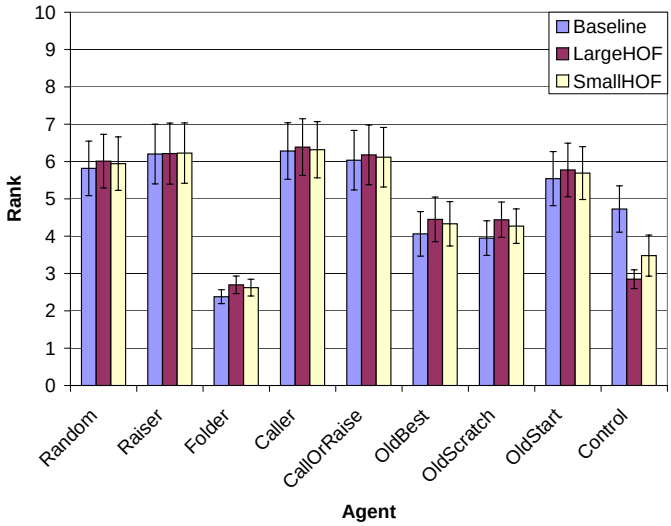

Figure 5.6: Results of evaluation of agents evolved with a hall of fame and no co-evolution

5.2.5 Agents evolved with co-evolution and a hall of fame

The objective of the final experiment was to evaluate the effectiveness of utilizing both a hall of fame and co-evolution. The results of this experiment are shown in Figure 5.7, where the Control agents were evolved over 500 generations from a population of 1,000 agents split into two sub-populations (known as *HOF2Pop-1* and *HOF2Pop-2*) of 500 agents, each with its own hall of fame of 500 agents drawn from the corresponding sub-population. At the completion of the tournaments for each generation, elite agents are selected in the same manner as the two population experiment described in Section 5.2.3, and these agents are inserted into their respective halls of fame in the same manner as the hall of fame experiments in Section 5.2.4. At the completion of the 500th generation, the most highly ranked agents from each of the two sub-populations were selected as the Control agents. The Control agents then played in their own evaluation tournaments against the other eight agents.

In Figure 5.7, the rankings for all of the agents are similar to those obtained in the hall of fame results shown in Figure 5.6. The Control agents from HOF2Pop-1 and HOF2Pop-2 have a 38.3% and 38.1% higher rank, respectively, than the baseline Control agent, and the median ranks are 5.74 and 5.73, respectively. Although the ranks of HOF2Pop-1 and HOF2Pop-2 (2.91 and 2.93, respectively), are not quite as good as the LargeHOF agent's rank of 2.85, there is another feature worth noting.

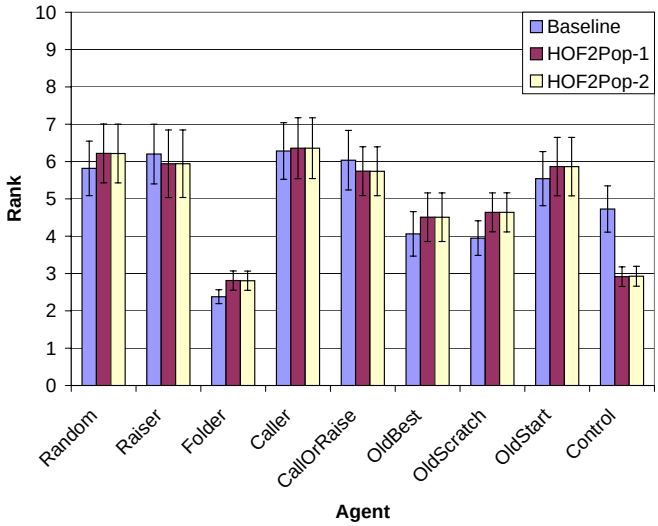

Figure 5.7: Results of evaluating agents evolved with a hall of fame and co-evolution

Whereas Folder in the LargeHOF experiment received a rank of 2.70, it received ranks of 2.81 in each of the 2PopHOF experiments. Thus, 2PopHOF-1 and 2PopHOF-2 were respectively only 3.7 and 4.3% worse than the Folder, while the LargeHOF agent was 5.6% worse. Although only by small percentages, the 2PopHOF agents were closer to being the best agents in their experiments than the LargeHOF agent.

5.2.6 Summary of Duplicate Table Tournaments

Table 5.1 shows the rankings of the Control agents for each experiment, sorted from least skilled to most skilled. The HandsWon and MeanMoney Control agents were the least skilled agents. Co-evolution improved the results over those of the baseline Control agent, with the best agent evolved with co-evolution but no hall of fame receiving a ranking of 3.66, compared to the baseline Control agent's ranking of 4.73. The hall of fame improved the agents even further. The SmallHOF Control agent out-performed all previous agents, but was out-performed in turn by the Control agents of HOF2Pop-1 and HOF2Pop-2. The best agent was evolved from LargeHOF, which used a large hall of fame, but no co-evolution.

Table 5.1: A summary of the duplicate tables tournament results

Agent	Rank
Hands20	5.97
Hands100	5.97
Hands40	5.69
Hands60	5.66
Hands80	5.49
MeanMoney	5.48
Baseline	4.73
2Pop-1	4.56
4Pop-1	4.48
4Pop-4	4.34
2Pop-2	3.73
4Pop-3	3.71
4Pop-2	3.66
SmallHOF	3.48
HOF2Pop-2	2.93
HOF2Pop-1	2.92
LargeHOF	2.85

5.3 Evolutionary progress

In Section 5.2, the best agents from the 500th generation were chosen as the Control agents. This selection was made based upon the assumption that the skill level of the agents was increasing as the generations were progressing. In order to test this assumption, *evolutionary progress tournaments* were run, and *evolutionary progress graphs* were generated from the results.

The best agent from each generation is inserted into a population of agents. These agents play in 10,000 single tournaments, and their average ranking is calculated. The evolutionary progress graph plots the average ranking of each of these agents. Evolutionary progress is defined in this thesis as either a gain or maintenance of skill from one generation to the next. For example, if the agent from generation 100 has an average ranking of 350th (out of 500 agents), then evolutionary progress is observed if the agent from generation 101 has at worst a ranking of 350th.

5.3.1 Baseline agents

The first experiment designed to evaluate the evolutionary progress of the agents was performed on the baseline agents. The best agent from each of the 500 generations of the baseline experiment performed in Section 5.2.1 was inserted into a population of 500 agents and competed in an evolutionary progress tournament. The resulting evolutionary progress graph is shown in Figure 5.8.

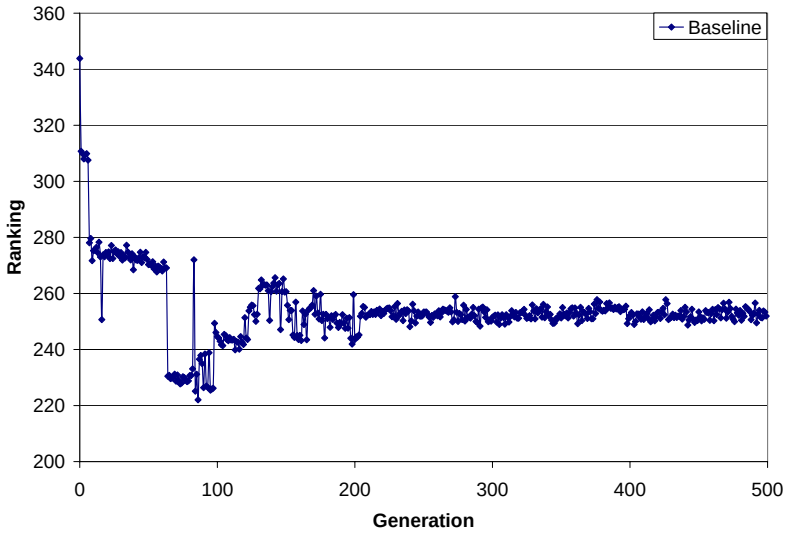

Figure 5.8: Results of playing the best baseline agent from each of 500 generations

The results in Figure 5.8 show that progress during during evolution is not monotonic. If it were monotonic, the rank of the agents should decrease as the generation increases (recall that a lower rank means a better placing in a tournament). The 500th generation agent is better than the 1-st generation agent, but between generation 50 and 150, there is a "bump" in the graph. At generation 52, the agent gets considerably better, only to return to approximately the same place as before by generation 140.

There are several possible explanations for the non-monotonic nature of the graph. First, it could simply be that not enough tournaments are being played to accurately classify the agents. Poker has a high degree of variance, and if a low number of games are played, a bad agent could seem to be better than another, or a good agent could be seen as worse than another. This possibility is explored in section 5.3.4. Another possibility is that without a strict control on the agent, it is actually getting better at generation 52, but is forgetting certain tactics along the way. By the time 100

generations have passed, although it has new techniques, it has forgotten old ones, and has returned to the skill level it previously enjoyed.

5.3.2 Evolutionary progress with the HandsWon and Mean-Money fitness functions

The HandsWon agents did not perform very well, and their evolutionary progress graphs may help to explain why. Even when the hands won was only weighted at 20%, the agents were notably worse than the baseline agents. Section 5.3.2.1 investigates the evolutionary progress graphs of the HandsWon experiments.

5.3.2.1 The HandsWon Fitness Function

The objective of the next experiments was to gauge whether or not any progress was being made on the HandsWon experiments. Like the baseline experiments, the top agents from each generation of the HandsWon experiments competed in an evolutionary progress tournament. Each of the different samplings (i.e., Hands20, Hands40, Hands60, Hands80, and Hands100) each participated in their own evolutionary progress tournament. The results for Hands20 (which are similar to the results obtained for Hands40 and Hands60), Hands100, and Hands80 are shown in Figures 5.9, 5.10, and 5.11, respectively.

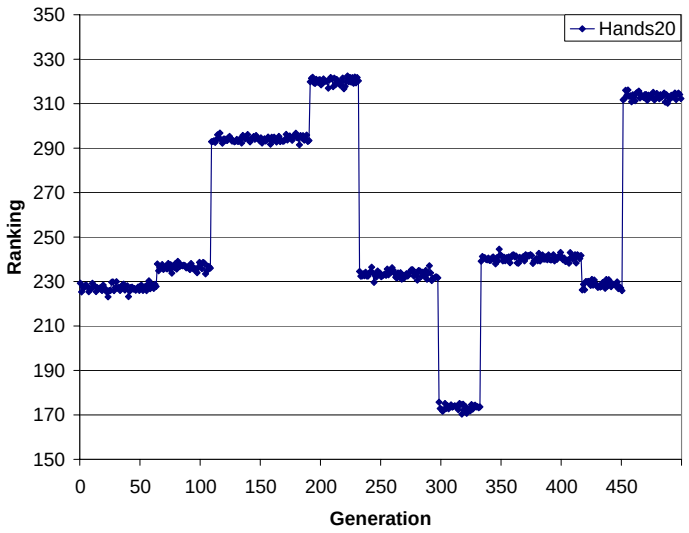

Figure 5.9: Results using best Hands20 agent from each of 500 generations

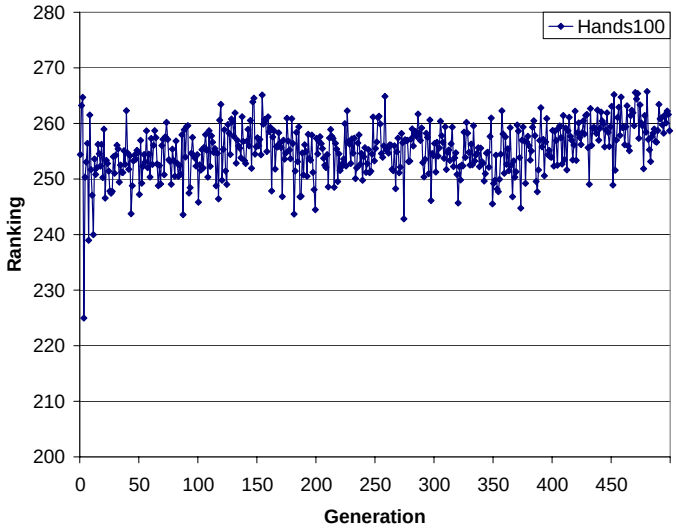

Figure 5.10: Results using best Hands100 agent from each of 500 generations

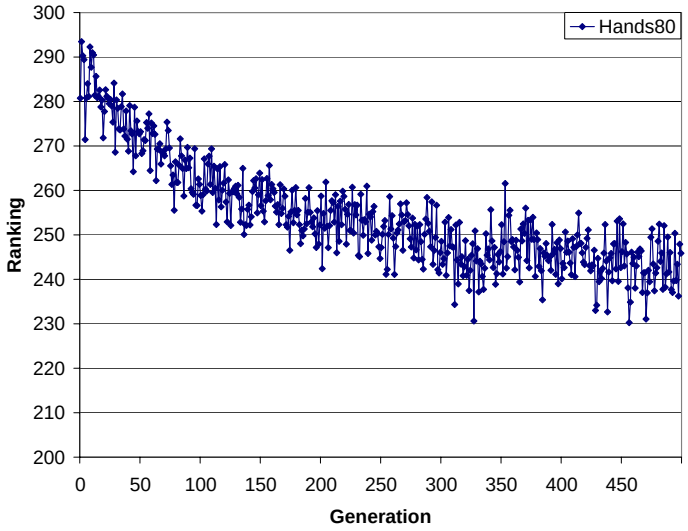

Figure 5.11: Results using best Hands80 agent from each of 500 generations

Figure 5.9 seems to confirm what was suspected for the HandsWon fitness function. Rather than show any progress or any continuity in the graph, the rankings seem to jump about almost randomly. Given that the 20Hands agent was worst of any evolved, it is not a surprising result. When the number of hands won was weighted at 20%, 40% or 60% it seems that a worse agent was unacceptably likely to get promoted.

Figure 5.10 is a little bit surprising. The Hands100 agent was the second worst of the HandsWon agents, but its graph is for the most part continuous and somewhat monotonic. However, instead of increasing in skill, the 100Hands agent actually shows a trend to decrease in skill. Generation 1 is the best agent, at a ranking of 225. The rankings sharply jump to a mean of approximately 250 by the 10th generation. By the 500th generation, the agents have a mean ranking of approximately 260. By itself, the number of hands won does more harm than good, as it randomizes the agents at every generation. Furthermore, section 5.2.2.1 showed that 100Hands performed worse than the Random agent, suggesting that the number of hands won might actually be not just randomizing the agents, but promoting skills that are undesirable.

Figure 5.11 shows the evolutionary progress for the Hands80 agent. This graph is similar to what was expected from a general evolutionary process. As the generations increase, the skill level increases, and as the generations proceed, the agents are getting better, and less likely to be defeated as consistently. As the generations increase, the slope of the graph decreases. The Hands80 agent was the best of the HandsWon agents, and it can possibly be explained by this graph. Unlike the other agents, that showed somewhat random or detrimental progress, the agents of Hands80 improve from one generation to the next.

5.3.2.2 The MeanMoney fitness function

Although the MeanMoney agent did not perform as well as the baseline agent, it performed better than any of the HandsWon agents. The objective of the next experiment was to determine if the MeanMoney agent showed similar evolutionary progress to the Hands80 agent. The top agent from each of 500 generations was inserted into a 500 agent population, and these agents played in an evolutionary progress tournament.

The results of the evolutionary progress tournament for the MeanMoney agents is shown in Figure 5.12. As with the Hands80 evolutionary progress graph, the agents generally show a monotonic improvement after generation 100. The agent does show

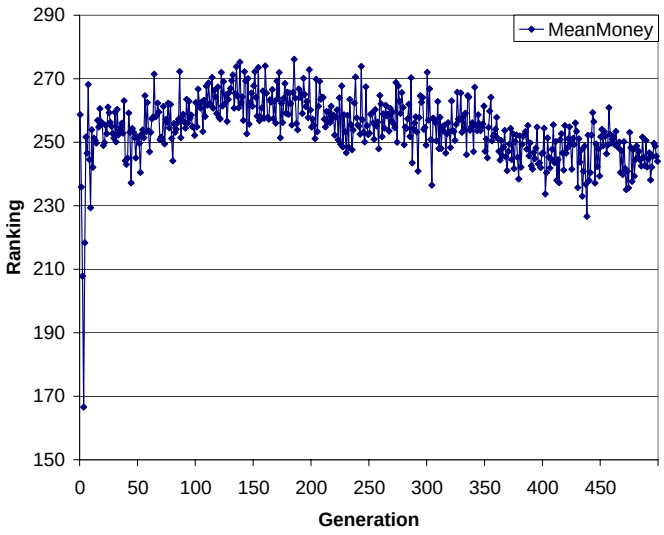

Figure 5.12: Results using best MeanMoney agent from each of 500 generations

progress, but according to Section 5.2.2.2, it is not as good as the baseline agent. The MeanMoney function does seem to promote progress, but the progress is not as fast as occurs with the baseline Control agent in Section 5.3.1.

5.3.3 Evolutionary progress with co-evolution

Generally, the co-evolutionary agents out-performed the baseline agent. For the alternative fitness function agents, it was seen that the best agents showed evolutionary progress (that is, as the generations increased, their average ranking decreased).

5.3.3.1 Two-population agents

The next experiment tests the evolutionary progress of the two-population agents. Two separate evolutionary progress tournaments were conducted, one for 2Pop-1, and one for 2Pop-2. The results of the evolutionary progress tournaments for 2Pop-1 and 2Pop-2 is shown in Figures 5.13 and 5.14, respectively.

The graphs in Figures 5.13 and 5.14 may seem surprising. At first glance, the graphs seem to be almost as random as the graph of Figure 5.9. The graphs have short periods of relative stability (i.e., up to 50 generations where the ranking hardly varies), followed by sharp increases or decreases in skill, followed by more short-term stability. The evolutionary progress graph for 2Pop-1 seems to follow this trend

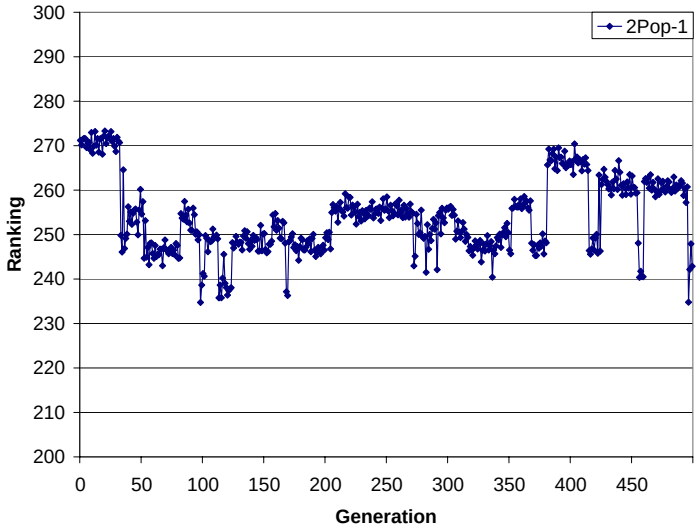

Figure 5.13: Results using best 2Pop-1 agent from each of 500 generations

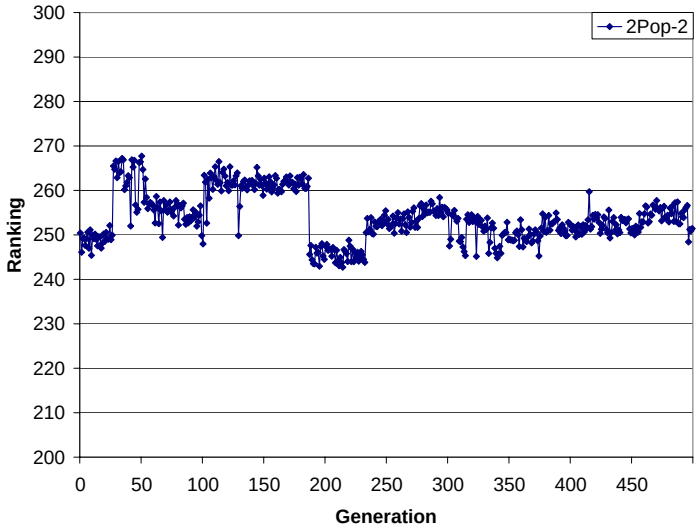

Figure 5.14: Results using best 2Pop-2 agent from each of 500 generations

more than the graph for 2Pop-2, which appears to converge after approximately 240 generations.

One possible explanation for the apparent randomness of the graphs is the "separate, but not independent" nature of the two populations. Although 2Pop-1 and 2Pop-2 are evolved separately, recall that the agents compete in a large population against members from both sub-populations. Although elite agents are selected every generation with respect to their own sub-population, their rankings are achieved through play against both populations. The evolutionary progress tournaments are conducted with members of one population only, and are not representative of the complete process used for selection. It is worth noting that some of the larger jumps on 2Pop-1's evolutionary progress graph, particularly the jumps at generation 40 and generation 210, are preceded by jumps on 2Pop-2's evolutionary progress graph. As agents in 2Pop-2 adopted a new strategy, the agents in 2Pop-1 reacted. When 2Pop-1 is making large jumps in the final 150 generations, 2Pop-2 has already converged. From Table 5.1, 2Pop-2 had a much better final agent than 2Pop-1, and would have had little reason to react to the changes in strategy exhibited by 2Pop-1.

5.3.3.2 Four-population agents

Like the two-population experiments, the evolutionary progress tournaments for 4Pop-1, 4Pop-2, 4Pop-3, and 4Pop-4 are run independently. The results are shown in Figures 5.15 through 5.18.

These graphs are remarkably different from the graphs in Section 5.3.3.1. Where the graphs for 2Pop-1 and 2Pop-2 appeared random and complement each other, all the graphs in Figures 5.15 through Figure 5.18 are generally decreasing in rank (i.e., increasing in skill) as generation increases. Splitting the population up into four sub-populations that are competing against not only one, but several sets of opponents seems to have promoted progress in the evolution. This progress translates to a small increase in the level of the agents (recall that the best agent from the four-population experiments received a rank of 3.65, while the best from the two-population experiments received a 3.73). In all four of the graphs, however, progress seems to have leveled off, suggesting that even if the evolution runs for more generations, no further improvement will be seen.

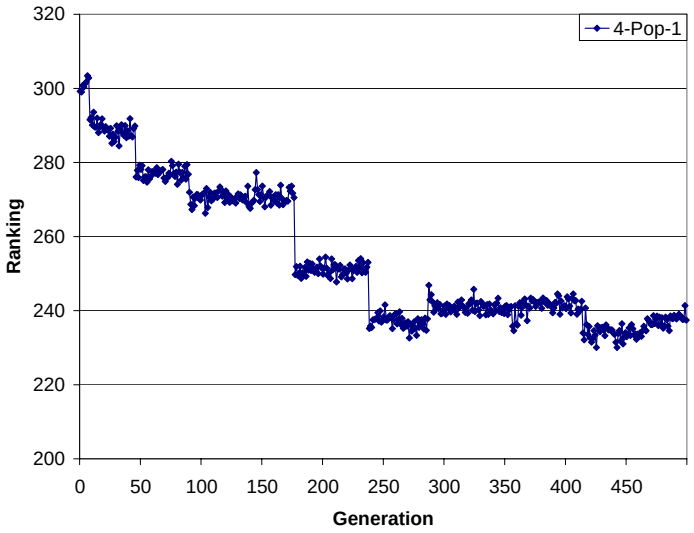

Figure 5.15: Results using best 4Pop-1 agent from each of 500 generations

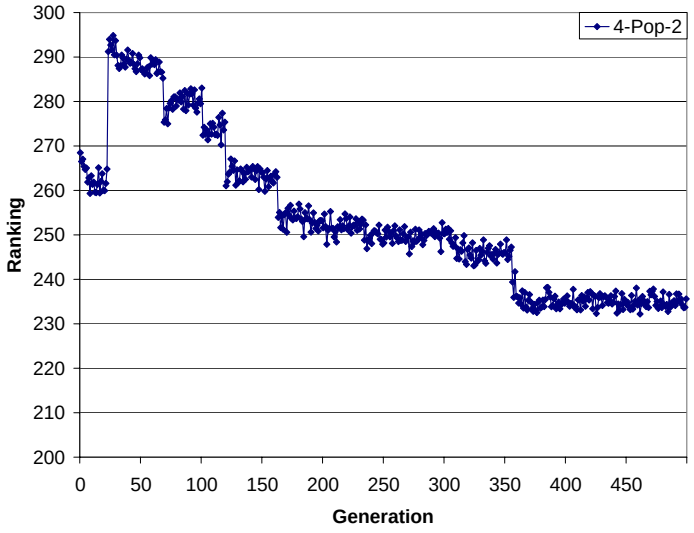

Figure 5.16: Results using best 4Pop-2 agent from each of 500 generations

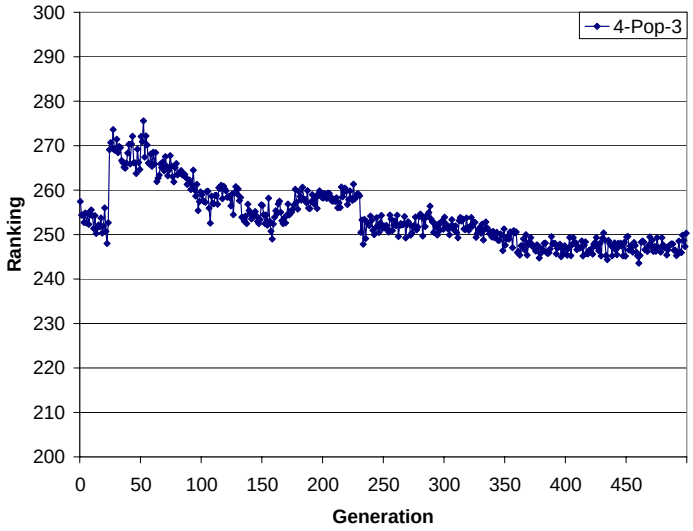

Figure 5.17: Results using best 4Pop-3 agent from each of 500 generations

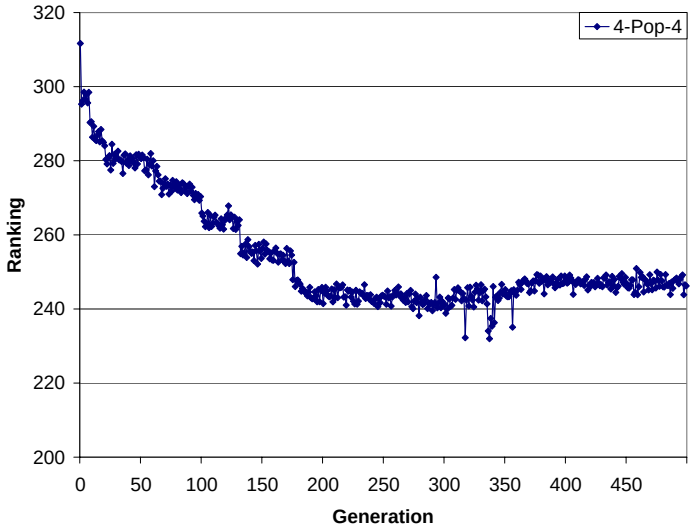

Figure 5.18: Results using best 4Pop-4 agent from each of 500 generations

5.3.4 Evolutionary progress with a hall of fame

The objective of the following experiments was to determine whether the hall of fame aided the evolution of agents. Recall from Section 5.2.4, that the agents evolved using a hall of fame were some of the best agents evolved using any method, and it is expected that these results will be evident in the evolutionary progress graphs.

5.3.4.1 LargeHOF

The first experiment evaluates the evolutionary progress of the agents evolved using a large hall of fame (i.e., LargeHOF). Recall that the LargeHOF agents are evolved for 500 generations using a population of 1,000 agents and a hall of fame of 1,000 additional agents. The LargeHOF agents participate in an evolutionary progress tournament, although a small change is made. Originally, the evolutionary progress of LargeHOF was observed in the same manner as the other experiments (i.e., the representatives participated in 10,000 tournaments). After observing the evolutionary progress graph for LargeHOF, several rather large jumps in skill level were observed. It was reasoned that the jumps seen on the evolutionary progress graph might be less indicative of vast differences in skill, and more indicative of an inability to properly classify the skill levels of the agents. Due to the quality of the LargeHOF agents seen in section 5.2.4, 100,000 tournaments were played, as opposed to the regular 10,000. As agents improve, it can be more difficult to differentiate between them, and thus more tournaments were used to rank the agents.

Figure 5.19 shows the evolutionary progress for the LargeHOF agents. Generally, the graph shows an increase in skill as the generations increase, although there are several locations on the graph that do not follow this trend. From generation 50 to 150, there is a slight decrease in the skill level of the agents. Generation 48 has a rank of 276, while generation 138 has a rank of 310. The rank of agents in consecutive generations are similar to each other; from generation to generation, there is rarely a difference in rank of more than 3. In the evolution phase, the limited games select agents that are occasionally less skilled, as the difference is small.

From generation 200 to 430, the graph is not smooth. After the period of decline in skill in the previous generations, the algorithm has found a section of the decision space that has a very high difference between good and poor strategies. The graph is noisy as different agents are selected as champions. Recall that the HOF experiments evaluated the agents against the best agents from the past. The poorer agents in

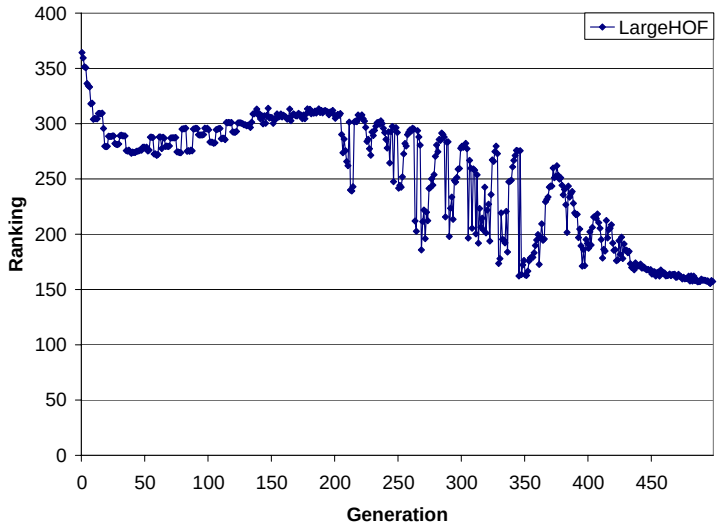

Figure 5.19: Results using best LargeHOF agent from each of 500 generations

this graph may correspond to agents that are skilled against previous agents (which occupy approximately 50% of the total population), but are poorer when they play against the best agents. The better agents in this area are very close to the poorer agents as far as the weightings on the neural networks are concerned, but defeat not only the hall of fame agents, but the other agents in the population as well. The graph has resumed a smooth curve by generation 422. It is also worth noting that the peaks in the noisy section of the graph are steadily decreasing. The agent from generation 260 has a rank of 294, while the agent from generation 266 has a rank of 293, the agent from generation 285 has a rank of 291 and the agent from generation 305 has a rank of 282 (all of these generations are local maxima of the graph). The end agents are, on average, approximately 150 ranks better than those at generation 200. Of further note, the agents slope of the graph over the final 50 generations is still negative (the rank of the agents is still decreasing). Where other agents, such as those in section 5.3.3.2 have leveled off, the LargeHOF agents might benefit from further evolution.

5.3.4.2 SmallHOF

The objective of the next experiment was to evaluate the evolutionary progress of the SmallHOF agents discussed in Section 5.2.4. Recall that these agents were evolved from a population of 500 agents along with a hall of fame of 500 additional agents.

The best agent from each of the 500 generations was inserted into a population of 500 agents, and this population competed in an evolutionary progress tournament. These agents were not as skilled as the agents from the LargeHOF experiment, but since the only heuristic being used is the hall of fame, it is expected that the evolutionary progress will be similar to that seen in Section 5.3.4.1. The results are shown in Figure 5.20.

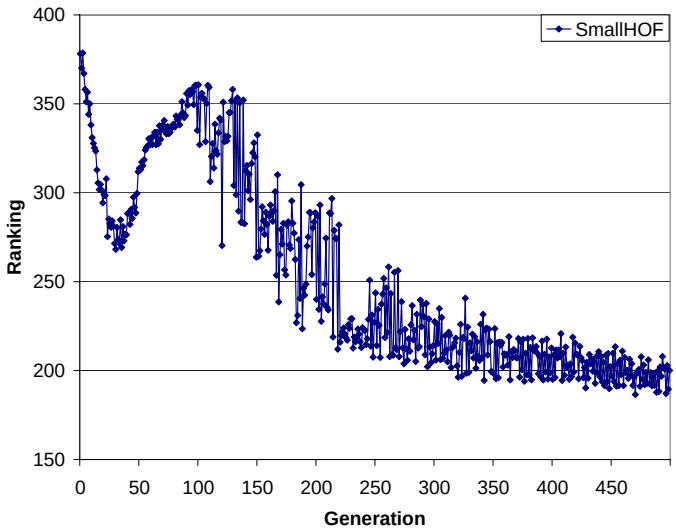

Figure 5.20: Results using best agent from each of 500 generations of SmallHOF agents

Figure 5.20 shows the evolutionary progress of the SmallHOF agents. Although noisy, the graph does seem to show a general improvement of the agents as the generations increase. As with the graph of the LargeHOF agents, there is a decrease in skill approximately 50 to 100 generations into the experiment. This decrease in skill is much sharper than the one observed in LargeHOF (a loss of approximately 75 ranking points over 75 generations, as opposed to a loss of 25 ranking points over 150 generations for LargeHOF). The population of SmallHOF is smaller that that of LargeHOF, and as such, there is a less competitive playing population. A smaller population decreases exploration, and increases the chances that inferior agents are inserted in the elite population, and that these inferior skills are passed on to the next generation.

After the 100th generation, progress is seen, and the agents show a relatively steady increase in skill from generation 100 to 500. Furthermore, as the agents become

more skilled, the graph becomes less noisy, suggesting that the agents are converging upon a plateau of the decision space where any small changes to the agents makes little difference in their skill. The end agents are, on average, approximately 150 ranks better than the agents at generation 100.

5.3.4.3 Evolutionary progress with co-evolution and a hall of fame

The objective of the final experiment was to evaluate the evolutionary progress of the HOF2Pop agents, which were described in Section 5.2.5. Recall that these agents were evolved using two populations of 500 agents apiece, along with separate halls of fame, each of 500 additional agents. The best agents from each of the 500 generations were chosen to represent their respective generation in an evolutionary progress tournament. As with the LargeHOF agents, preliminary testing indicated that 10,000 tournaments might not be enough to differentiate between skilled agents, and thus, 100,000 single tournaments were conducted. The results are shown in Figures 5.21 and 5.22.

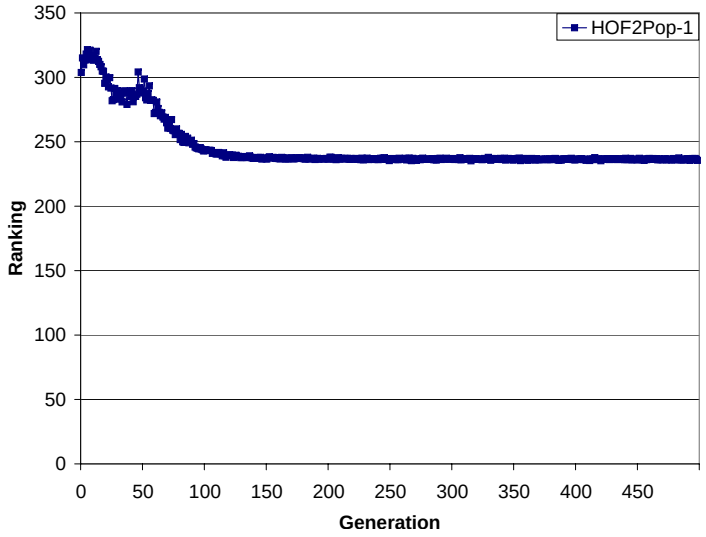

Figure 5.21: Results using best HOF2Pop-1 agent from each of 500 generations

Figures 5.21 and 5.22 are remarkably different. The HOF2Pop-1 agents appear to steadily increase in skill, until they converge at generation 150, and show little improvement beyond that point. As with the other hall of fame agents, there is a period at approximately generation 50 where the agents show a slight decrease in skill, and this early decrease can again be attributed to the fact that the hall of fame

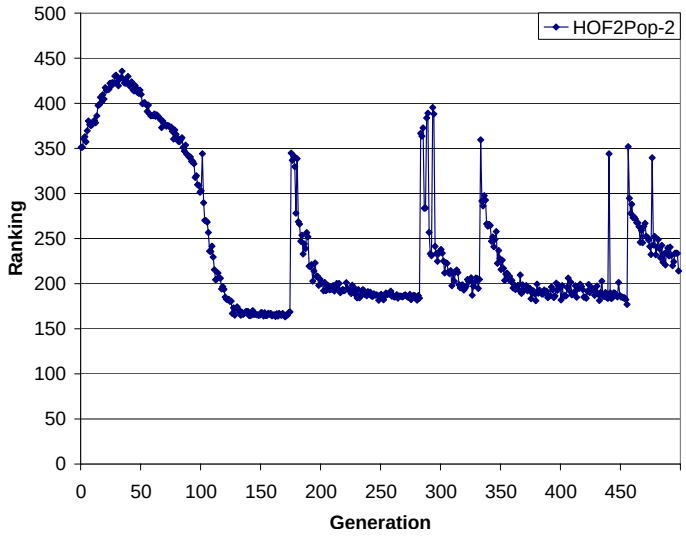

Figure 5.22: Results using best HOF2Pop-2 agent from each of 500 generations

is not full until generation 10. Unlike the other hall of fame agents, the HOF2Pop-1 agents only increase by approximately 60 ranks on average from the beginning to the end of the experiment. This can be attributed to the early convergence of the evolution. By generation 150, the best agents are in the population. In the evolutionary progress experiment, there are 350 agents that are at approximately the same skill level. Section 5.2.5 suggests that these agents are some of the best evolved agents. The evolution converges quite early, and very little progress is achieved after generation 150 (the agent from generation 150 has a rank of 237.1, and the agent from generation 500 has a rank of 235.5). Unlike the LargeHOF and SmallHOF agents seen in sections 5.3.4.1 and 5.3.4.2, respectively, very little would be gained from further generations of evolution.

Figure 5.22 is different from what was expected. Instead of showing a steady improvement in skill, after generation 100, the agents follow cycles of improvement followed by a sharp decrease in skill. Upon further analysis, this cycling can be explained. The first jump in skill occurs at approximately the same generation that the HOF2Pop-1 agents begin to converge. Recall that although the selection procedure is restricted to agents within their respective sub-populations, the fitness of an agent is determined by how it can perform against all agents. After HOF2Pop-1 has converged to one strategy, HOF2Pop-2 tries to find strategies that can defeat the best agents in the competitive population. This may result in strategies that are able to

defeat agents in HOF2Pop-1, but not agents in HOF2Pop-2 (thus accounting for the apparent drops in skill). It is worth noting, that although HOF2Pop-2 is continually showing increases and decreases in skill, its final agents were still some of the best agents evolved, as shown in Section 5.2.5.

Chapter 6

CONCLUSION

This thesis sought to develop No Limit Texas Hold'em playing agents using Evolutionary Neural Networks, and to modify the neural network approach with evolutionary heuristics such as a hall of fame and co-evolution.

This chapter is organized as follows. In Section 6.1, we review the experimental results, and generalize conclusions obtained from those results. In Section 6.2, we suggest possible deficiencies regarding our system, as well as areas whereby the system could be improved.

6.1 Summary

We present the summary in two parts. In Section 6.1.1, we summarize the agents developed in this thesis. In Section 6.1.2, we summarize the experimental results.

6.1.1 Evolving Neural Networks

No Limit Texas Hold'em Poker playing agents were developed using evolving neural networks. A call-based aggressiveness function is implemented and used to model opponents. Both recent and overall aggressiveness are observed and used as input to the neural network. Evolution of agents is promoted by recombination of the neural networks of selected parent agents, and mutation through random noise.

Shortfalls of evolutionary methods suggested by [32, 34] are investigated and solutions are adapted to the game of Texas Hold'em. In [26, 32, 34], co-evolution is explored for 2-player games such as Backgammon and Go. We extend the co-evolutionary heuristic to Texas Hold'em Poker, a game with many more than 2 players. Halls of fame [34] are also implemented, and adapted to Texas Hold'em Poker.

Evaluation of the Poker agents is done using two different methods: duplicate table tournaments, and evolutionary progress graphs. Duplicate table tournaments attempt to remove randomness from the evaluation of agents by repeating cards. The method was inspired by a process used in professional bridge tournaments, and expanded to include nine players at a table. Evolutionary progress graphs attempt to monitor the effectiveness of evolutionary algorithms. Agents are sampled from each generation, and play tournaments against each other. The graphs are an attempt to determine whether agents become more skilled as generations progress.

6.1.2 Experimental Results

Our experimental results allowed us to study the effectiveness of evolutionary methods. Using duplicate table tournaments, it was determined that by themselves, evolving neural networks are capable of learning how to play Poker, although not at the same level as pre-developed agents [3]. Adding co-evolution, agents were evolved that showed an increase in skill over the baseline agents, as well as an increase in skill over previous No-Limit Texas Hold'em agents. Halls of fame led to the greatest increase in skill. Adding co-evolution to the halls of fame saw a slight decrease in skill.

The evolutionary progress graphs showed that the baseline agents were improving, although they tended to converge very early, and showed increases and decreases in skill in rather sharp jumps. Co-evolution, appeared to do little to halt early convergence, although four sub-populations did show that agents were improving as generations increased. Agents evolved with a hall of fame followed a smoother progression, and evolution was slower in converging. Adding co-evolution to the halls of fame sped up convergence of the evolution.

6.2 Future work

There are several areas where the system could be expanded and improved. The hall of fame is used for competition, but not for reproduction. From generation 200 to 400, the evolutionary progress graph of LargeHOF makes large jumps, indicating that at certain generations, agents are less skilled than in previous ones. The skilled agents were still in the hall of fame, but did not prevent this loss in skill. If the hall of fame ever passes the population in skill level, it probably suggests a decrease in

ability on the part of the evolving population, and as such, the population should be restored to a previous generation. In hindsight, this implementation would have been extremely useful, and could have prevented losses as seen in several results.

The neural network could use optimization. The number of hidden nodes and the number of layers in the network were chosen after only minimal consideration. Although the main focus of the thesis was on the addition of the heuristics and the development of players using evolution, and not necessarily the overall skill of the agents, the number of layers and hidden nodes for the network are probably not optimal. Perhaps the networks could be allowed to grow and shrink as necessary, as guided by evolution. If so, testing would be needed to ensure that any gains in the skill levels of evolved agents could be attributed to self-optimisation of the networks, rather than through the evolutionary algorithms themselves.

Many of the agents' decision methods are overly simplistic, and could possibly be improved. The opponent modeling in particular only assigns a relatively restricted value to the opponents. As it stands, an opponent that folds all the time is treated the same as one about whom the agent knows nothing, as both will have an aggressiveness score of 0.

Another possible area for future research is concerned with the fitness function of the algorithm. By necessity, the number of tournaments that are played at any one generation is kept relatively low, at 500 per generation, in order to keep the running times of the simulations down to manageable levels. Although this seems to have produced worthwhile agents, in a game of such high variance, 500 games might not be enough to determine the best players. Some of this bias is overturned by the selection of an elite population instead of a single champion, but it bears investigating.

Other games of imperfect information and large decision spaces present another possible area of study. Games with more than one source of imperfect information (i.e., more than just hole cards of Texas Hold'em) could be investigated. Although most Poker variants restrict hidden information to one source, situations such as silent auctions, where an agent does not know the current high bid, nor the number of competitors, may yield interesting research. Other variants of Poker, such as HORSE, which implements changing rules after set hand intervals, would require robust agents that can change their strategy as the game dictates.

Bibliography

[1] Nolan Bard. Using state estimation for dynamic agent modelling(sic). Master's thesis, University of Alberta, 2008.

[2] Luigi Barone and Lyndon While. An adaptive learning model for simplified poker using evolutionary algorithms. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 153–160, Mayflower Hotel, Washington D.C., USA, 6-9 1999. IEEE Press.

[3] Brien Beattie, Garrett Nicolai, David Gerhard, and Robert J. Hilderman. Pattern classification in no-limit poker: A head-start evolutionary approach. In *Canadian Conference on AI*, pages 204–215, 2007.

[4] D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker, 2003.

[5] D. Billings, L. Pea, L. Pena, J. Schaeffer, and D. Szafron. Using probabilistic knowledge and simulation to play poker. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference*, pages 697–703, 1999.

[6] Darse Billings. *Algorithms and Assessment in Computer Poker*. PhD thesis, University of Alberta, 2006.

[7] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1-2):201–240, 2002.

[8] Darse Billings, Aaron Davidson, Terence Schauenberg, Neil Burch, Michael Bowling, Robert Holte, Jonathan Schaeffer, and Duane Szafron. Game-tree search

with adaptation in stochastic imperfect-information games. In *Computers and Games*, pages 21–34, 2006.

[9] Darse Billings and Morgan Kan. A tool for the direct assessment of poker decisions. Technical report, University of Alberta, 2006.

[10] Darse Billings, Jonathan Schaeffer, and Duane Szafron. Opponent modeling in poker. In *In AAAI National Conference*, pages 493–499, 1998.

[11] Lawrence R. Booker. A No Limit Texas Hold'em Poker Playing Agent. Master's thesis, University of London, 2004.

[12] Murray Campbell, A. Joseph Hoane Jr., and Feng Hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134:57–83, 2002.

[13] Aaron Davidson. Using artifical neural networks to model opponents in Texas Hold'em.

[14] Aaron Davidson. Opponent modeling in poker: Learning and acting in a hostile and uncertain environment. Master's thesis, University of Alberta, 2002.

[15] Aaron Davidson, Darse Billings, Jonathan Schaeffer, and Duane Szafron. Improved opponent modeling in poker. In *Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI'2000)*, pages 1467–1473, 2000.

[16] C. Donninger and U. Lorenz. The hydra Project. *Xcell Journal*, 53:94–97, 2005.

[17] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification.* John Wiley & Sons, Inc., New York, 2001.

[18] David B. Fogel, Timothy J. Hays, Sarah L. Hahn, and James Quon. A self-learning evolutionary chess program. In *Proceedings of the IEEE*, 2004.

[19] Roderich Groß, Keno Albrecht, Wolfgang Kantschik, and Wolfgang Banzhaf. Evolving chess playing programs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2000: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747, San Francisco, California, 2002. Morgan Kaufmann Publishers.

[20] A. Hauptman and M. Sipper. Gp-endchess: Using genetic programming to evolve chess endgame players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, 2005.

[21] Michael Bradley Johanson. Robust stategies and counter-strategies: Building a champion level computer poker player. Master's thesis, University of Alberta, 2007.

[22] Morgan Kan. Postgame analysis of poker decisions. Master's thesis, University of Alberta, 2007.

[23] Graham Kendall and Glenn Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 IEEE Congress on Evolutionary Computation*, 2001.

[24] Cactus Kev. Cactus kev's poker hand evaluator. http://www.suffecool.net/poker/evaluator.html.

[25] Rational Poker School Limited. Pokerstars. http://www.pokerstars.net.

[26] A. Lubberts and R. Miikkulainen. Co-evolving a go-playing neural network. In *Proceedings of the GECCO-01 Workshop on Coevolution: Turning Adaptive Algorithms upon Themselves*, 2001.

[27] J. Von Neumann, D. G. Gillies, and J. P. Mayberry. Two variants of poker. *Contributions to the Theory of Games*, 2, 1953.

[28] John Von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[29] Jason Noble. Finding robust Texas Hold'em poker strategies using pareto coevolution and deterministic crowding. In *Proceedings of the 2002 International Conference on Machine Learning and Applications*, 2002.

[30] M.J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

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

[32] Jordan B. Pollack and Alan D. Blair. Co-evolution in the successful learning of backgammon strategy. *Mach. Learn.*, 32(3):225–240, 1998.

[33] Norman Richards, David E. Moriarty, and Risto Miikkulainen. Evolving neural networks to play Go. *Applied Intelligence*, 8(1):85–96, 1998.

[34] Christopher D. Rosin. *Coevolutionary Search among adversaries*. PhD thesis, University of California, San Diego, 1997.

[35] A. L. Samuel. Some studies in machine learning using the game of checkers, 1959.

[36] A. L. Samuel. Some studies in machine learning using the game of checkers ii - recent progress. *IBM Journal*, pages 601–617, 1967.

[37] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[38] Terence Schauenberg. Opponent modelling and search in poker. Master's thesis, University of Alberta, 2006.

[39] Nicol N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing 6*, pages 817–824. Morgan Kaufmann, San Francisco, 1994.

[40] Brian Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

[41] P.D. Straffin. *Game Theory and Strategy*. Mathematical Association of America, 1993.

[42] Gerald Tesauro. Temporal difference learning of backgammon strategy. In *ML '92: Proceedings of the Ninth International Workshop on Machine Learning*, pages 451–457, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

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

[44] Sebastian Thrun. Learning to play the game of chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 1069–1076. The MIT Press, Cambridge, MA, 1995.