Online Enhancement of existing Nash Equilibrium Poker Agents

Online Verbesserung bestehender Nash-Equilibrium Pokeragenten
Master-Thesis von Suiteng Lu
Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Eneldo Loza Mencía
3. Gutachten: Christian Wirth

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Knowledge Engineering
Prof. Dr. Johannes Fürnkranz

Online Enhancement of existing Nash Equilibrium Poker Agents
Online Verbesserung bestehender Nash-Equilibrium Pokeragenten

Vorgelegte Master-Thesis von Suiteng Lu

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Eneldo Loza Mencía
3. Gutachten: Christian Wirth

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den December 18, 2016

(Suiteng Lu)

Contents

1 Introduction

Since the beginning of artificial intelligence research, games have been a benchmark environment for artificial intelligence techniques due to their well-defined set of rules and distinguishable objectives. Although game domains are typically finite, finding a robust strategy is a non-trivial task. Defeating world class human experts is comparably more difficult as most games are quite complex in nature and exhibit a vast state space. Game-playing agents have already been developed for numerous games such as chess, backgammon, Go and poker. Whereas chess and backgammon AI have already surpassed the skill level of world class human players more than a decade ago, Go and poker are still being actively studied by many researchers. However, there are also recent breakthroughs in both games [34, 35]. The game of poker presents a challenging task because it is a complex large scale **imperfect information** game. Games, such as chess or backgammon belong to the class of **perfect information** games, where the players always have access to all information that define the exact state of the game. In poker, some key information is hidden from the players (for instance the private cards of the opponents), which makes poker more difficult to solve.

The **no-limit Texas hold'em** variant of poker is particularly popular as well as in the AI community. It is a particularly challenging domain as it has on the order of 10^{164} game states [22]. The state of the art techniques for developing robust strategies for poker are based on the concept of finding Nash equilibria. Unfortunately, state of the art equilibrium finding algorithms can only scale up to games with approximately 10^{14} game states[22]. Therefore, significant abstraction and approximation are necessary to apply a game theoretic approach. The results are approximations of equilibrium strategies within the given abstraction. Often, the approximation error is quite significant and the computed strategies are still highly exploitable. The results of the recent first human versus computer no-limit Texas hold'em competition, organized by Carnegie Mellon University [8] shows that the human supremacy is still preserved in the domain of no-Limit Texas hold'em and there remains plenty room for further improvements for the state of the art agents.

However, a heuristic technique called **endgame solving** has proven to be fairly effective during the competition. It can be incorporated into existing Nash equilibrium agents and seems to combat some of the biggest weaknesses all the state of the art Nash equilibrium agents have in common. In this thesis, we investigate endgame solving in no-Limit hold'em and provide a detailed instruction for its implementation which has not been fully covered in the original work [9]. Therefore, compared to [9], this work is enhanced by providing further implementation details and explanations. This includes highlighting the strengths and weaknesses of the original approach, providing possible implementation alternatives as well as a guide for the final essential implementation step of endgame solving which was not further discussed in the original work. Furthermore, we introduce a novel heuristic technique to combine the endgame solver with opponent models to exploit opponents, which we call **endgame exploitation**. Additionally, this technique is capable of online opponent exploitation based on only a few opponent observations. In conclusion, we examine **asymmetric action abstractions** in no-limit Texas hold'em. So far very little research has been devoted to this topic. Most prior works focused on card abstractions and none of them have conducted experiments on the impact of asymmetric action abstractions on the solution quality. In this work, we empirically evaluate the impact of asymmetric action abstraction on agent performance by comparing the performances of agents with different action abstraction designs and granularity, therefore providing guidance for general action abstraction design choices.

This thesis is organized as follows: Chapter 2 gives an overview of the rules and terms of no-limit Texas hold'em, which is our main application domain. Chapter 3 introduces general background knowledge for developing poker agents. It includes surveys of various prior works related to different areas of poker AI that are essential for the understanding of this thesis. Chapter 4 provides an explanation for the use of endgame solving and a description of how to implement endgame solver into existing poker agents. Chapter 5 introduces the endgame exploitation technique and compares the empirical results before and after applying endgame exploitation. Chapter 6 presents our analysis of asymmetric action abstractions. The last chapter 7 briefly summarizes the contributions of this thesis and suggests areas for future research.

2 Poker

Poker is a challenging card game that exhibits many interesting features. It is a stochastic game with imperfect information in a multi-agent environment. There are numerous variants of poker, but they all share a similar set of rules. Before the start of the game, the card deck is shuffled, and cards are randomly dealt to each player. Typically, first dealt cards are considered private information and are only seen by the player holding the respective cards. In the course of the game, more and more cards are dealt, therefore the strength of players' holdings may vary from each round to the next. Due to the stochastic nature of the game, even if a player is holding the best hand at some point during the game, he might still lose at the end due to bad luck.

Since the private cards of other players are hidden, a poker player can only estimate his expected payoff of each action he chooses. This imperfect information property of the game enables deceptive plays such as **bluffs** (pretending to have a stronger hand by aggressive plays) or **slowplay** (pretending to have a weaker hand by passive plays). In fact, these deceptive plays are essential for a successful strategy.

In this chapter, the rules and terms of no-limit Texas hold'em are introduced. The rules and terms have been obtained from [33]. Subsequently, a toy game called Kuhn poker will be presented, which was used in later experiments.

2.1 No-Limit Texas Hold'em

The most popular variant of poker today is called **no-Limit Texas hold'em** (NLH). The game is usually played with two to ten players at a table. This thesis will focus on the two-player variant called Heads Up (HU). A standard deck of 52 cards is used, where the **Ace** has the highest rank followed by **King**, **Queen**, **Jack** and **Ten** down to **Two**. Each player is assigned a position, which shifts after each game. The positions in a Heads Up game are the **small blind** (SB) and the **big blind** (BB). Before the start of the first betting round, each player is obliged to put a certain amount of poker-chips into the **pot**. These investments are called blinds, with the big blind being twice the size of the small blind. The pot is the sum of all investments made by each player within a single game. After each game, all chips in the pot belong to the winner of the game. Due to the blinds in the pot, each player is forced to gamble to avoid going bankrupt by simply paying the blinds in every game, which makes poker a complex game. Otherwise, without these forced investments, the optimal strategy would be trivial, where all players would be able to simply wait for the best possible hand to play. Each player has a finite amount of poker-chips, called a **stack**. The stack size defines the maximum amount a player can wager in each game. On a player's turn, if a player is not faced with a prior bet (or blind), then that player may either:

- **check** - pass the turn to the opponent without investing any chips.
- **bet** - place an amount of chips into the pot. In no-Limit Texas hold'em players are allowed to make unlimited number of arbitrary bets up to his stack size. A bet of the size of a player's stack is referred to as an **all-in**.

When faced with a bet (or blind), a player typically has three options:

- **fold** - forfeit the pot. In this case, the player has surrendered and the opponent immediately wins the pot.

Hand	Definition	Example	Tiebreaker
Straight Flush	Five cards with the same suit and consecutive rank	A♣ K♣ Q♣ J♣ T♣	High Card
Four of a Kind	Four cards with the same rank	A♣ A♢ A♠ A♡ K♣	Rank of Four of the kind; if equal, rank of fifth card
Full House	Three cards with the same rank and a pair	A♣ A♢ A♠ K♡ K♣	Rank of trips; if equal, rank of pair
Flush	Five cards with the same suit	A♣ Q♣ T♣ 8♣ 2♣	High card
Straight	Five cards with consecutive rank	A♣ K♢ Q♠ J♡ T♣	High card
Three of a Kind	Three cards with the same rank, other cards are of different ranks	A♣ A♢ A♠ K♡ Q♣	Rank of trips; if equal, rank of highest other card
Two Pair	Two pairs, fifth card of different rank	A♣ A♢ K♠ K♡ Q♣	Rank of trips; if equal, Rank of high pair; if equal, rank of low pair; if equal, rank of fifth card
Pair	Two cards of same rank, other cards are of different ranks	A♣ A♢ T♠ 9♡ 2♣	Rank of trips; Rank of pair; if equal, rank of highest other card
High Card	No other hand conditions satisfied	A♣ Q♢ T♠ 9♡ 2♣	Rank of trips; if equal, Rank of highest card; if equal, repeat with next highest card

Table 1: Hand ranks in descending order

- **call** - match the last bet by investing the equal amount of chips as the opponent. In case a player has less chips than the opponent's total investment, he can call by placing all his remaining chips into the pot.
- **raise** - increase the previous bet by placing additional chips into the pot after matching the difference in investments. The raise size is also capped by the stack size.

All actions are public information and can be performed in each betting round. There are four betting rounds, also called **streets**.

- **Preflop** - first betting round. Each player is dealt two private cards, which are also called **hole cards**. The hole cards can later be combined with the board cards to form a **hand**. Both players are forced to play their blinds. The player in the small blind position is acting first.
- **Flop** - second betting round. Three public cards are revealed before the players start. Contrast to the preflop, the betting order is inverted for all subsequent betting rounds. Now, the player in the big blind position is always acting first.
- **Turn** - third betting round. Another public card is revealed.

- **River** - final betting round. Again similar to turn, one public card is revealed.

A betting round ends, when every player except the last player to raise has either called the outstanding bet or folded. At the end of the last betting round, all players who did not fold enter the **showdown** and reveal their hole cards. The winner of the game is determined at the showdown by comparing the strengths of each player's hand. A draw is also possible, when both players have equal hand strength. In this case, each player gets his prior investment back, which is also referred to as **split pot**. In any other case, the winner takes the entire pot. The hand strength is determined by the best possible five card combination chosen from the two hole cards and the five board cards. The ranking of hands is depicted in Table 1.

2.2 Kuhn Poker

Kuhn poker [20] is a small toy poker game, where game theoretic solutions can be easily computed. It is a two-player zero-sum imperfect information game as well. The deck consists of merely three cards: King, Queen and Jack. Each player starts with two chips and receives one card from the deck at the start of the game. Prior to any betting, both players must first pay a blind of one chip. Thus, both players have only one chip left for the betting round. There is only one betting round. Players are allowed to either check or to bet. If the betting round ends and none of the players folded, then both players enter the showdown, where the player with higher card wins.

3 Background and Related Work

In this chapter, we start by giving a summary of the fundamentals of game theory and poker specific terminology essential for the thesis. Here, we focus on extensive-form games and two-player no-limit Texas hold'em in particular. Then, we introduce and investigate prior research on techniques for creating strategies in extensive-form games, which we build on in this work. We also describe abstraction techniques for reducing large scale extensive-form games to tractable sizes. Lastly, we present background work on opponent modeling and exploitation techniques which are worth mentioning.

3.1 Extensive-form Games

An extensive-form game represents a game with sequences of actions taken in turn by multiple agents. In each turn, a single player chooses an action that leads the game into a new state. Intuitively, an extensive-form game can be viewed as a rooted game tree. Figure 1 illustrates a sequential presentation of the well known game Rock-Paper-Scissors. Instead of making decisions simultaneously, the first player chooses his move hidden from the second player. Then, the second player also chooses his move and both show their choices at the end. A node of the game tree represents either a chance event (dice roll, hand deal, etc) or a game state. An arc represents the outcome of a chance event or the action taken in a game state. The root node represents the start of the game and the leaves (terminal nodes) represent the end of the game.

Each node can be reached by a particular sequence of actions. Let $\mathcal{N} = \{1..n\}$ denote the set of players. A **history** h is a particular sequence of actions that may occur during the game, starting from the root, and denote $\mathcal{H}$ the set of all sequences, that also include the empty history $\emptyset$. An action sequence from the root to a leaf is called a **terminal history** $z \in \mathcal{Z} \subseteq \mathcal{H}$. Each terminal history is associated with a **utility** (payoff or reward) $u_i(z)$, where $u_i : \mathcal{Z} \to \mathbb{R}$ is an utility function of player$_i$. In case of two-player zero-sum game, for all $z \in \mathcal{Z}$, $u_1(z) = -u_2(z)$. A history h' is a prefix of h, if h starts with h'. A direct successor of h can be written as ha that represents the history h after taken action a. Thus if $h = h'a$, then h' is the prefix of h, and h is a successor of h' which is denoted with $h' \sqsubseteq h$ [5, p.5].

At each non-terminal history, let $A(h)$ denote the set of possible actions the acting player $P(h)$ can choose from. The function $P : \mathcal{H}/\mathcal{Z} \to \mathcal{N} \cup \{c\}$ is a player function that determines whose turn it is to act at a particular history. Besides all players $n_i \in \mathcal{N}$, $P(h) = c$ indicates that chance is on the turn, where chance selects action a according to a known fixed probability [5, p.5].

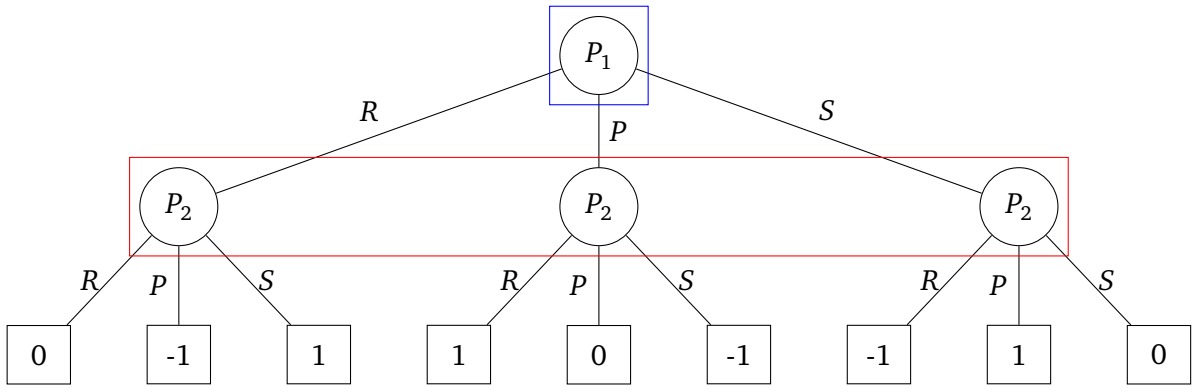

Figure 1: Rock-Paper-Scissors represented as a tree. Leaves represent utility of player$_1$. Blue and red bounding boxes represent information sets of player$_1$ and player$_2$ respectively.

An **information set** $I \in \mathcal{I}_i$ is a set of histories that cannot be distinguished by the player$_i$ due to the imperfect information of the opponents. Thus, each game state within the same information set is treated equally by the acting player. In perfect information games, all information sets consist exclusively of one particular history, since the players can observe all information. However, in imperfect information games, multiple histories are grouped into the same information set. For any two histories h,h' in the same information set, $A(h)$ must equal $A(h')$. Thus, the actions available at an information set is simply denoted as $A(I)$. Also $P(I)$ means $P(h)$ such that $h \in I$ [5, p.5-6]. For instance, in Figure.1, there are two information sets, one is the empty history belonging to player$_1$, and one containing $\{R, P, S\}$ belonging to player$_2$.

A two-player extensive-form game has **perfect recall** if players do not forget any moves that were made by all players including chance during the course of the game. Hence, there is only a single path to every node. Formally, let $X_i(h) = ((I, a), (I', a')...)$ denote the sequence of player$_i$'s information set action pair that were encountered and taken to reach h in the same order as they were during game-play. A two-player extensive-form game has perfect recall if $\forall I \in \mathcal{I}_i : h, h' \in I \Rightarrow X_i(h) = X_i(h')$ [13, p.11]. Otherwise, it is called **imperfect recall**, where several paths can lead to the same node and a player can forget previous actions or chance events. A practical example of perfect and imperfect recall can be found later in chapter 3.3.2.

A **behavioral strategy** is a function that assigns a distribution over actions to each information set. During the course of a game, each player selects actions according to his strategy. More precisely, a strategy σ_i for player$_i$ is a probability distribution over all possible actions $a \in A_i$ at a particular information set [5, p.6]. For example, if $A_i = \{Rock, Paper, Scissor\}$, a valid probability distribution could be

$$\sigma_i(a) = \begin{cases} 0.5, & \text{if } a = Rock \\ 0.3, & \text{if } a = Paper \\ 0.2, & \text{if } a = Scissor \end{cases} \tag{1}$$

Note that strategies are only assigned to information sets, which means that all game states within the same information set follow the same strategy. A **strategy profile** $\sigma = (\sigma_1 ... \sigma_{|\mathcal{N}|})$ is a collection of strategies, one for each player.

There are two types of strategies, **pure strategies** and **mixed strategies**. A pure strategy always selects a single action at a choice node, whereas a mixed strategy defines a probability distribution over possible actions [13, p.10]. For example, (1) is a mixed strategy and (2) is a pure strategy that only plays rock.

$$\sigma_i(a) = \begin{cases} 1, & \text{if } a = Rock \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Assume a two-player game, where a strategy profile $\sigma = (\sigma_1, \sigma_2)$ is used. The **sequence probability** $\pi^\sigma(h)$ of a given history h is the probability of reaching h if all players acted according to σ. Formally, the sequence probability is defined as

$$\pi^\sigma(h) = \prod_{h'a \sqsubseteq h} \sigma_{P(h')}(a|h')$$

where $\pi^\sigma(h)$ is the product of probabilities of each player taking his actions in the sequence h according to σ [5, p.6]. For example, in Figure 2, the marked edges constitute a terminal history z with the utility

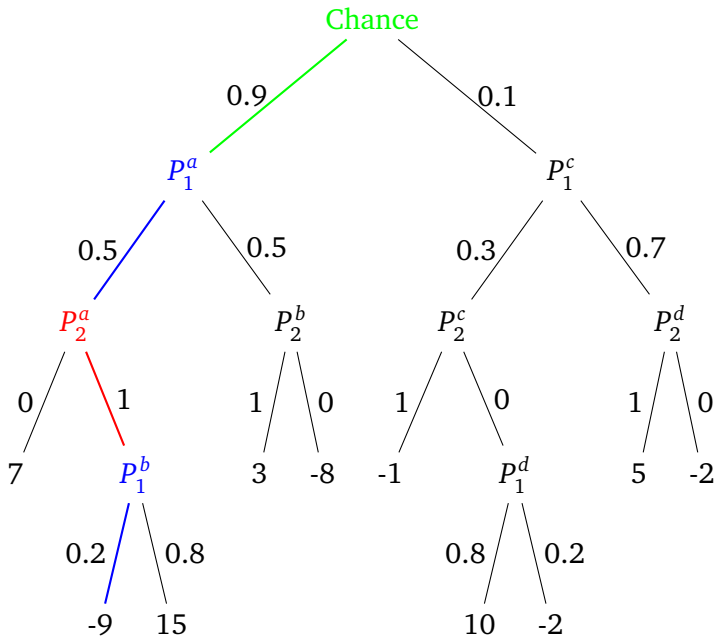

Figure 2: A random two-player extensive-form game. Player$_2$ plays a pure strategy and player$_1$ plays a mixed strategy. Terminal nodes represents the payoffs for player$_1$.

$u_1(z) = -9$. The labels on the edges represent the probability distribution induced by σ. Thus, the sequence probability $\pi^\sigma(z) = 0.9 \cdot 0.5 \cdot 1 \cdot 0.2 = 0.09$.

The product can be decomposed into individual sequence probability for each player and chance: $\pi_1^\sigma(h)$, $\pi_2^\sigma(h)$ and $\pi_c^\sigma(h)$. In other words,

$$\pi^\sigma(h) = \pi_1^\sigma(h) \cdot \pi_2^\sigma(h) \cdot \pi_c^\sigma(h)$$

In Figure 2, the decomposed sequence probabilities of the terminal history z are $\pi_1^\sigma(z) = 0.5 \cdot 0.2 = 0.01$, $\pi_2^\sigma(z) = 1$ and $\pi_c^\sigma(z) = 0.9$. The **expected utility** for player$_i$ under strategy profile σ can be computed by a game tree traversal that computes the sum

$$u_i(\sigma) = u_i(\sigma_i, \sigma_{-i}) = \sum_{z \in Z} \pi^\sigma(z) u_i(z)$$

where σ_{-i} denote the strategies in strategy profile σ excluding σ_i. In our example, $u_1(\sigma) = (0.09 \cdot -9) + (0.36 \cdot 15) + (0.45 \cdot 3) + (0.03 \cdot -1) + (0.07 \cdot 5) = 6.26$.

In game theory, there are generally two basic strategy types: **best response** and **nash equilibrium**. In two-player zero-sum game, a best response for player$_1$ is a strategy that yields the greatest possible payoff against the strategy of the player$_2$. The expected utility of a best response for player$_1$ is thus defined as

$$b_1(\sigma_2) = max_{\sigma_1' \in \sum_1} u_1(\sigma_1', \sigma_2)$$

where $\sum_i$ denotes the set of all possible strategies of player$_i$. Nash equilibrium is a special case of best response, where the strategies for both players represent best responses to each other at the same time, which means that no player would gain more by deviating from his current strategy. Therefore, if

player$_1$ deviates from his Nash equilibrium strategy, as a result, he would lose ε expected utility. Formally, $min_{\sigma_2' \in \sum_2} u_1(\sigma_1, \sigma_2') + \varepsilon = b_1(\sigma_2)$. In other words, player$_1$ has become **exploitable** by an amount ε. A strategy profile which can be exploited by an amount ε at most is called ε**-Nash equilibrium**, which means that the strategy profile σ would lose ε playing against a worst-case opponent averaged over all strategies in σ. Formally for two-player zero-sum game, it is $\varepsilon_\sigma = (b_2(\sigma_1) + b_1(\sigma_2)/2$. Since Nash equilibria have an exploitability of zero, computing nash equilibria is also referred to as *solving* the game [5, p.7].

3.2 Finding Nash Equilibrium

A Nash equilibrium is a robust, static strategy which provides a guarantee about its expected utility against a worst-case opponent. Usually, a set of strategies is referred to as "*in equilibrium*", if neither of the strategies could improve its expected utility by deviating from it. Thus, an agent who plays according to a Nash equilibrium would never lose on expected utility and would be safe from any exploitation attempts by its opponent. Moreover, if the opponent were to deviate from a Nash equilibrium, the player might in fact win money [29]. For that reason, finding Nash equilibria represents the main part of Game-AI development.

In this section, we describe different approaches to computing Nash equilibria. First, we introduce a traditional technique which is based on solving linear optimization problems. This approach computes an exact equilibrium given the LP formulation of the game. Then, we describe a entirely different approach that approximates a Nash equilibrium by repeatedly modifying strategies over time. The idea is to incrementally improve the current solution until it converges to a Nash equilibrium. The best-known and successful ϵ-Nash equilibrium approximation algorithm is the **Counterfactual Regret Minimization**, which we will focus on in the last subsection.

3.2.1 LP Approach

Extensive-form games can be converted into a system of linear equations called **Linear Programming (LP)**, which can be solved in polynomial time using conventional LP-algorithms [19]. Linear programs are problems that can be expressed in canonical form as

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\mathbf{T}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \quad \geq 0 \end{aligned} \tag{3}$$

where $\mathbf{x}$ represents the vector of variables that needs to be determined, $\mathbf{c}$ and $\mathbf{b}$ are vectors of known coefficients, and $\mathbf{A}$ is a known matrix of coefficients. The goal is to find values for $\mathbf{x}$ that maximize or minimize the objective function ($\mathbf{c}^\mathbf{T}\mathbf{x}$ in (3)) under some linear constraints (such as the inequities $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ and $\mathbf{x} \geq 0$ in (3)) [26, p.41].

Every linear program can be expressed in two ways, as the **primal** or the **dual** problem (duality principle). The dual problem to (3) can be formulated as

$$\begin{aligned} \underset{\mathbf{y}}{\text{maximize}} \quad & \mathbf{b}^\mathbf{T}\mathbf{y} \\ \text{subject to} \quad & \mathbf{A}^\mathbf{T}\mathbf{y} \leq \mathbf{c} \\ & \mathbf{y} \quad \leq 0 \end{aligned} \tag{4}$$

The solution to the dual problem provides a lower bound to the solution of the primal problem. For convex optimization problems, the solutions to the primal and the dual problem are equal, if the strong duality condition is satisfied [26, p.51]. In the case of two-player zero-sum games, the primal solutions correspond to strategies for one player while the dual solutions correspond to the strategies of his opposition.

Traditionally, for solving any extensive-form game, a normal form of the game is constructed first, where all possible pure strategy pairs of the players are tabulated, along with the expected payoff for each player when such a strategy pair is played. Nevertheless, this method results in an exponential increase in the size of the problem. Prior work introduced an alternative LP representation, called **sequence form**, which avoids the exponential increase associated with the normal-form [19].

The sequence-form is based on a different representation of the strategic variables. Player$_i$'s strategy σ_i is encoded using a **realization plan**, $\beta_i \in \mathcal{B}_i \subseteq \mathbb{R}^{\sum_{I \in \mathcal{I}_i} |A(I)|}$, that stores the strategy's **realization weights** [5, p.10]. Due to perfect recall, each information set $I \in \mathcal{I}_i$ uniquely identifies a sequence of all previous information sets of player$_i$ and the corresponding actions taken by player$_i$ to reach I. Given some strategy σ_i, let $\sigma_i(a|I)$ be the probability of taking action a at I. A realization plan is defined as

$$\beta_i(I, a) = \sigma_i(a|I)\pi_i^{\sigma}(h), h \in I$$

In other words, the realization weight of a sequence of player$_i$ is the product of all action probabilities within the sequence according to the player's current strategy ignoring the chance and the other player's moves. For example, considering the information set at P_1^b in Figure 2, the realization weight for the blue marked action is $\beta_1(P_1^b, left) = 0.2 \cdot 0.5 = 0.1$.

The objective is to find an optimal realization plan (a vector containing all realization weights). A valid realization plan must satisfy the following three constraints:

$$\beta_i(I, a) \geq 0 \qquad \text{(non-negative weights)}$$

$$\sum_{a \in A(I)} \beta_i(I, a) = \beta(I', a') \qquad \text{(chidren sum to parent)}$$

$$\beta_i(\emptyset) = 1 \qquad \text{(root has weight 1)}$$

where (I', a') is a parent of (I, a) or $\emptyset$ if it has no parent. The second constraint indicates that the sum of all realization weight of sequences coming out from a certain information set is equal to the realization weight of the parent sequence that led to this information set [5, p.10].

A strategy can be extracted from every valid realization plan by normalizing the realization weight of a information set action pair by the realization weight of the parent. Formally, it is accomplished by

$$\sigma_i(a|I) = \frac{\beta_i(I, a)}{\beta_i(I', a')} = \frac{\beta_i(I, a)}{\sum_{a' \in A(I)} \beta_i(I, a')}$$

[5, p.10]. For example, in Figure 2:

$$\beta_1(P_i^b, left) = 0.5 \cdot 0.2 = 0.1$$
$$\beta_1(P_i^b, right) = 0.5 \cdot 0.8 = 0.4$$
$$\beta_1(P_i^a, left) = 1 \cdot 0.5 = 0.5$$
$$\sigma_i(left|P_i^b) = \frac{0.01}{0.1 + 0.4} = 0.2$$

For a two-player zero-sum game, assume $\mathbf{x}$ and $\mathbf{y}$ are the realization plans β_1 and β_2 for player$_1$ and player$_2$ in vector notation. $\mathbf{A}$ denotes the payoff matrix of player$_1$ and $\mathbf{B} = -\mathbf{A}$ denotes the payoff matrix of player$_2$. The Nash equilibrium profile $(\mathbf{x}, \mathbf{y})$ can be obtained by solving the optimization problem:

$$max_{\mathbf{x}\in\mathcal{B}_1} min_{\mathbf{y}\in\mathcal{B}_2} \mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{y} = min_{\mathbf{y}\in\mathcal{B}_2} max_{\mathbf{x}\in\mathcal{B}_1} \mathbf{x}^\mathbf{T}\mathbf{A}\mathbf{y}$$

under the constraints for valid $\mathbf{x}$ and $\mathbf{y}$ derived from above [5, p.11].

A linear program can be created from this optimization problem by taking the dual of the inner minimization/maximization. The inner optimization taken in isolation computes a best response. For example, the maximization problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximize}} \quad & \mathbf{x}^\mathbf{T}(\mathbf{A}\mathbf{y}) \\ \text{subject to} \quad & \mathbf{E}\mathbf{x} = \mathbf{e} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{5}$$

computes a best response $\mathbf{x}$ for player$_1$ to $\mathbf{y}$ of player$_2$. Let $\mathcal{C}_i$ denote the set of all choices of player$_i$ $\bigcup_{I\in\mathcal{I}_i} A(I)$. $\mathbf{E}$ represents the constraint matrix of player$_1$, which has the size of $(|\mathcal{I}_1| + 1) \times (|\mathcal{C}_1| + 1)$. Therefore, $\mathbf{x}$ has $|\mathcal{C}_1| + 1$ variables that represent the realization weight of each sequence including the empty sequence $\emptyset$. The constraint matrix ensures that the two constraints for realization plans are satisfied (chidren sum to parent, root has weight 1). The payoff matrix $\mathbf{A}$ has the size of $(|\mathcal{C}_1|+1)\times(|\mathcal{C}_2|+1)$ and has at most $|Z|$ non-zero entries which represents the number of terminal histories. $\mathbf{E}$ and $\mathbf{A}$ are both sparse, thus keeping the linear program in linear size of the game tree.

The dual of (5) is defined as follows:

$$\begin{aligned} \underset{\mathbf{p}}{\text{minimize}} \quad & \mathbf{e}^\mathbf{T}\mathbf{p} \\ \text{subject to} \quad & \mathbf{E}^\mathbf{T}\mathbf{p} \geq \mathbf{A}\mathbf{y} \end{aligned} \tag{6}$$

Each equation in the constraints is expanded with a new unconstrained variable, yielding a vector $\mathbf{p}$ with the dimension of $|\mathcal{I}_1| + 1$. As a result, combined with the outer optimization problem, the following linear program (7) can be formed, whose solution is a part of a Nash equilibrium in the given zero-sum game [19, p.5].

$$\begin{aligned} \underset{\mathbf{y},\mathbf{p}}{\text{minimize}} \quad & \mathbf{e}^\mathbf{T}\mathbf{p} \\ \text{subject to} \quad & \mathbf{E}^\mathbf{T}\mathbf{p} - \mathbf{A}\mathbf{y} \geq 0 \\ & \mathbf{F}\mathbf{y} = \mathbf{f} \\ & \mathbf{y} \geq 0 \end{aligned} \tag{7}$$

The dual of (7) is defined as:

$$\begin{aligned} \underset{\mathbf{x},\mathbf{q}}{\text{maximize}} \quad & \mathbf{f}^\mathbf{T}\mathbf{q} \\ \text{subject to} \quad & \mathbf{F}^\mathbf{T}\mathbf{q} - \mathbf{A}^\mathbf{T}\mathbf{x} \leq 0 \\ & \mathbf{E}\mathbf{x} = \mathbf{e} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{8}$$

The solution to (8) is the other part of the same Nash equilibrium. Optimal solutions $(\mathbf{y}, \mathbf{p})$ and $(\mathbf{x}, \mathbf{q})$ to (7) and (8) fulfill $\mathbf{e}^\mathbf{T}\mathbf{p} = \mathbf{f}^\mathbf{T}\mathbf{q}$ by strong duality and are therefore equilibrium strategies of the game [19,

p.5]. However, only the primal variables **x** and **y** are needed that represent the optimal realization plans for each player.

There are several existing algorithms for solving LPs. The most common one is the **Simplex Algorithm** [26, p.61-92]. It solves the LP by constructing a feasible solution at a vertex of a polytope that defines the feasible region of the LP. It has been proven that an optimal solution must be located at one of the vertices. The simplex algorithm then follows a path on the edges of the polytope to vertices with non-decreasing values of the objective function until an optimum is reached. In practice, the simplex algorithm is quite efficient despite its exponential worst-case behavior. Another noteworthy algorithm for solving LP is the **interior point algorithm** [26, p.97-114]. In contrast to the simplex algorithm, it moves through the interior of the feasible region rather than on the edges. One of the current most commercially successful implementation of the above mentioned LP-algorithms is the **Gurobi Optimizer** [16]. It is capable of using both algorithms for solving by dynamically switching from one to another during runtime to achieve maximum efficiency. However, the memory required to solve such sequence-form LPs makes them rather impractical for large games such as two-player no-limit Texas hold'em.

Despite the memory limitation, LP techniques have been successfully applied in smaller games or abstractions of larger games. Prior works [14, 7] used the LP-Approach on a strongly abstracted game of limit Texas hold'em. Recently, Ganzfried et al.(2013)[9] applied LP on endgames of no-limit Texas hold'em, which will be investigated further in chapter 4.

3.2.2 Counterfactual Regret Minimization

Despite the fact that the sequence-form allows much larger games to be solved compared to the traditional normal-form, it still fails in representing and solving large-scale games such as full-scale Texas hold'em owing to the enormous memory requirements. Prior work indicated that the LP approach only scales to games with around 10^8 nodes in their game tree [14], while newer approximation techniques scale to games with around 10^{13} nodes.

The leading state of the art algorithm for computing ε-Nash equilibrium in two-player zero-sum games is **Counterfactual Regret Minimization (CFR)**[40]. It only requires a memory linear in the size of the information sets instead of game states. Instead of storing the entire extensive-form game tree in the memory, it solely stores an information set tree for each player. Each node in an information set tree corresponds to a complete information set within the extensive-form game tree. During the computation of the strategy, the algorithm is only required to traverse the extensive-form game tree, while the strategy itself is stored in the information set trees.

The CFR algorithm is a regret minimizing algorithm. The **regret** for player$_i$ is defined as

$$R_i^T = \max_{\sigma^* \in \sum_i} \sum_{t=1}^{T} (u_i(\sigma^*, \sigma_{-i}^t) - u_i(\sigma_i^t, \sigma_{-i}^t))$$

R_i^T measures the amount of utility player$_i$ could have gained by following the best strategy in hindsight instead of following σ_i. The **average regret** is defined as R_i^T/T. An algorithm is regret minimizing if the overall average regret converges to zero over time ($\lim_{T\to\infty} \frac{R_i^T}{T} = 0$) [5, p.13]. The difference between

$$\max_{\sigma_i^* \in \sum_i} (u_i(\sigma_i^*, \sigma_{-i}) - u_i(\sigma))$$

quantifies player$_i$'s regret for playing his current strategy compared to the strategy with the highest expected utility.

However, the CFR algorithm does not minimize one overall regret value, but rather disperses regret values into separate information sets while minimizing individual regret values. Each individual regret is associated with an information set and is therefore weighted by the probability of reaching this particular information set. The dispersed regrets are referred to as **counterfactual regrets**. Formally, a counterfactual regret is defined as

$$r(I,a) = u_i(\sigma_{I\to a}, I) - u_i(\sigma, I)$$

$u_i(\sigma, I)$ is called **counterfactual utility** and represents the expected utility, given information set I is reached and all players play according to σ. $u_i(\sigma_{I\to a}, I)$ denote the utility of playing according to σ, except the player$_i$ who uses a pure strategy of always choosing action a given $a \in A(I)$. Thus, $r(I,a)$ describes the degree to which player$_i$ desires having played action a instead of following the current strategy σ. A high regret value would therefore indicate that the player should pick action a more often. Therefore, he should adjust his current strategy in favor of action a. The adjusted strategy has a probability distribution proportional to the accumulated positive value of the regret. If a particular action had a high regret in the current iteration, it would be assigned a higher probability, hence the action would be used more often in the next iteration. As the number of iterations increases, the average counterfactual regret at each information set approaches zero as well.

The sum of counterfactual regret from each information set provides a bound on the overall regret, which when minimized gives a strategy that approaches a Nash equilibrium [40]. Hence, the strategy profile associated with the average overall regret, called average strategy, converges to a Nash equilibrium profile. It has been proven that CFR converges to a Nash equilibrium in two-player zero-sum perfect recall games [40]. However, CFR has also been successfully applied in multiplayer games [30] and imperfect recall games [37] as well.

Figure 3 depicts a part of Figure 2 and illustrates how counterfactual regrets are calculated at a single node in an extensive-form game tree. At P_1^b, player$_1$'s current strategy is $(left, right) = (0.2, 0.8)$. The expected value for player$_1$ given his current strategy is

$$0.2 \cdot (-9) + 0.8 \cdot 15 = 10.2$$

For each action a at this node, player$_1$'s regret for not taking action a is the difference between the expected value for taking this action and the expected value given player$_1$'s current strategy:

$$\begin{aligned} left &: -9 - 10.2 = -19.2 \\ right &: 15 - 10.2 = 4.8 \end{aligned}$$

The results indicate that player$_1$ regrets not taking the $right$ action more often than the $left$ action. Then, the values need to be weighted by the probability of actually reaching P_1^b (multiply all probabilities on the red edges):

$$\begin{aligned} weight &: 0.9 \cdot 0.5 \cdot 1 = 0.45 \\ left &: -19.2 \cdot 0.45 = -8.64 \\ right &: 4.8 \cdot 0.45 = 2.16 \end{aligned}$$

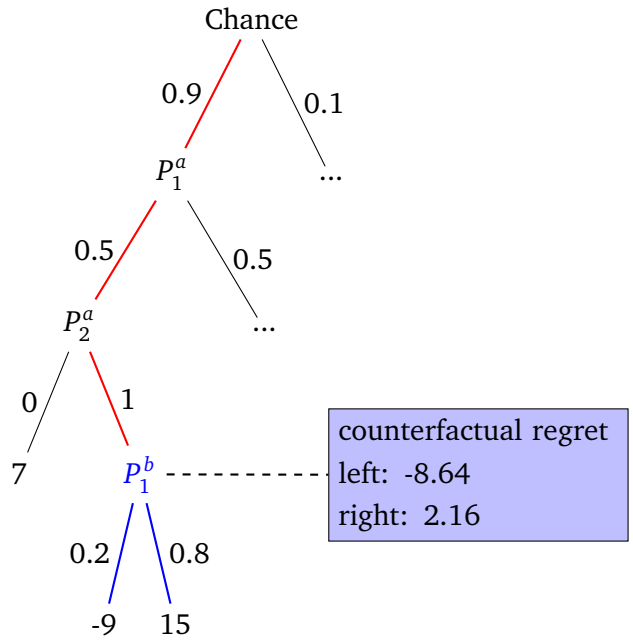

Figure 3: A game tree highlighting the regret calculation at the blue marked node.

Now, player$_1$ will adjust his strategy proportional to the positive values of the regrets. The adjusted strategy is now a pure strategy that will always pick the $right$ action.

$$left : \frac{\max(0,-8.64)}{\max(0,-8.64)+\max(0,2.16)} = 0$$
$$right : \frac{\max(0,2.16)}{\max(0,-8.64)+\max(0,2.16)} = 1$$

3.3 Abstraction

Two-player no-Limit Texas hold'em, played with a stack size of 200 big blinds, a big blind of the size of 100 chips and with only integral bet sizes allowed, is a game with about 6.31×10^{164} game states and 6.37×10^{161} information sets [22]. Solving this game using a standard CFR implementation with a loss-free compression technique (which will be described in the following section) would require 1.094×10^{138} yottabytes of RAM [22], which is far too large to be solved in its complete form on modern hardware. Hence, the game and information set trees must be scaled down to a tractable size. The most common approach for this matter is to abstract the state space. In the domain of poker, we consider **card abstraction** and **action abstraction**, which will be explained later.

The abstracted version of the game is solved with an equilibrium finding algorithm. Then, the solution is mapped back to the original game by translating the actions of the abstracted game solution back into the real game. This process is illustrated in Figure 4. Typically, the state space abstraction is carried out prior to computing a strategy.

Although there are techniques such as **card isomorphism** which guarantee loss-free state space reduction [38], in order to produce a tractable abstracted game, at least some loss of information is unavoidable. The quality of the solution in the real game depends on the size of the abstracted game and the abstraction techniques used to produce it. Only if the strategical properties of the real game are preserved in the abstracted game, the solution will produce a good approximation for the full game.

Usually, increasing the size of the abstracted game would reduce the loss of information, but also increases the time and memory requirements for the strategy computation. In most cases, solving larger abstracted games results in less exploitable strategies. However, Waugh et al.(2009)[36] established that this is not always the case and provided examples in toy games, where increased abstraction size even made the resulting strategy more exploitable.

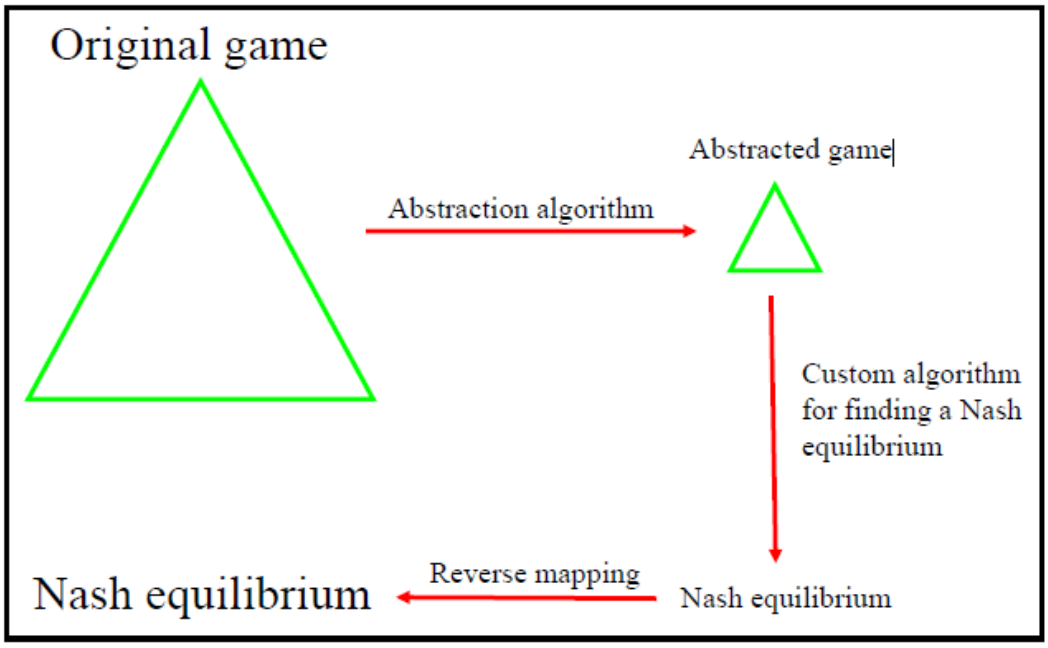

Figure 4: state of the art approach for solving large-scale games [10, p. 3]

3.3.1 Isomorphism

In Texas hold'em and as well as in many other card games, certain card combinations are strategically equivalent to each other. This is the case, when only the rank of a card and not its suit determines its strength. Hands that are strategically equivalent and only differ by a rotation of their suits are called **isomorphic** [38]. In the first betting round for instance, the hole cards A♣K♣, A♢K♢, A♡K♡ and A♠K♠, in other words all Ace-King combinations with both cards of the same suit, are isomorphic to each other. One of those four combinations can be arbitrarily chosen as representative of the set. This representative is called **canonical**. An equilibrium-finding algorithm therefore simply needs to find the strategy for the canonical hand. Since every non-canonical hand is isomorphic to a canonical hand, the strategy of the canonical hand can later be used for all hands in the set of isomorphic hands. This reduces the state space by a factor of up to 4!, which, however, is not yet sufficient to be able to solve a game as large as no-Limit Texas hold'em.

Isomorphism is a loss-free abstraction technique. Since the merged states are strategically equivalent, this reduction will not lead to a worse result compared to the solution obtained from the full game tree.

3.3.2 Card Abstraction

A well-known technique for state space reduction is **bucketing** [23, p.23-24]. In each betting round, the cards of each player are mapped down to a certain number of buckets, with the intention of grouping

hands with similar strategic properties into the same bucket. Similar to card-isomorphism, bucketing can be regarded as a many-to-one mapping from the information sets of the full game to information sets in the abstracted game as well with the difference that bucketing also merges strategically similar (not necessarily equivalent) card combinations.

In order to group the cards, it is necessary to first extract distinguishing features from the cards to represent the cards as points in a feature space, as cards themself lack a metric to measure the strategic distance between poker hands. The features are the measurable attributes of the card combinations . They should measure the strength of a player's hand given the current state and as well as how likely it would be to win with this particular hand in the given situation. One possibility of grouping hands is by their **expected hand strength (**$\mathbb{E}[HS]$**)** [7], thus hands with a high probability of winning will be grouped with other strong hands and hands with a low probability of winning will be grouped with other weak hands. The **hand strength (**HS**)** measures the probability of winning with a given card combination against an opponent, where ties are counted as half a win. The opponent usually could hold a range of hands. Since the exact hand range of the opponent is unknown, it is usually assumed that the opponent plays all private card combinations with equal probability. In other words, a uniform distribution over the opponent's private cards is assumed. $\mathbb{E}[HS]$ is computed by averaging wins per hand against this uniform distribution over hole cards and adding the average number of ties divided by two to take split pots into account. For instance, the combination A♣K♣ wins in about 66.22% and ties in about 0.83% of all games against a uniform distribution of opponent card holdings. Thus, the corresponding $\mathbb{E}[HS]$ value is 0.6704.

However, $\mathbb{E}[HS]$ value ignores the potential of a hand to possibly develop into a stronger hand due to future chance events, which is an important property of a hand. In other words, drawing hands such as 7♣8♣ may have low $\mathbb{E}[HS]$ in the first betting round, but future community cards may turn the hand into a flush or a straight with high $\mathbb{E}[HS]$. Hence, a strong drawing hand should be considered to be comparably strong as hands with medium or even high $\mathbb{E}[HS]$ values. **Expected hand strength squared (**$\mathbb{E}[HS^2]$**)** takes the hand potential into account [23, p.25]. In order to calculate $\mathbb{E}[HS^2]$, the HS values of all river card combinations are computed and subsequently these HS values are squared, summed and averaged. Accordingly, this method assigns higher values to hands that exhibit a high variance in winning, as for instance a Flush or Straight Draws.

In order to capture the hand potential more precisely, histograms can be used to represent a **hand strength distribution** instead of summarizing it into a single expected value [21]. Figure 5 depicts the distribution of hand strengths over the final round of four Texas hold'em poker hands in the first round of the game. Each distribution is discretized into a histogram with values ranging from 0 (a guaranteed loss) to 1 (a guaranteed win). The height of each bar indicates the probability of receiving the corresponding hand strength in the future, and the vertical black line and label shows $\mathbb{E}[HS]$ [21, p.5].

All card features discussed so far are based on hand strength and hand strength squared values, which primarily are measures of the quality of a player's hand that consists of players private cards and the given community cards. The public information of board texture may only be indirectly inferred by these features, since public cards affect player's hand strength. However, knowledge about the board texture is crucial for an agent to distinguish between important situations. For example, a monotone connected board such as 7♣8♣9♣ requires entirely different consideration than a paired rainbow board such as K♣K♠2♡ which exhibits no draws. In order to add explicit knowledge of the board texture, either public card features should be added to the feature set, or public buckets are implemented where public cards

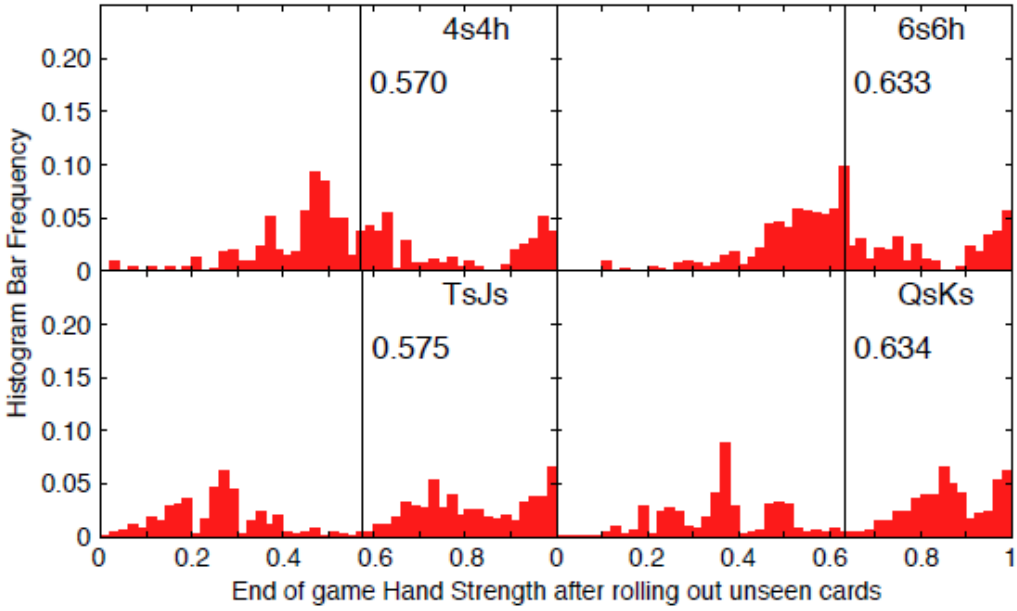

Figure 5: Hand Strength histograms for four poker hands on Preflop [21, p. 4]

are grouped separately. Public buckets can be used for a bucketing technique called **nested bucketing**, where only hands are partitioned that are consistent with given information [23, p.26]. For example, one could first partition only based on public information into M buckets, then partition the hands within each of M buckets into N buckets based on $\mathbb{E}[HS]$ values, for a total of $M \times N$ buckets.

In each betting round, a player's hand is assigned a different bucket due to additional community cards in each new betting round. The **bucket sequence** is the sequence of buckets to which the cards of a player were mapped in each respective round [30]. Assuming that the buckets are sorted by strength, if a player holds a weak hand in the first betting round improving in combination with the community cards of the second round, the bucket sequence could for instance be: bucket_1 preflop $\rightarrow$ bucket_9 flop. A player's strategy is not defined for a hand, but for a bucket sequence. In an abstracted game, hands with the same bucket sequence will be merged. Here we distinguish two types of card abstractions: **perfect recall abstractions** and **imperfect recall abstractions** [13, p.11-12]. In a perfect recall abstraction, no information about buckets in earlier betting rounds is ever lost. Nested bucketing is usually used to create perfect recall abstraction by conditioning on all previous revealed information, in other words the current bucket sequence. In imperfect recall abstractions, information about earlier betting rounds can be completely or partially forgotten. Perfect recall abstractions therefore consider the whole bucket sequence, while imperfect recall abstractions only consider a part of the bucket sequence. Hands with the following bucket sequences: $\text{bucket}_2 \rightarrow \text{bucket}_8$ and $\text{bucket}_5 \rightarrow \text{bucket}_8$ would be treated differently by a perfect recall abstraction, even though they have the same bucket on the current betting round. An imperfect recall abstraction would merge both hands thus forgetting the bucket of the first round.

Since perfect recall abstractions do not forget past information but use nested bucketing, the bucket sequence forms a tree. An abstraction with 10 buckets in the first round and a split in 10 additional buckets in each subsequent round would result in a total of 10000 bucket sequences. The number of buckets in earlier rounds therefore has a significant impact on the size of the whole abstraction.

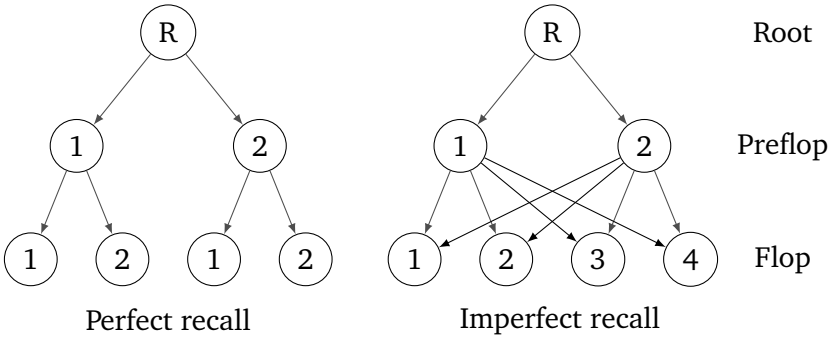

Figure 6: Perfect and imperfect recall games

In an imperfect recall abstraction, the number of buckets in earlier rounds does not affect the number of buckets in later rounds. Bucket sequences are not perceived as a tree, but rather as an acyclic graph. Cards, which were assigned to different buckets in previous rounds, can later be merged into the same bucket. Imperfect recall abstractions ignore past information in favor of a finer granulated abstraction in the current round. So instead of using nested bucketing with past betting sequence as split feature, one can discard all information about past rounds that are no longer strategically important and partition the current round separately only based on relevant features (e.g finer-grained $\mathbb{E}[HS]$).

In order to better illustrate the difference between perfect and imperfect recall abstractions, let us take Figure 6 into consideration. On preflop, the card abstraction partitions all hands into two buckets, 1 or 2, indicating that a hand is in the top or bottom half of all possible hands. At this point, both variants still have the same buckets. Continuing with the flop, in the perfect recall abstraction, the agent must remember its bucket sequence. A nested bucketing is used by conditioning the bucket number in the preflop round. Hence, each bucket on preflop is also partitioned further into either the top or bottom half, resulting in a total of four flop buckets. These four buckets may have overlapping ranges according to metrics such as $\mathbb{E}[HS]$. For example, a weak hand which is assigned to bucket_1 on preflop can improve to a strong hand, and a strong hand in bucket_1 on preflop can turn into a weak hand on the flop. Hence, $\text{bucket}_{1\to2}$ and $\text{bucket}_{2\to1}$ for instance could both have hands ranges from 0.5 to 1 $\mathbb{E}[HS]$-values. Imperfect recall, on the other hand, discards all prior bucket information and is able to partition all flop hands into four non-overlapping buckets resulting the same total flop bucket size as perfect recall. The partition of $\mathbb{E}[HS]$ values is twice as fine. Since $\mathbb{E}[HS]$ is more important than past bucket information, a finer partitioning according to $\mathbb{E}[HS]$ is better than a nested bucketing conditioned on all previous bucket information. Even though imperfect recall abstractions lose the guarantee of convergence to a Nash equilibrium, empirical tests demonstrate that imperfect recall abstractions usually lead to better results than perfect recall abstractions of equal size [37].

3.3.3 Action Abstraction

Another option of abstracting a game such as no-Limit Texas hold'em is to reduce the number of possible actions, as player actions contribute significantly to the size of the game [23, p.23]. Unlike typical card abstraction techniques, which group information sets into buckets, action abstraction usually removes player actions entirely. In case a removed action is actually played in the real game, it will be mapped to one of the remaining actions in the abstracted game using one of the various action translation techniques, which will be discussed in the following section.

Usually, folding and calling are not removed in an action abstraction. These actions are allowed in all situations of the abstracted game where they are also legal actions of the full game. The abstraction usually focuses on reducing the number of raise sizes. A widely used approach is to define certain allowed raise sizes relative to the current size of the pot, which are usually chosen by hand. A popular and simple action abstraction is the FCPA-abstraction (fold, call, pot size bet, all-in) [32].

An action abstraction can define more or fewer allowed raise sizes depending on the situation [17]. More actions might be allowed in a state, which occurs more frequently, as it will determine the outcome more strongly. Therefore, the agent will benefit from a finer abstraction in that state. For example, the initial raise in each betting round has the most allowed raise size variations. As the number of raises increases, the number of allowed sizes decreases in order to limit the number of bets a player can make.

A poorly chosen abstraction, or an abstraction which is not sufficiently accurate, can lead to many problems. An agent could misjudge the size of the pot as well as his opponent's hand distribution. For instance, a raise falsely mapped to a significantly larger size can lead to the agent folding reasonably strong hands, as he will think that he would have to invest a lot of chips for a call as his opponent has indicated a very strong hand. Several techniques have been proposed to improve action abstractions as well as the action translations. An action abstraction can, for instance, change the size of the agent's bet thus the pot size in the real game will be closer to a size defined in the abstracted game [32]. Action translations may be designed in a way, that they will minimize the agent's exploitability resulting from its abstracted action space [32].

3.3.4 Action Translation

An Action Translation is a reverse mapping from an action sequence performed in the real game to one of the action sequences defined in the abstracted game. We can distinguish between **hard translations** and **soft translations** [32]. A hard translation is a function, which deterministically maps an action sequence of the full game to a most similar action sequence of the abstracted game, while a soft translation is a probabilistic mapping. The biggest drawback of hard translations is their deterministic nature, which makes them predictable and exploitable. In many cases, soft translations also take the most similar action, but they additionally mix in certain probabilities for the second most similar action. For example, if an abstracted game only has the bet sizes, 1 or 2 units and the opponent actually made a bet of 1.55 units. A hard translation would map it to 2 units every time, since 1.55 is closer to 2 than to 1. However, a soft translation for instance could map it to 2 55% of the time and 45% of the time to 1.

There are various similarity metrics to measure the similarity between action sequences. In the following, a few popular similarity metrics will be listed.

Arithmetic

Let A and B be neighboring bet sizes defined in the abstracted game with $B > A$. Let x be the observed bet size in the real game, with a size between A and B. Then, if $x < \frac{A+B}{2}$, x is mapped to A, otherwise x is mapped to B. In other words, the bet size is always mapped to the nearest bet size defined in the abstracted game. This approach can be improved by adding randomness. In that case, the probability of the bet size x being mapped to the abstracted bet size A is $f_{A,B}(x) = \frac{B-x}{B-A}$ [11].

Geometric

Instead of looking at the distance of the bet x to the abstracted bet sizes A and B, the geometric mapping looks at the ratio of x, A and B. If $\frac{A}{x} > \frac{x}{B}$, the bet will be mapped to A, otherwise it will be mapped to

B. Therefore, the threshold between a bet being mapped to the bigger abstracted bet size B is $x^* = \sqrt{AB}$ instead of $\frac{A+B}{2}$ in the arithmetic mapping [11].

Pseudo Harmonic

All aforementioned mappings are heuristic mappings without game theoretical justification. The Pseudo Harmonic mapping attempts to generalize the game theoretical solution of a simplified poker game, the **Clairvoyance Game**, to the full game of no-limit hold'em [11].

In the clairvoyance game, player$_1$'s distribution is made of 50% winning hands and 50% losing hands, while player$_2$ always holds a hand, which will beat player$_1$'s losing hands, but will lose against his winning hands. Both players have a stack size of n and both have to pay a blind of 0.5 for an initial pot of 1. Player$_1$ is allowed to check or bet any amount x up to the size of his stack, player$_2$ is allowed to call or fold to player$_1$'s bet, he is not allowed to raise. Player$_2$ has to check if player$_1$ checked.

This game can be solved analytically, and the solution is as follows:

- P_1 bets n with probability 1 with a winning hand
- P_1 bets n with probability $\frac{n}{n+1}$ with a losing hand (and checks otherwise)
- P_2 calls a bet of size x with probability $\frac{1}{x+1}$

The Pseudo Harmonic mapping is the solution to $f_{A,B}(x) \cdot \frac{1}{1+A} + (1 - f_{A,B}(x)) \cdot \frac{1}{1+B} = \frac{1}{1+x}$, which results in $f_{A,B}(x) = \frac{(B-x)(1+A)}{(B-A)(1+x)}$. This is the only mapping consistent with the solution of the Clairvoyance Game of player$_2$ calling a bet of size x with probability $\frac{1}{1+x}$. It also has a lower exploitability than the previous mappings, in the toy games such as Kuhn Poker [20] and Leduc Poker, which suggests that the solution of the Clairvoyance Game can improve an agent in more complex poker games.

3.4 Opponent Exploitation

Nash equilibrium solutions are static and robust strategies that provide a worst-case performance guarantee. They make the pessimistic assumption that the opponent always plays perfectly against them. In a two-player zero-sum game, where Nash equilibrium solutions have zero exploitability, an agent playing according to a true Nash equilibrium strategy will never lose to any opponent in the long run. However, human players or even the best computer poker agents have significant weaknesses that can be exploited to obtain higher payoffs than the optimal Nash equilibrium strategy would have, since Nash equilibrium strategies do not take advantage of any kind of weaknesses. In order to capture these weaknesses, **opponent modeling** is required. A perfect opponent model describes the exact behavior of the opponent. Opponent models can be derived from prior observation of the opponent. Based on the opponent model, a counter strategy can be computed that maximizes its payoff against the given opponent model. The strategy with the highest payoff against a given opponent model is called **best response** (3.1).

However, it is almost impossible to construct a 100% accurate opponent model due to estimation errors and noisy observations. The opponent further might change his behavior dynamically, which is also common among advanced human players. Thus, playing according to a tailored counter strategy tends to be risky, since it deviates from the Nash equilibrium. This deviation exposes the agent to potential counter-exploitation. For instance, consider the game of Rock-Paper-Scissors, where the opponent plays a rock only strategy. In this case, the best response strategy is obviously to always play paper. However, the only paper strategy is as brittle as the opponent's only rock strategy and can be easily counter exploited by

any scissor dominating strategy. Therefore, besides the challenge of opponent modeling, another crucial part of opponent exploitation is to compute robust response strategies that limit their own exploitability while still being able to exploit the opponent. Such responses balance the two extremes, best response and nash equilibrium, and are referred to as *safe exploits* [5, p.13-16].

In this section, we first describe several response computation techniques that are widely used in state of the art poker agents. They are primarily game theoretic counter strategies that are computed offline. Subsequently, we will discuss the differences between two general approaches of opponent exploitation systems, **explicit modeling** and **implicit modeling**. The latter one also has the ability to perform online exploitation and dynamically adapt to various opponent types.

3.4.1 Game Theoretic Responses

Best Response

Despite the vulnerability against potential counter-exploits best response remains the most efficient exploitation technique with regard to weak static opposition. Additionally, best responses are also used in agent evaluations to determine the worst-case exploitability of agents. Computing a best response to a strategy in large-scale extensive games is challenging. It was even impossible to compute best response for limit Texas hold'em for a long period of time. Instead, it was necessary to compute the best response to a strategy σ_{opp} inside of the same abstracted game that σ_{opp} uses. This abstracted best response exploits σ_{opp} to a maximum extent in the abstracted game and provides a lower bound of the real exploitability of σ_{opp}.

However, Johanson et al.(2011)[39] proposed an implementation of an accelerated best response calculation that employs a public state tree for traversal instead of the conventional extensive game tree, which in turn allows more opportunities for caching and reusing information. Moreover, they reduced the size of the game tree by using game-specific isomorphisms and employed parallel computation to solve subtrees independently. These improvements allow at least efficient best response computation for limit Texas hold'em. Unfortunately, best response computation for the no-limit variant of Texas hold'em is still not feasible due to the vast action space. Therefore, one must revert to the abstracted best response for the no-limit variant.

The core principle behind best response is simple. The algorithm traverses over all information sets of the best response player $br(opp)$. For each game state in the information set, the probability of σ_{opp} reaching every terminal node descending from that game state, and the associated utility of both players reaching that terminal node are computed. Then, the expected utility for each action is determined, and the single action for the $\sigma_{br(opp)}$ that maximizes this utility is selected [23, p.55]. The resulting $\sigma_{br(opp)}$ is a pure strategy that always selects the action with the highest expected payoff at each information set.

Frequentist Best Response

Frequentist best response (FBR) [23] is an exploitation technique that involves two steps. First, it builds an offline opponent model (called frequentist model) for a particular opponent by using fully observed game histories of the opponent (including hole cards) and then computes an abstracted game best response to this model.

In preparation for the model creation, a default policy and an abstraction are defined. The default policy is employed in situations where no observations of opponent have been made. The opponent model is constructed by mapping observations of played hands to frequency counts in the chosen abstraction. The quantity of observations is essential for the accuracy of the model. Johanson(2008)[23]

reported that one million observed games are required to generate an appropriate opponent model in a five bucket abstraction. In practice, it is impossible to collect observations of this size from a human player, notwithstanding the missing information on hole cards in non-showdown games. Another drawback of this technique is its poor performance against other opponents that the model is not designed for, which appears obvious, as standard best response strategies are always brittle. Nevertheless, frequentist best response showed good results against static computer agents.

Restricted Nash Response

The Restricted Nash Response (RNR) [23] is a technique that compromises the advantages and drawbacks of best response and Nash equilibrium. While best response strategies are tailored to maximally exploit a specific type of opponent, they can perform very poorly against other types of opponents. Unless the computed opponent model is very likely to be accurate and static, best response should rather not be employed. Unlike best response, RNR has the ability to exploit a given opponent while remaining as robust as possible to counter-exploitations.

The RNR algorithm balances best response and Nash equilibrium computations by restricting the opponent's strategy. In Nash equilibrium computations, the opponent is free to change his strategy constantly in order to approach the equilibrium. In best response computations, the opponent plays a fixed strategy and only the best response player adapts his strategy to exploit the fixed strategy. RNR combines both by forcing the opponent to play the fixed strategy with probability p, and leaving him to play an unrestricted strategy with probability $1 - p$. In other words, any equilibrium finding algorithm can be used with the modification that the opponent plays a given fixed strategy p percent of the time. The other unrestricted player is always free to choose his actions. As a result, the strategy of the unrestricted player is a counter strategy that exploits the known fixed strategy, while still preserving the robustness of a Nash equilibrium strategy.

In order to determine the opponent's fixed strategy, frequentist model technique can be utilized. The degree of trade-off between best response and Nash equilibrium is controlled by the choice of p. p can vary between 0 and 1, increasing it results in increase of exploitative power and brittleness. If $p = 0$, the produced strategies would be Nash equilibrium strategies, whereas when $p = 1$ the result is a best response to the fixed strategy.

Although RNR does not fully exploit its opponents like FBR does, the exploitability of RNR is vastly reduced compared to FBR [23]. However, the sacrifice of exploitative power is not nearly as high as the decrease of exploitability, which reflects the overall effectiveness of RNR. Although RNR provides substantially more robust responses, empirical results demonstrated [24] that RNR can perform poorly when the fixed strategy is estimated from observations of the opponent rather than the opponent's real strategy. Data biased response (in the next paragraph) addresses this problem and is more suited for estimated agent models.

Data Biased Response

As mentioned in the previous section, data biased response (DBR) [24] is a similar approach to RNR with the difference that DBR is capable of dealing with estimated opponent models. DBR also uses the p value to restrict the opponent. While RNR uses a constant value of p for all information sets, DBR assigns individual p_{conf} values to each information set that defines the probability the opponent has to play according to the fixed strategy at the given information set.

The problem of using a global p value for all information sets (as RNR does) is that the opponent model is not equally accurate at all information sets. Depending on how frequently an information set is observed, the opponent model's accuracy can vary immensely. In an extreme case, where no observations have been made in a particular information set, the opponent is still forced to use the default policy p percentage of the time, which is clearly not optimal. Reversely, the algorithm should put more faith in the opponent model at information sets that have already been observed numerous times. DBR uses p_{conf} as a confidence measure that reflects ones belief in the opponent model. p_{conf} is a function of the number of times the corresponding information set was observed. The more observations, the higher is p_{conf}. Information sets with zero observations also receive zero p_{conf}, allowing the opponent to play an unrestricted strategy. Thus, a default policy is no longer necessary. By tuning p_{conf}, DBR adjusts the trade-off between exploitation and exploitability. Johanson et al.(2009)[24] deployed an additional p_{max} value as the upper bound of exploitation which p_{conf} cannot exceed. Thus, p_{max} sets the final trade-off between best response and Nash equilibrium.

There is another benefit of DBR as both strategies resulting from the solution of the modified game are useful. The strategy of the unrestricted player is the robust response strategy. The strategy of the restricted player is a mimic of the opponent that incorporates the opponent's behavior derived from the observations while being robust to the other player's strategy. These mimic strategies can be quite useful. For example, Bard et al.(2016)[5] used mimic strategies for various agent evaluations.

3.4.2 Implicit Modeling

Each of the aforementioned techniques attempts to model a single opponent explicitly. In real life domains, it is often impossible to model every single parameter of opponent's strategy correctly as this would usually require a large number of opponent observations. Even if there is already a given model, computing an appropriate response is still too time consuming to be accomplished online. Furthermore, there is a wide range of different opponent types that cannot be covered by a single opponent model.

Bard et al.(2014)[4] proposed an implicit modeling approach, where the agent may choose from a **portfolio** of precomputed responses during game-play. All the agent has to do is to estimate the utility of each response in the portfolio and selects the one with the highest estimated utility against the current opposition. This approach allows agents to adapt efficiently to a broad range of different opponents in real time. The quality and selection of the portfolio strategies are crucial. Each strategy in the portfolio is computed offline. Hence, there is enough time to build more sophisticated responses for the portfolio. The portfolio should be able to exploit a variety of opponents. Typically, responses are computed for the two to three most common player types by using one of the robust response computation techniques such as RNR or DBR. Robust responses are capable of exploiting without being completely vulnerable to counter-exploits at the same time. With the right portfolio computed offline, the online adaption task becomes a utility estimation problem. The selection of responses is accomplished by using multi-armed bandit algorithms augmented by variance reduction techniques. A complete illustration of an implicit modeling process is provided in Figure 7.

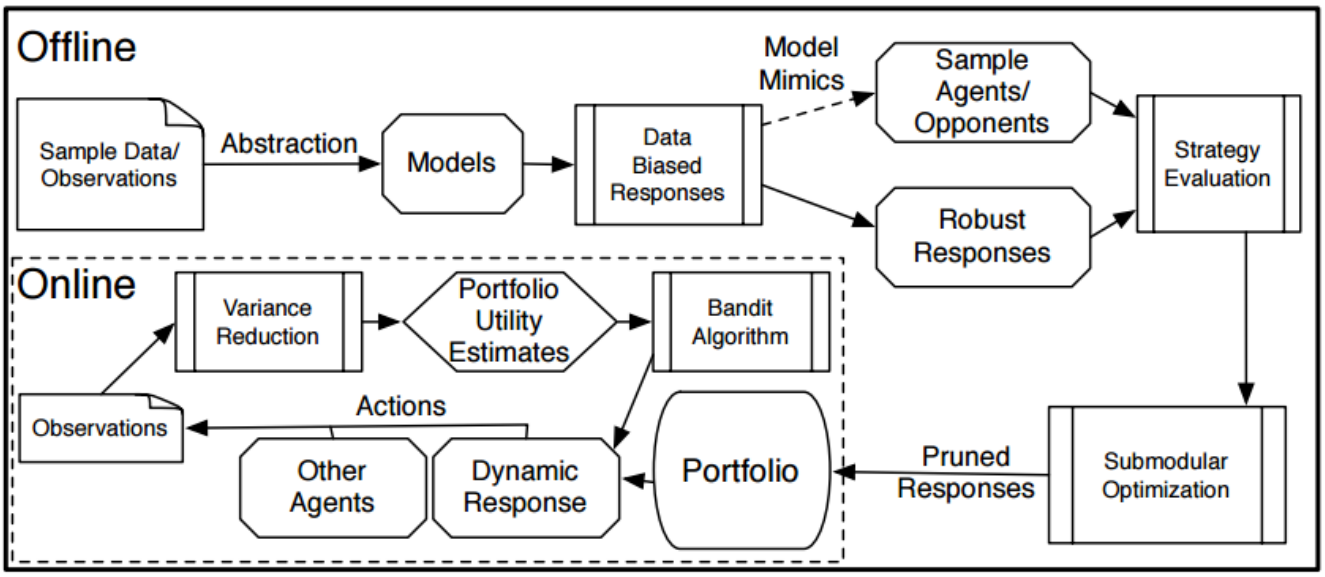

Figure 7: An example of implicit modeling process [4, p.5]

4 Enhancements of existing Nash Agents I: Endgame Solving

Endgame solving is not a new concept. It has already been successfully implemented in perfect information games such as chess or checkers [6]. In large imperfect information games, the common approach for creating Nash equilibrium agents is to compute an ϵ-Nash equilibrium strategy of an abstracted version of the full game offline. Due to the immense size of most popular games such as Texas hold'em poker even reasonable abstractions of the original game are too large to be solved exactly by the state of the art equilibrium-finding algorithms such as linear programming. Therefore instead of an exact computation approximate equilibrium-finding algorithms such as CFR are usually used for the offline computation. During actual game-play, the agent can simply consult the offline computed strategy and performs its actions accordingly.

In two-player no-limit Texas hold'em, the total number of game states with 200 big blinds starting stacks is about 10^{165} [22]. However, this size clearly does not scale to the capacity of modern hardware, which is why the number of game states must be abstracted down to only 10^{12} at least. The dramatic reduction of game states consequently has a negative impact on the quality of the solution. Game states with different strategic properties may be abstracted down to one state, even though they should be processed differently. Assuming that the same abstraction technique is being used, the solution of a bigger abstracted game with more game states often outperforms the solution of a smaller one. However, solving larger abstracted games comes at the expense of computation time and memory space. So it is a trade-off between accuracy of the abstraction and time/memory efficiency. Endgame solving partially addresses this trade-off problem by resolving the endgame to a greater degree of accuracy in real time, since it is only required to solve a small portion of the game that is actually reached during game-play. Additionally, there is sufficient time for small size computation in real-time application scenarios such as online poker games where the time limit for one action is 10 to 20 seconds.

The endgame solving approach uses an existing Nash equilibrium strategy for the initial portion of the game called the **trunk**, but then discards the strategy for the final portion called the **endgame**, and solves it separately using finer granulated abstractions [9, p.1]. In the example of a poker game, the first

three betting rounds proceed as usual by performing table lookups in the precomputed Nash equilibrium strategy. When the last public board card is revealed, the endgame solving starts and solves the last betting round in real-time. The new computed endgame strategy can then be used until the end of the round. An illustration of endgame solving is provided in Figure 8.

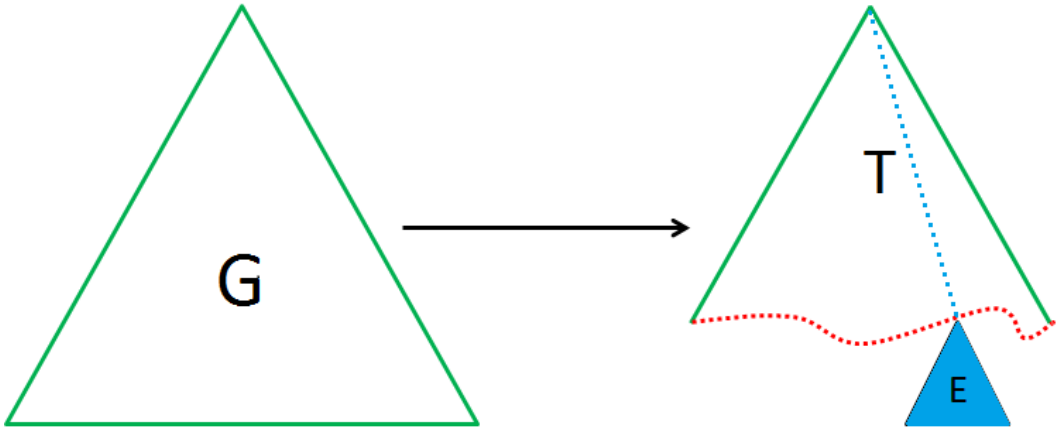

Figure 8: G is the entire game tree of the offline strategy. T is the trunk. Blue dashed line is a path that has been taken during actual game-play. Red dashed line represents the end of the trunk, where the endgames start. E is the endgame to be solved [10, figure 2].

Ganzfried et al.(2015)[9] first introduced endgame solving in large imperfect-information games. Their experiments on no-limit Texas hold'em showed that endgame solving can improve the performance of Nash equilibrium agents against other computer poker agents. In the first human vs computer no-limit Texas hold'em competition, endgame solving was also implemented in the computer agent which competed against four world class human poker players. Although the computer agent lost to humans in the competition, the professional poker players regard the endgame component as the strongest part of the AI [10, p.4]. In this chapter, we will first review the endgame solving algorithm proposed in [9] and highlight the strengths and weaknesses of endgame solving. However, one key component of the implementation was left out by the authors of [9], which is essential for a reproduction of the endgame solver. The missing part is an explanation for the formulation of the linear program, which is used to describe the entire extensive game that needs to be solved. Our main contribution in this chapter aims to provide a comprehensive instruction for the implementation of an endgame solver in the domain of no-limit Texas hold'em. Besides reviewing the techniques used in the original work, we will present certain alternatives, and as well as a sample implementation for the LP generation. For reasons of simplicity, we will refer to the authors of the orginal work [9] on endgame solving as the authors for the following chapter.

4.1 Theoretical Background of Endgame Solving

In order to understand the endgame solving approach, it is important to first define what an endgame is. The authors define an endgame as follows [9, definition.1]:

E is an endgame of game G if the following properties hold:

- The set of E's nodes is a subset of the set of G's nodes.
- If s' is a child of s in G and s is a node in E, then s' is also a node in E.

- If s is in the same information set as s' in G and s is a node in E, then s' is also a node in E.

In other words, G can be viewed as a rooted tree and E as a subtree of G whose root is a node of G and all descendants of that root are also included in E. Hence, E is the final strategic portion of G. The size of E can range from as large as G itself to as small as a single terminal node. The size of E determines the quality and the required amount of computational resources of the resulting endgame strategy. Hence, choosing an adequate endgame size is essential. Obviously, it makes no sense to choose E as large as G itself, otherwise it would the same as to solve the entire game at once which would represent the same approach as any regular Nash equilibrium computation. Choosing a too small size for E can also be problematic. The produced equilibrium strategy of E may fail to be a real Nash equilibrium strategy in the full game when combined with the equilibrium strategy of the trunk. This can be illustrated in an example of Kuhn poker.

Consider the last decision of the player$_1$ in Kuhn poker (Figure 9), whether to call or to fold to a bet of player$_2$ while holding a Queen, as an endgame E. The correct Nash equilibrium solution here would be to call $1/3$ and fold $2/3$ of the time [20]. The private hand distribution of the equilibrium strategy of player$_2$ is: $1/4$ Jack, $3/4$ King and zero Queen. However, a pure strategy of always call or always fold has the same expected outcome of -1 as the real equilibrium strategy, which means that the endgame solver might just as well output a pure strategy, which would not represent an equilibrium strategy in the full game.

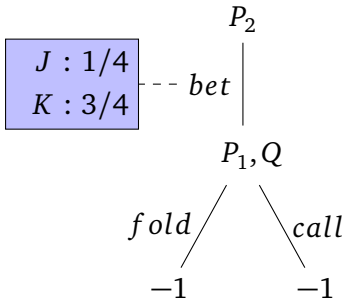

Figure 9: Endgame example in Kuhn poker: player$_1$ (holding a Queen) is facing a bet of player$_2$. Player$_1$ is indifferent of folding or calling, since both options have the same expected utility.

However, the authors have proven that endgame solving produces strategies with low exploitability in games, where the endgame is a significant strategic portion of the full game. In other words, if the opponent can exploit our strategy by adapting his strategy just within the endgame, the endgame solving will lead to reasonable results. This property can be summarized in the following proposition.

[9, Proposition 2]. *If every strategy that has exploitability strictly more than ϵ in the full game has exploitability of strictly more than δ within the endgame, then the strategy output by a solver that computes a δ-equilibrium in the endgame induced by a trunk strategy t would constitute an delta-equilibrium of the full game when paired with t.*

Proof. Suppose a strategy is a δ-equilibrium in the endgame induced by t, but not an ϵ-equilibrium in the full game when paired with t. Then by assumption, it has exploitability of strictly more than δ within the endgame, which leads to a contradiction. □

However, it remains difficult to measure the importance of the trunk for the endgames. In general, the bigger the endgames, the lower the end exploitability when paired with the trunk. In the domain

of Texas hold'em poker, the betting rounds form an intrinsic division of the full game. The final betting round (river) has the most money in the pot, thus each river decision has a stronger impact on the outcome than in other previous rounds. At the same time, with no more cards to come, the river is small enough to be solved efficiently. Hence, the final betting round appears to be an obvious choice for the endgame.

Despite the theoretical limitation of possibly producing highly exploitable strategies, the authors have reported several potential benefits of endgame solving in large-scale imperfect information games. The main benefit of endgame solving is the reduction of the problem complexity by decomposition. The endgame solving approach breaks the full game down into two separate entities, namely the trunk portion and the endgame. Since each endgame is solved separately from the main part of the game tree, the memory and scalability constraints are much lower. While the full game may be too large to fit in the memory, each endgame can easily fit into memory, and thus can be solved individually. Equilibrium finding algorithms will also more likely scale to endgames. The authors proposed to use LP algorithms for endgames that produce exact equilibrium solutions, unlike conventional approximate equilibrium-finding algorithms that are often confronted with large approximation errors. The current best general-purpose LP algorithms scale to games with up to 10^8 game states [14]. The scale of a reasonable abstraction of full game poker usually exceeds the computational limit of LP algorithms. However, LP algorithms do scale to abstracted endgames, since 10^8 nodes are enough for a reasonable endgame abstraction, for instance the river in Texas hold'em.

In no-limit Texas hold'em, the final betting round contains the most information sets of all betting rounds. Therefore, more information sets must be compressed into buckets than in previous betting rounds. By solving the endgame separately, much finer granulated card abstractions can be computed in real time. Additionally, the card abstractions can be computed in consideration of a more realistic private hand distribution of the opponents. Instead of grouping information sets together that perform similarly against a uniform distribution of the opponent's private information like standard approaches, hands which perform similarly against the relevant distribution of hands the opponent actually might have can be grouped together. For example, if the opponent has called large bets in previous rounds, it is considered unlikely that he is holding a weak hand, thus the hand distribution is shifted towards strong hands. Advanced human players also consider the opponent's hand distribution given the action history while making decisions. Action abstraction can be fine tuned as well. Endgame solving allows the user to adjust the granularity of the action abstraction dynamically for each endgame in real time.

Action abstraction reduces the number of actions by leaving a small number of discrete actions to represent the entire action space. Usually, the discretizing process is performed by simply removing actions from the action space. During actual game-play, the opponent may take an action that is not allowed in the abstraction. In order to interpret this action, it needs to be translated by mapping it to a known action of the action abstraction. It will be mapped to either the next bigger size or the next smaller size in the action abstraction. Usually, soft translation (see 3.3.4) is used. The difference between the interpreted bet size and the actual bet size could be very large, thus resulting in unwanted agent behaviors. For example, suppose the pot value is 100 and the remaining stacks have a value of 200. Besides call and fold, the action abstraction only allows bets of size 10 and 100. Suppose the opponent bets 70 in an actual game, which happened to be mapped to 10. Now, the agent would assume that the opponent has made a bet of 10. If the agent calls the bet, it will assume that the pot size is 120 and the remaining stack is 190. In reality, the pot size is 240 and the remaining stack size is 130, which is a completely different situation with a significantly bigger pot to stack ratio. By overestimating the

opponent's bet size, one could falsely assume a stronger hand distribution of the opponent, thus folding more hands which would have been called otherwise, if the correct bet size would have been assumed. This is what the authors refer to as the off-tree problem. Endgame solving corrects the pot/stack size gaps by constructing an abstracted endgame using the actual pot/ stack sizes. Therefore, the off-tree problem is solved at the beginning of the endgame. However, the off-tree problem still can be an issue within an endgame, since the action space is still abstracted.

4.2 Endgame Solver Implementation

In this section, we provide a deep insight into the endgame solver implementation. Firstly, we will review and explain the implementation provided by the authors. It includes the computation of the distribution of the players' private information leading into the endgames from the precomputed trunk strategies, the computation of the strategy-biased card abstraction and the action abstraction computation. Additionally, we will also add our own implementation details to complete a more comprehensive instruction. We also provide alternatives for the card and action abstraction computation. Finally, we describe the required steps to formulate the corresponding LP to the endgame.

4.2.1 Joint Hand Distribution Computation

One of the major benefits of endgame solving is being able to compute strategy-biased and history-aware card abstractions. More concretely, this means that instead of assuming a uniform random distribution for the opponent's hand distribution, the endgame solver uses a more realistic opponent hand distribution deriving from the precomputed Nash equilibrium strategy given the current betting history and the board cards. In other words, the basic assumption is that the opponent acts according to the base Nash equilibrium agent, which is comparably more reasonable than assuming a uniform distribution. Hence, the necessary first step is to compute the joint private hand distribution which reflects the probabilities of each private hand combinations for both players entering the given endgame. In case of the last betting round in Texas hold'em (river), there are $\binom{52-5}{2} = 1081$ possible hands. The joint hand distribution can be represented as a 1081×1081 matrix $\mathbf{D}$, where the rows represent the possible private hands of player_1 and the columns the private hands of player_2. Therefore, an index function ((Algorithm 2)) is required to assign a unique number between 0 and 1080 to each hand. An entry in $\mathbf{D}$ is the probability of both players holding respective private hands according to the base Nash equilibrium strategy in the given endgame.

In order to reduce the computation time of $\mathbf{D}$, which usually requires 1081^2 lookups in the base strategy table for all possible private hand combinations, the authors proposed an idea to make a simplifying assumption that the hand distributions of both players are independent. This implies that it would only be necessary to iterate once through the 1081 hands and look up the probabilities for each player separately in the base strategy table, therefore resulting in two probability vectors $\mathbf{d}_1$ and $\mathbf{d}_2$ for each player respectively. According to the independence property, the joint probability of two independent variables can be calculated by simply multiplying their probabilities. In our case, the probabilities from $\mathbf{d}_1$ and $\mathbf{d}_2$ are multiplied to achieve the corresponding joint probability in $\mathbf{D}$. Note that the hands of each player cannot share a common card, otherwise the corresponding joint probability is set to zero. In order to avoid additional computation in real-time, the authors computed a table of the shared common cards between each pair of private hands prior to any real-time computation. Since this table is designed for all endgames, it has the size of 1326×1326 dimensions including all possible two cards combinations

($\binom{52}{2} = 1326$). A corresponding index function for two cards combinations is given in Algorithm 3. This card conflict table can be implemented as a boolean matrix, in which entries are set to true if the corresponding hand indices share a card in common, otherwise the entry is set to false. After each entry of $\mathbf{D}$ is set, it is normalized, thus all entries sum up to one. The pseudo-code for the entire joint hand distribution computation can be found in Algorithm 4.

4.2.2 Card Abstraction Computation

In the previous section, we have seen the computation of the joint distribution of private hands induced by the base Nash equilibrium strategy. Now, by using the joint hand distribution, the equity arrays $\mathbf{e}_1$ and $\mathbf{e}_2$ that contain the equities for each possible private hand of player$_1$ against player$_2$'s private hand distribution, and vice versa. These equity arrays will be used to group the private hands into buckets according to their equities against the opponent's private hand distribution instead of a uniform random hand distribution.

Both equity arrays have the number of possible private hands (1081 on the river) many entries and are initialized with zeros. Then, for each pair of private hands h_1 and h_2, the entry $\mathbf{D}[h_1][h_2]$ is added to $\mathbf{e_1}[h_1]$ if the rank of h_1 is higher than h_2, and $\frac{\mathbf{D}[h_1][h_2]}{2}$ is added if both hands have the same rank. Analogously, $\mathbf{D}[h_2][h_1]$ or $\frac{\mathbf{D}[h_2][h_1]}{2}$ is added to $\mathbf{e_2}[h_1]$ on the same hand rank conditions. The rank of a hand indicates the strength of a hand. As the board is complete on the last betting round, the ranks of the hands are also fixed. Finally, each entry is normalized so that all equities are between zero and one. Some of the entries, of which the indices correspond to private hands with zero probability of being played according to the base Nash equilibrium strategy, are irrelevant and are thus set to -1. Pseudo-code of this part is given in Algorithm 5.

Now, card abstractions can be computed by grouping $\mathbf{e}_1$ and $\mathbf{e}_2$ into card buckets. The number of card buckets should be chosen in a way that ensures a reasonable computation time of the endgame solving. The size of the abstracted endgame should not exceed a certain threshold in order to guarantee fast computation. The abstracted game size can be measured by the number of information sets, which in turn can be estimated by the product of the number of card buckets and the number of action sequences. The number of action sequences depends on the action abstraction, which will be addressed in the following section. The number of card buckets for player$_i$ k_i can be calculated by $k_i = \lfloor \frac{T}{b_i} \rfloor$, where T denotes the number of information sets of player$_i$ and b_i denotes the number of player$_i$'s action sequences.

The card bucketing can be realized by using various clustering algorithms. These have the same objective, namely to assign hands that are strategically similar to each other into the same bucket. The feature space is one-dimensional with the equity against the opponent's hand distribution as the only feature. In the following, we will look at a couple of clustering techniques that we have implemented.

k-Means Algorithm

k-Means algorithm [18] is a simple iterative clustering algorithm that partitions data into k clusters by assigning each single data point to its closest cluster center. The initial cluster centers can be computed by the k-means++ Initialization [2] that spreads out the initial cluster centers to achieve better cluster solutions in the main algorithm. $\mathbf{e}_1$ and $\mathbf{e}_2$ are clustered separately for each player. Each entry in $\mathbf{e}$ represents a data point in the data space. The distance between two data points is measured by the difference of their corresponding equity values. k-Means is only capable of finding local optima given the initial clusters means. In order to improve the quality of the clustering, the algorithm can be repeated

multiple times with different initial cluster means and select the clustering with the lowest average distance to the mean.

However, the authors pointed out that k-means could produce sub-optimal clustering in some settings. Suppose there are many hands with an equity of 0.855, and also many hands with an equity of 0.859. In this case, k-means would likely create separate clusters for these two equity values, and possibly assign very different equities, for instance 0.1 and 0.2, into the same cluster if there are only few hands with those equities. Nonetheless, k-means clustering still produces reasonable results in general.

Percentile Hand Strength

The authors introduced a clustering technique called percentile hand strength, which addresses the concerns of potentially putting different equities into the same bucket. This technique clustering groups hands in the same hand strength range together by using simple hashing method [23, p.26]. Concretely, the equity interval $[0, 1]$ is divided into k regions of equal length, where k denotes the number of card buckets. The bucket index of a private hand h can be calculated by calculating the hash function $\lfloor \mathbf{e}[h] \rfloor \cdot k$. To ensure that the absolute top strength hands are grouped together, the first bucket is reserved just for hands with equities higher than a predefined threshold a. Then, the remaining equity interval $[0, a]$ is divided according to the hash function $\lfloor \frac{\mathbf{e}[h] \cdot (k-1)}{a} \rfloor$. This approach often produces significantly fewer buckets than k, since there may be zero hands that fall into certain regions. In this case, the number of card buckets is reduced accordingly. In the end, the resulting card abstraction may contain a very different number of buckets for each player, which is not very surprising, since both players could be in possession of completely different hand distributions. Pseudo-code of this clustering algorithm is provided in Algorithm 6.

Although this clustering technique does not have the problem of grouping completely different equities together as k-means does, it may create too few clusters if the majority of hands have similar yet different equities. Consider the following scenario: suppose we want to create five clusters, and all hands have equities between 0.6 and 0.8. The five evenly divided equity regions are $[0, 0.2]$, $[0.2, 0.4]$, $[0.4, 0.6]$, $[0.6, 0.8]$ and $[0.8, 1]$. Therefore, all hands are assigned to cluster $[0.6, 0.8]$ which is clearly not optimal, as we have enough capacity to create five buckets.

Percentile Hand Strength with Equally Sized Buckets

The percentile hand strength method can be modified to create buckets containing equal number of hands. Instead of using the hash function to assign hands into buckets, all hands are sorted in ascending order according to their respective equities and the bottom $\frac{100}{k}$ percent of hands are assigned to the lowest bucket. The next $\frac{100}{k}$ percent of hands are assigned to the next higher bucket, and so on. The pseudo-code is given in Algorithm 7. This method ensures that all buckets are equally sized and the available abstraction capacity is fully used. The drawback of this approach is that it may create unnecessarily many buckets when for instance all hands have the same equity.

4.2.3 Action Abstraction Computation

The action abstraction determines which actions are available at each game state, and thus also determines the number of possible action sequences for each player. We say an action abstraction is finer granulated than another action abstraction when it enables more action sequences. As mentioned previously, the product of the number of action sequences and the number of card buckets determines the size of the abstracted game. In this equation, there is a clear trade-off between these two kinds of ab-

stractions. Without affecting the total abstracted game size, creating a finer granulated card abstraction clearly implies that the size of the action abstraction has to be reduced, and vice versa. The optimal balance is application specific. However, there are some rules of thumb which abstraction to favor in certain situations. For example in endgames with larger pot sizes, the action abstraction should have more bet sizes available because the game tree cannot grow too large due to the shallower stacks (once the players are all-in, no additional bets are allowed); and larger pots are more important, since there is more money at stake and an error caused by an off-tree problem would be more expensive.

By using endgame solving, the granularity of the action abstraction could be adjusted dynamically. The authors propose to create a set of different action abstractions of varying granularity in advance of game-play. During game-play, an abstracted game is created for each action abstraction in descending order of granularity, and thereby, the number of action sequences in each action abstraction can be determined. If the number of action sequences multiplied with the number of card buckets is below the predefined maximum abstracted game size, then this action abstraction is utilized. Since the action abstractions are processed in descending order of granularity, the finest abstraction possible is always chosen. However, this approach can be time consuming in case of small pots, because the endgames are bigger due to bigger remaining stacks (more action sequences possible). In the worst-case, the number of action sequences has to be computed for all available action abstractions just to find out that only the last (most coarse) one is suitable. In our implementation of this approach, this process could take up to two seconds for three to four different action abstractions, which is too slow for real-time applications such as online poker if we take into consideration that this is only a small computation part of the entire endgame solving.

One alternative approach would be to define fixed pot size ranges for each granularity of action abstractions. Coarser action abstractions are assigned to handle small pot sizes while finer action abstractions are used for big pot sizes. It should be ensured that each action abstraction still produces feasible endgame sizes in its application range, which means that all endgame sizes would still remain below the predefined maximum size. The advantage of this approach is that merely the computation of a single action abstraction is needed

We can also define one single fixed action abstraction and adjust the size of the card abstraction dynamically. Similar to adjusting the granularity of the action abstraction, we can use more card buckets in bigger pots and fewer in smaller pots. The obvious drawback is that we are forced to use a rather coarse action abstraction even in big pots, which may lead to more translation errors.

The design of the predefined action abstractions is essential for the quality of the endgame strategy. A good action abstraction should reduce the frequency and severity of action translation errors and at the same time cause translation errors in the opponent's strategy. We can distinguish between two major approaches of action abstraction design, symmetric and asymmetric action abstractions. Symmetric action abstractions assign identical action abstractions to all players. Asymmetric action abstractions assign different action abstractions to the agent than to his opponent. For example, we can assign more bet sizes to the opponent, in order to interpret more types of bets so that the impact of translation errors is minimized; since more bet sizes are given to the opponent, we have to only assign fewer bet sizes to the agent if we want to maintain the total abstraction size. In chapter 6, we will discuss these two approaches in more depth.

4.2.4 Linear Program Generation

The final step of endgame solving is to generate a linear program for the given endgame using the sequence-form formulation (see 3.2.1). The LP generation process can be imagined as the construction of the game tree. The action abstraction tells us how to connect the nodes. The card abstraction is responsible for the payoffs in the terminal nodes and the chance events. To better understand the procedure, we will now use Kuhn poker as an example throughout this section. The complete game tree of Kuhn poker is depicted in figure 13.

One last preparation has to be made initially before the LP generation. After the card buckets are computed, we will only use buckets instead of single hands, as hands in the same card bucket are indistinguishable. Henceforth, card buckets replace the use of individual hands. In order to construct the game tree later with card buckets, the joint bucket distribution **C** and the bucket equity table **T** need to be computed first, which are used for the chance node of the game tree and for the utility value of the terminal nodes. In order to compute the joint bucket distribution, we iterate over all pairs of private hand combinations h_1, h_2. In each iteration, we look up the buckets c_1, c_2 that the private hands are assigned to, and add joint probability $\mathbf{D}[h_1][h_2]$ to the joint bucket probability $\mathbf{C}[c_1][c_2]$ (Algorithm 8). The bucket equity table is computed in a similar manner, where we count the number of higher ranked hands in each bucket against other buckets, and normalize at the end of the loop (Algorithm 9). In the Kuhn poker example, the joint bucket distribution **C** provide the probabilities of each chance event ($Q|J, Q|K$ etc), and the bucket equity table **T** is used to determine the winner at the showdowns.

$$\mathbf{C} = \begin{array}{c} \\ J \\ Q \\ K \end{array} \begin{array}{c} \begin{array}{ccc} J & Q & K \end{array} \\ \begin{pmatrix} & 1/6 & 1/6 \\ 1/6 & & 1/6 \\ 1/6 & 1/6 & \end{pmatrix} \end{array} \qquad \mathbf{T} = \begin{array}{c} \\ J \\ Q \\ K \end{array} \begin{array}{c} \begin{array}{ccc} J & Q & K \end{array} \\ \begin{pmatrix} & & \\ 1 & & \\ 1 & 1 & \end{pmatrix} \end{array}$$

Figure 10: Joint bucket distribution and bucket equity table of Kuhn poker. Zeros are omitted

Recall the two LPs (7) and (8) of which the solutions are the strategies of the Nash equilibrium strategy profile. The task is now to construct the constraint matrices (**E** and **F**) and the payoff matrix (**A**). The constraint equations $\mathbf{Ex} = \mathbf{e}$ and $\mathbf{Fy} = \mathbf{f}$ cover the latter two constraints for valid realization plans ("chidren sum to parent" and "root has weight 1") for the player$_1$ and player$_2$, respectively. Figure 11 depicts the constraint matrices for Kuhn poker.

For example, consider now the constraint matrix **E**, which represents the constraints on player$_1$'s realization plan. It has $|\mathcal{I}_1| + 1$ rows, with each of them being a constraint on the realization weights at the respective information set, except for the first row, which represents the root constraint. Each column represents a possible action sequence of player$_1$ including the empty sequence ($\emptyset$) for a total of $|\mathcal{C}_1| + 1$ columns. The first variable of $\mathbf{x}$ represent the realization weight of the empty sequence ($\emptyset$). Hence, we have exactly $1\mathbf{x}_1 = \mathbf{e}_1 = 1$ that covers the constraint "root has weight 1". The remaining constraints resulting from $\mathbf{Ex} = \mathbf{e}$ cover the constraint "chidren sum to parent" for each of player$_1$'s information sets $I \in \mathcal{I}_1$. For example, let $I_1^1 \in \mathcal{I}_1$ denote the information set blue labeled as 1 in figure 13. Two action sequences are available at this information set $A(I_1^1) = \{A_1, D_1\}$. The parent sequence of these two is the empty sequence. The second linear constraint from $\mathbf{Ex} = \mathbf{e}$ describes exactly this parent child relation by $-\mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_5 = \mathbf{e}_2 = 0$, from which $\mathbf{x}_2$ and $\mathbf{x}_5$ represent the realization weights on A_1 and D_1, respectively.

$$\mathbf{F} = \begin{array}{c} \begin{array}{ccccccccccccc} \emptyset & a_1 & b_1 & c_1 & d_1 & a_2 & b_2 & c_2 & d_2 & a_3 & b_3 & c_3 & d_3 \end{array} \\ \left(\begin{array}{ccccccccccccc} 1 & & & & & & & & & & & & \\ -1 & 1 & 1 & & & & & & & & & & \\ -1 & & & 1 & 1 & & & & & & & & \\ -1 & & & & & 1 & 1 & & & & & & \\ -1 & & & & & & & 1 & 1 & & & & \\ -1 & & & & & & & & & 1 & 1 & & \\ -1 & & & & & & & & & & & 1 & 1 \end{array}\right) \end{array} \qquad \mathbf{f} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\mathbf{E} = \begin{array}{c} \begin{array}{ccccccccccccc} \emptyset & A_1 & A_1B_1 & A_1C_1 & D_1 & A_2 & A_2B_2 & A_2C_2 & D_2 & A_3 & A_3B_3 & A_3C_3 & D_3 \end{array} \\ \left(\begin{array}{ccccccccccccc} 1 & & & & & & & & & & & & \\ -1 & 1 & & & 1 & & & & & & & & \\ & -1 & 1 & 1 & & & & & & & & & \\ -1 & & & & & 1 & & & 1 & & & & \\ & & & & & -1 & 1 & 1 & & & & & \\ -1 & & & & & & & & & 1 & & & 1 \\ & & & & & & & & & -1 & 1 & 1 & \end{array}\right) \end{array} \qquad \mathbf{e} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Figure 11: Constraint matrices and vectors for Kuhn poker in Figure.13. Zeros in the matrices are omitted.

Analogously, $\mathbf{Fy} = \mathbf{f}$ can be interpreted in the same manner with the exception that they are constraints for the realization weights of player$_2$.

$\mathbf{A}$ is the payoff matrix of player$_1$ (Figure 12). In Kuhn poker and two-player zero-sum games in general, the payoff matrix of player$_2$ is simply $-\mathbf{A}$. $\mathbf{A}$ has $|\mathcal{C}_1|+1$ rows representing the action sequences of player$_1$, and $|\mathcal{C}_2|+1$ columns representing the action sequences of player$_2$. An entry of $\mathbf{A}$, of which the combined action sequences of both players form a terminal history z, contains the utility $u_1(z)$ of player$_1$. All other entries are zeros. For example, $\mathbf{a}_{33}$ shows player$_1$'s utility at the terminal history $z = c_{Q|J}A_1b_1B_1$. At z, player$_1$ has folded to player$_2$'s bet, thus lost the paid blind of one. However, the chance event $c_{Q|J}$ of z has a probability of $1/6$ that needs to be multiplied to the payoff of -1 resulting in a utility value of $-1/6$. $\mathbf{a}_{32}$ corresponds to an action sequence $c_{Q|J}A_1a_1B_1$, which is not a valid action sequence. Thus, the entry $\mathbf{a}_{32}$ is zero.

$$
\mathbf{A} = \begin{array}{c|cccc:cccc:cccc}
 & \emptyset & a_1 & b_1 & c_1 & d_1 & a_2 & b_2 & c_2 & d_2 & a_3 & b_3 & c_3 & d_3 \\
\emptyset & & & & & & & & & & & & & \\
A_1 & & 1/6 & & & & -1/6 & & & & & & & \\
A_1B_1 & & & -1/6 & & & & -1/6 & & & & & & \\
A_1C_1 & & & 1/3 & & & & -1/3 & & & & & & \\
D_1 & & & & 1/6 & 1/3 & & & 1/6 & -1/3 & & & & \\
\hdashline
A_2 & & & & & & -1/6 & & & & -1/6 & & & \\
A_2B_2 & & & & & & & -1/6 & & & & -1/6 & & \\
A_2C_2 & & & & & & & -1/3 & & & & -1/3 & & \\
D_2 & & & & & & & & 1/6 & -1/3 & & & 1/6 & -1/3 \\
\hdashline
A_3 & & 1/6 & & & & & & & & 1/6 & & & \\
A_3B_3 & & & -1/6 & & & & & & & & -1/6 & & \\
A_3C_3 & & & 1/3 & & & & & & & & 1/3 & & \\
D_3 & & & & 1/6 & 1/3 & & & & & & & 1/6 & 1/3
\end{array}
$$

Figure 12: Payoff matrix of Kuhn poker. Non terminal histories are omitted.

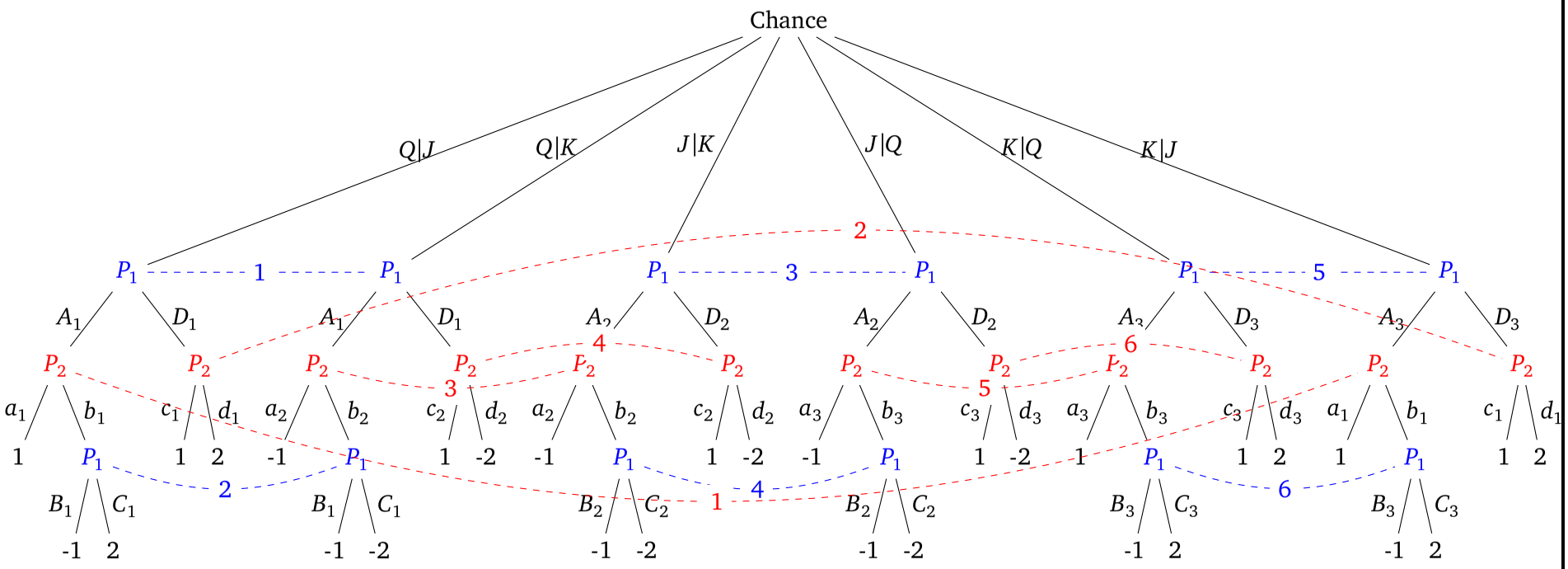

Figure 13: The game tree of Kuhn poker. Dashed paths connect nodes in the same information set. Blue denotes player$_1$ and red player$_2$. Terminal nodes are denoted with payoffs for player$_1$.

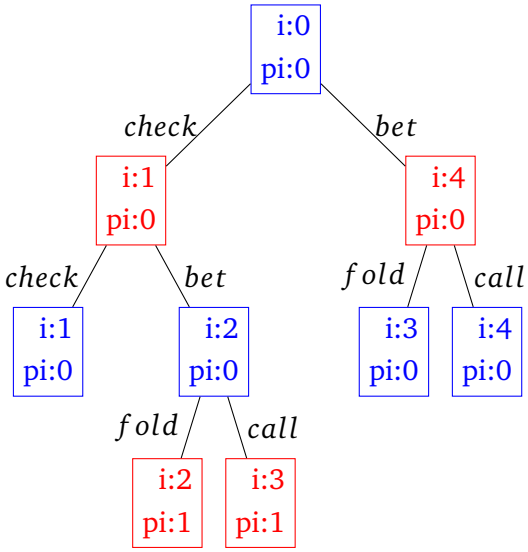

Figure 14: Public tree of Kuhn poker. Terminal nodes show the total pot size in chips.

However, it is not necessary to create the entire game tree for the LP generation. As can seen, each subtree from the root possesses the same structure. Further repeating patterns can be found in the constraint matrices **E** and **F** and the payoff matrix **A**. The dotted lines in figure 11, 12 highlight each subtree section in the matrices. It will be sufficient to generate a constraint matrix for only one subtree and subsequently use its pattern to create the full constraint matrix.

A subtree of the root has the same structure as the **public tree**, which is the tree from the view of an outside observer, who cannot see the private cards of either player. Thus, the public tree summarizes all subtrees of the original game tree into one tree. Each node of the public tree represents a set of game states that cannot be distinguished by the observer. The public tree can be constructed by using only the action abstraction.

We used the recursive node-to-node representation for the public tree. The tree was created in pre-order. Each node is labeled with a unique action sequence of player$_i$, who is currently acting at the node. Each action sequence is identified with a number, which is assigned to the respective node in the order of the traversal. Additionally, the index of the parent sequence is also assigned to each node. The public tree of Kuhn poker is depicted in figure 14, where i represents the index of the action sequence to which the node is assigned to, and pi represents the corresponding index of the parent action sequence. The pseudo-code for the public tree generation is given in Algorithm 10.

Having constructed the public tree, we can easily extract the constraint and payoff matrices from it using the indices stored in the nodes. The indices of the blue nodes correspond to the column numbers of player$_1$'s constraint matrix and also the row numbers of the payoff matrix. Analogously, the indices of the red nodes correspond to the column numbers of player$_2$'s constraint matrix and the payoff matrix. Remember that we are creating the matrices for the public tree, and not for the full game tree. However, our objective is to use the matrices of the public tree and its patterns to generate the matrices of the full game tree. The pseudo-code for the extraction of public tree matrices is provided in Algorithm 11. Note that we utilize two additional matrices, namely showdown matrix **SD** and terminal pot size matrix **P**, instead of one regular payoff matrix. The reason for this is that in a public tree we do not have the private card information to determine the exact payoffs. Thus, we can only store the pot size and a boolean for each terminal node, which tell us whether there was a showdown or not. These are also the

only observable information at the public tree. An exception is made for terminal nodes where a player has folded. Here, we can determine the winner at the terminal node, and thus also the payoffs. **SD** and **P** apply to all subtrees of the full game tree root. The only difference among individual subtrees is the associated chance event.

Everything is now set for the creation of the matrices **E**, **F** and **A** of the full game tree. Algorithm 12 describes the creation of **A**. This algorithm takes the joint bucket distribution **C** and the bucket equity table **T** as inputs, which have been created in the card abstraction computation. The bucket equity table is used for calculating the shares of the pot at terminal nodes with showdowns. The joint bucket distribution stores the probability of each possible chance event. We make use of the repeating patterns of the subtrees, iterate through all possible bucket constellations and fill in each entry of **A**. Algorithm 13 computes the full constraint matrix **E** or **F** depending on the inputs. It works in a similar manner as Algorithm 12 and also exploits the repeating pattern in the matrices.

The final step is to solve the linear program. In our implementation, we used the commercial LP-solver Gurobi optimizer to solve the LPs. During game-play we choose between (7) and (8) to solve depending on our position. If we are player_1, we choose (8), otherwise we choose (7).

4.3 Evaluation

In order to evaluate the endgame solver, we compare the performance of a Nash equilibrium agent with and without the endgame solving component. We call the agent without endgame solver **base agent** and the agent with endgame solver **endgame agent**. First, we have them compete directly against each other. Then, we will test their performances against other benchmark agents and compare their results.

4.3.1 Experimental Setup

Base Agent

The base agent is a static ϵ-Nash equilibrium agent trained by **Pure CFR** [13, p.70] (which belongs to the family of CFR algorithms) with an imperfect recall abstraction. The trained game is no-limit Texas hold'em with starting stacks of 25 big blinds. Each big blind has the value of 10$ while each small blind has the value of 5$. The buckets were calculated according to public card textures and k-means clustering over a hand strength distribution [15]. The card abstraction uses 169, 1286792, 60000 and 180000 buckets on each round of the game, respectively, with the first two rounds being unabstracted.

The action abstraction is symmetric and always allows fold, call and all-in at every decision node. The raise action is abstracted down to a number of bets relative to the pot size that include 0.25, 0.5, 0.75, 1, 1.5, 2 or 3 times the pot size. The action abstraction has a **commitment threshold** which defines the maximum amount of a bet size. Usually, the stack sizes define the maximum amount of a bet. Commitment thresholds are less than the stack sizes and prevent players making bets of the sizes close to their stack sizes. Those kinds of bets leave the players only a short amount of remaining stack, where players often cannot do anything else but call due to the high pot odds. Thus, removing those kinds of bets could decrease the size of the abstracted game without overly affecting the end strategy. Hence, if a bet size were to exceed the commitment threshold, it would be filtered out. The commitment threshold is set to be 65% of the stack size in the first betting round and 75% for other betting rounds. Furthermore, a soft action translation with geometric similarity metric is used (see 3.3.4). The resulting base agent has a size of nearly 20 gigabytes and 100 billion Pure-CFR-iterations were performed.

Endgame Agent

The endgame agent uses the base agent for the trunk part of the strategy. It uses one single predefined action abstraction for the endgame solving and adjusts the number of endgame card buckets during game-play accordingly. The action abstraction is symmetric as well, but finer granulated than the base agent's action abstraction. It allows more sizes for the initial bet than for raises with the intuition that initial bets are more common than raises. For initial bets it allows the following ten bet sizes expressed as fractions of the pot size: 0.1, 0.25, 0.375, 0.5, 0.625, 0.75, 1, 1.5, 2, 3. For raises we allow the same bet sizes as the base agent with the only difference being that there is no commitment threshold. This means that more bet sizes are allowed as no bets were filtered out. The endgame card buckets are created by using the percentile hand strength clustering while the top bucket threshold α is set to 0.98. The same action translation as base agent was used in the endgame.

Benchmark Agents

- **Always-Call-Bot** plays naively, always calls as its name suggests.

- **Always-Raise-Bot** plays naively, always raises to 0.5 of the pot size. If there has been an all-in, the agent then calls.

- **Random-River-Bot** plays according to our precomputed Nash equilibrium strategy for the first three rounds and bets randomly generated sizes in endgames in order to cause translation errors in his opponent.

- **ArizonaStu** [25] implements a list of expert rules and follows these. Additional opponent statistics are collected and these are used in the rules. A backup strategy is used if no expert strategy has been found. The backup strategy plays according to the expected hand strength of the current holdings.

Match Setup

All test matches are carried out in the form of duplicate games [23, 14]. When we run a match between two agents, we first play a series of games, with the cards being dealt according to a random number generator given a certain seed. Afterwards, the players switch their positions, and replay the same number of games with the same hole cards being dealt. Both players reset their settings (only necessary for dynamic agents) thus they do not remember the previous match. After both matches are finished, the total score is calculated. Therefore, both players received the same opportunities and luck. Playing duplicate games results in a huge decrease of the variance and therefore reduces the required number of games to determine the stronger agent.

Since the objective of the evaluation is to determine the performance of the endgame solver, we are only interested in hands where both agents actually reach an endgame. In our experiment, roughly 32% of hands made it to an endgame by playing according to the trunk strategy. We only count the results of this 32% of hands and discard other hands where either a player folded or both players went all-in before an endgame had started. Ganzfried et al.(2015)[9] has proven that this technique is unbiased.

Despite the already mentioned variance reduction techniques, it is still very time-consuming to determine the real performance of endgame solver with statistical significance, since the endgame solver requires five seconds to compute an endgame on average which is much slower than typical strategy table lookups. Thus, we can only run a very limited number of hands for the evaluation.

4.3.2 Experimental Results

Base Agent vs Endgame Agent

We managed to run four duplicate matches of 10,000 hands per match which means a total of 80,000 hands were played out. Out of these 80,000 hands, roughly 25,000 made it to an endgame. The duration of all matches combined was approximately 35 hours. The results of these four duplicate matches are provided in Table 2 where only hands that made it to an endgame have been considered.

The endgame agent yielded +24,120$ in 25,425 reached endgames which results in around 9.49 big blinds per 100 hands. This win-rate is relatively high and indicates a clear dominance of the endgame agent over the base agent. One reason for this dominance could be that the action abstraction of the endgame agent had caused many translation errors in the base agent due to having additional bet sizes, which happened to be in the exact middle of two bet sizes in base agent's action abstraction, causing maximum translation errors and off-tree problems (e.g. 0.375 is in the middle of 0.25 and 0.5).

Match	Sub Match	Result	#Endgames
1	1	−3737$	3257
	2	+7976$	3174
2	1	+3740$	3169
	2	+3278$	3142
3	1	−1336$	3229
	2	+3389$	3141
4	1	+8751$	3128
	2	+2059$	3185
Total		+24,120$	25,425

Table 2: Results by using the endgame agent against the base agent. Each match is a duplicate match consisting of two sub matches with switched hole cards.

Performance Comparison against Always-Call-Bot

We ran three duplicate matches of 3000 hands per match against Always-Call-Bot for the base agent and for the endgame agent. Since the callbot never folds, around 90% of hands actually reach an endgame, resulting in a similar number of played endgames as the previous duplicate matches. The results are depicted in Table 3.

Match	Sub Match	#Endgames	Base Agent	Endgame Agent
1	1		+22,715$	+20,967$
	2		+27,718$	+29,102$
2	1		+30,533$	+35,267$
	2		+31,645$	+32,755$
3	1		+26,956$	+39,141$
	2		+31,320$	+27,387$
Total		∼ 16,200	+170,907$	+184,619$

Table 3: Results against Always-Call-Bot. Around 2700 endgames were reached in each sub match, resulting around a total of 16,200 endgames.

The endgame agent performed slightly better than the base agent by yielding 13,712$ (∼ 7%) more profit. The result is less obvious than the direct comparison of both agents. Unlike the direct match-up against the endgame agent, the base agent is not confronted with any unusual bet sizes, therefore no off-tree situations occurred during the matches. Since both agents are Nash equilibrium based agents, their strategies are also very similar facing an always calling opponent. Hence, it is not surprising that the gap of the result is relatively small.

Performance Comparison against Random-River-Bot

Against Random-River-Bot we ran three duplicate matches of 10,000 hands per match, resulting in around 20,000 reached endgames (Table 4). As expected, the endgame agent performed better and won

13% more against Random-River-Bot than the base agent. Again, the result has proven the importance of correct bet size interpretation.

Match	Sub Match	#Endgames	Base Agent	Endgame Agent
1	1		+14,415$	+20,005$
	2		+24,233$	+26,654$
2	1		+29,239$	+22,590$
	2		+24,245$	+24,742$
3	1		+26,122$	+33,246$
	2		+19,332$	+31,364$
total		∼ 20,000	+137,586$	+158,583$

Table 4: Results against Random-River-Bot. Around 3300 endgames were reached in each sub match.

Performance Comparison against ArizonaStu

Against ArizonaStu we also ran three duplicate matches of 10,000 hands per match, resulting in a total of 25,000 endgames (Table 5). The endgame Agent yielded ∼ 24% more profit than the base agent.

Match	Sub Match	#Endgames	Base Agent	Endgame Agent
1	1		+15,715$	+10,067$
	2		+27,718$	+39,252$
2	1		+30,533$	+45,267$
	2		+25,645$	+29,056$
3	1		+23,112$	+29,141$
	2		+23,298$	+39,387$
total		∼ 25,000	+146,021$	+192,170$

Table 5: Results against ArizonaStu. Around 4200 endgames were reached in each sub match.

5 Enhancements of existing Nash Agents II: Endgame Exploitation

The primary objective of endgame solving is to improve the quality of the existing ϵ-Nash equilibrium strategy. It is an attempt to mitigate the negative effects of abstraction and approximate equilibrium finding. The desired outcome of the endgame solving is still a defensive and robust strategy, which is ideally less exploitable than the original. However, minimizing its own exploitability is not optimal against opponents with obvious weaknesses. A deviation from the equilibrium in order to exploit these weaknesses can achieve greater profits.

The ultimate objective of exploitation is to maximize an agent's utility against all types of opponents. The key part hereby is to quickly identify the opponent's strategy and subsequently develop an appropriate counter strategy. However, it is impossible to compute an opponent model and counter strategy for every opponent in advance. Thus, online opponent modeling and exploitation are essential for real life applications, where the agent will be facing numerous types of opponents. Sophisticated exploitation techniques such as RNR (3.4.1) and DBR (3.4.1) require large amounts of opponent data and computation time. Thus, they are computationally infeasible to be used online in real time.

Online opponent exploitation has been investigated in several prior works. Ganzfried et al.(2011)[12] developed an algorithm for opponent modeling by observing the opponent's action frequencies and building an opponent model by combining information from a precomputed equilibrium strategy with the observations. Afterwards, it computes an abstracted best response to the presumed opponent model. However, this algorithm was only applied to limit Texas hold'em with small abstractions. The precomputed approximate equilibrium strategy and all benchmark agents had only single digit numbers of card buckets which allows to perform online computation. However, for larger abstractions this algorithm also became impractical to use in real-time.

Bard et al.(2014)[4] introduced the concept of implicit opponent modeling (3.4.2). Rather than using observations to estimate a generative model of the other agent's behavior, implicit modeling summarizes their behavior through the expected utility of a portfolio of expert strategies. Basically, an agent uses a portfolio of precomputed strategies and selects the one with the highest expected utility against his current opponent out of this portfolio during game-play. The agent only has to evaluate the quality of each portfolio strategy online and does not have to perform any explicit opponent modeling or complex response calculation. Implicit modeling has proven to be successful in Bard's experiment and is regarded as the state of the art of efficient opponent exploitation technique. However, this approach relies on many components. The quality of each expert strategy in the portfolio and the strategy evaluation also determine the quality of the whole exploitation system. A lot of precomputation has to be performed offline before actually using it in real games.

In this section, we introduce an exploitation technique called endgame exploitation that is based on the idea of endgame solving. It tries to decompose the full game and to separately compute a solution for the endgame as well. Instead of calculating the exact equilibrium strategy by feeding the inputs, extracted from the trunk strategy, into the LP-solver, we also take into account the opponent model by manipulating the hand joint distribution. The result is a robust equilibrium strategy with respect to the given opponent model. Furthermore, we attempted to adjust the private hand joint distribution online according to our observations of the opponents. The first challenge herewith is how to quickly identify opponents' playing tendencies from the observations. The second challenge is how to correctly interpret these tendencies and incorporate them into the private hand distribution. We will present our attempts to overcome these challenges.

5.1 Endgame Exploitation Implementation

The basic concept of endgame exploitation is simple, as it is only a minor modification of the endgame solving algorithm. The difference lies in the computation of joint input distributions of private information. Let us briefly recap this aspect of the endgame solver. The first step of the endgame solving algorithm was to compute the joint input distribution D of private information. For instance, the entry $D[h_1][h_2]$ stores the probability of player$_1$ holding hole cards h_1 and player$_2$ holding hole cards h_2 at the same time. The joint probability represents the product of probabilities that each player would have taken the given action sequence (which led to the current endgame) according to the trunk strategy. Thus, our base assumption is that the opponent plays according to our trunk strategy as well. The joint hand distribution is then used for the computation of equity arrays, which are in turn used for card abstraction computation. Now, instead of entirely using the base strategy to compute the joint input distribution, we use the actual opponent model to determine the actual range of the opponent. The resulting joint input distribution D reflects the actual joint probabilities of each hole cards combination reaching the endgame according to the given action sequence.

From this point forward everything remains the same as in the endgame solving: the new card and action abstractions are computed and the abstracted endgame is solved with an LP-solver. Hence, we still compute an equilibrium strategy with the difference of having accurate opponent ranges as inputs. We do not explicitly exploit our opponent, but rather "correct" our initial inputs to match reality. As we still compute an equilibrium strategy, we make the pessimistic assumption that the opponent plays optimally in endgames. We expect that the endgame exploiting strategy is a tailored equilibrium strategy which slightly deviates from the real equilibrium in order to exploit the opponent. As a result, endgame exploit produces a conservative exploit which is still robust and relatively safe against counter-exploits.

5.1.1 Extracting Player Tendencies from Observations

In practice, the exact opponent model is usually unknown to us, as it would otherwise be more effective to compute one of the previous mentioned safe responses (e.g DBR) to exploit the opponent. Most of the time, the only information we have about our opponents are our observations collected from prior encounters. The number of observations is rather limited in practice. In the example in online poker, we will likely be facing unknown opponents constantly, thus we would only be able to collect about 100 to 200 hands on average against a particular opponent during the session. It is impossible to create a perfect opponent model based on such a small sample, not to mention that most of the observations would be even incomplete due to the hidden private cards of the opponent.

However, some useful information can still be obtained from even small samples. Instead of creating a detailed opponent model, we only gather general playing statistics of the opponent. This is a common approach among advanced human poker players at online poker tables, where they use tools (e.g Pokertracker [28]) to track opponent statistics. The statistics are statistical mean values that reflect how on average an opponent reacts to a certain situation (e.g. how often he raises or calls the preflop) [27]. Human players take the opponent statistics into consideration in their decision-making process, in order to be able to classify the general opponent type. We will now introduce some of the common statistics [27] which we also included in our endgame exploitation system:

- **VPIP (Voluntary put $ in Pot)** - represents the percentage of games a player actively put money into the pot on the preflop including calling and raising.

- **PFR (Preflop Raise)** - represents the percentage of games a player raised on the preflop.
- **Raise Frequency Flop/Turn/River** - represent the raise frequency of each betting round respectively.
- **Call Frequency Flop/Turn/River** - represent the call frequency (including checks) of each betting round respectively.

These general statistics do not require large sample sizes to converge. Especially VPIP and PFR do not change much more after only 100 games, as each game starts with the preflop. Hence, both statistics are being updated almost after each game. The higher the round number the more games are required for the corresponding statistics to converge, since the frequency of reaching decreases from round to round. We can use these statistics to estimate the opponent's player type. For example, a high VPIP indicates that the opponent plays a wide range of private hands. Conversely, we can assume that a low VPIP player plays a few but relatively strong private hands, if we consider the opponent to be reasonable.

The idea is to compare the statistics of our Nash equilibrium strategy with the statistics of our opponent. The difference between the statistics indicates how much the opponent deviates from the Nash equilibrium. Our objective is now to adjust the joint hand distribution according to this distance. However, it is not a trivial task to translate the deviation into a meaningful adjustment. The deviation only tells us whether the opponent has a bigger or smaller range of hands than us, but we still do not know its distribution. In the following section, we will describe the heuristic method we used to tackle this problem.

There remain some unanswered questions regarding our statistic comparison approach: which statistics should we use and how to calculate the deviation. Theoretically, we can create any kind of opponent statistics by conditioning the action frequencies on the exact public information and action history. For example, we can use statistics as precise as:

$$P_{opp}(raise|board = A\clubsuit 2\spadesuit 2\clubsuit K\spadesuit T\spadesuit, action\_sequence = (check, raise...))$$

that would perfectly describe a given situation, and we can directly compare this to our corresponding statistic. However, such a detailed conditioned statistic requires a large sample to converge, which clearly does not serve the purpose of online exploitation. Therefore, we have to rely on the more general but less precise statistics like VPIP, PFR etc. In order to calculate the opponent's private hand range reaching an endgame, we made the simplifying assumption that statistics are independent. Then, we simply multiply all statistics that when put together, best possibly match the current situation. For example, consider the following actions made by the opponent to reach the river (endgame), which we denote as s :

PREFLOP	FLOP	TURN
raise	check call	check

$$\begin{aligned} PFR = P_{opp}(raise|preflop) &= 0.2 \\ Flop\_Call\_Freq = P_{opp}(call, check|flop) &= 0.4 \\ Turn\_Call\_Freq = P_{opp}(call, check|turn) &= 0.5 \\ P_{opp}(s) = PFR \cdot Flop\_Call\_Freq^2 \cdot Turn\_Call\_Freq & \\ 0.2 \cdot 0.4 \cdot 0.4 \cdot 0.5 = 0.016 & \end{aligned}$$

$P(s)$ is the probability that the opponent plays according to action sequence s. In other words, he could have 1.6% of his starting hands at this point. Note that these statistics are not conditioned on the board, which means that they summarize over all possible public boards. Therefore, the same action sequence on a different board would have the same result. Now, we can compare $P_{opp}(s)$ to $P_{us}(s)$ to estimate the magnitude of the deviation. Suppose $P_{us}(s) = 0.008$. The probability of the opponent reaching the given endgame is twice as high as ours. In order to match this deviation, we need to expand our Nash equilibrium range to twice of its original size by assigning higher probabilities to some of our private hands. Figure 15 illustrates the difference between two private hand distributions.

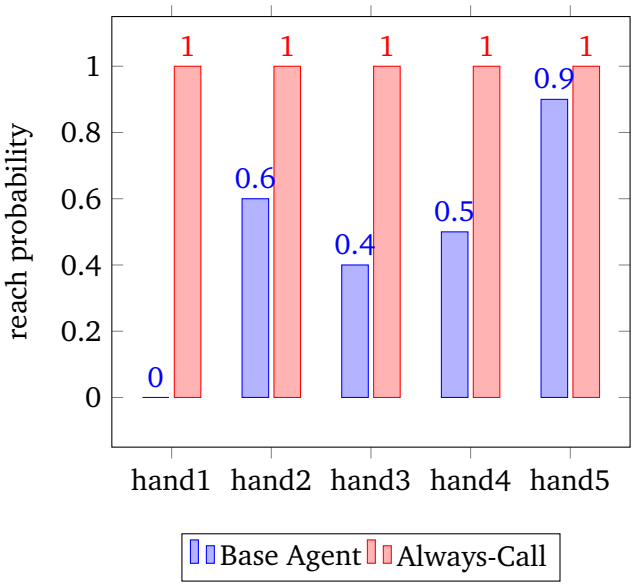

Figure 15: There are five possible private hands. The bars represent the probability reaching the endgame while holding the corresponding hand. Base agent needs to increase the reach probabilities of all hands in order to match Always-call.

5.1.2 Adjustment of prior Hand Distribution according to the statistics distance

So far, we have estimated the difference in reach probabilities between the opponent's strategy and our Nash equilibrium strategy. We now know how much probability we have to add (or to remove if we have a higher probability than the opponent) to our private hand distribution to fit the opponent. However, we do not know which hand probabilities we should modify. Some kind of assumptions about the opponent's behavior must be made. The first assumption we made is that the opponent is at least somewhat reasonable. This means that he rather plays overall stronger hands than he plays weaker hands. The second assumption is that the opponent stays as close to the Nash equilibrium as possible. Ideally, if the Nash equilibrium hand distribution could be expressed as a continuous function from expected hand strength to probability, we could modify the function parameters to fit the deviation. For example, in the case of a Gaussian distribution, we could tune the mean and variance parameters. Unfortunately, the Nash equilibrium cannot be translated into a reasonable function of expected hand strength ($\mathbb{E}[HS]$) to probability (see Figure 16). One possible reason for that could be that the $\mathbb{E}[HS]$-feature alone is not sufficient to describe the problem. A multidimensional feature space may be required to characterize the behavior of the Nash equilibrium strategy. Apart from the challenge of finding the "right" features, machine learning techniques would also be necessary in order to accurately estimate the

relationships between those features. However, further investigations are required and go beyond the scope of this work.

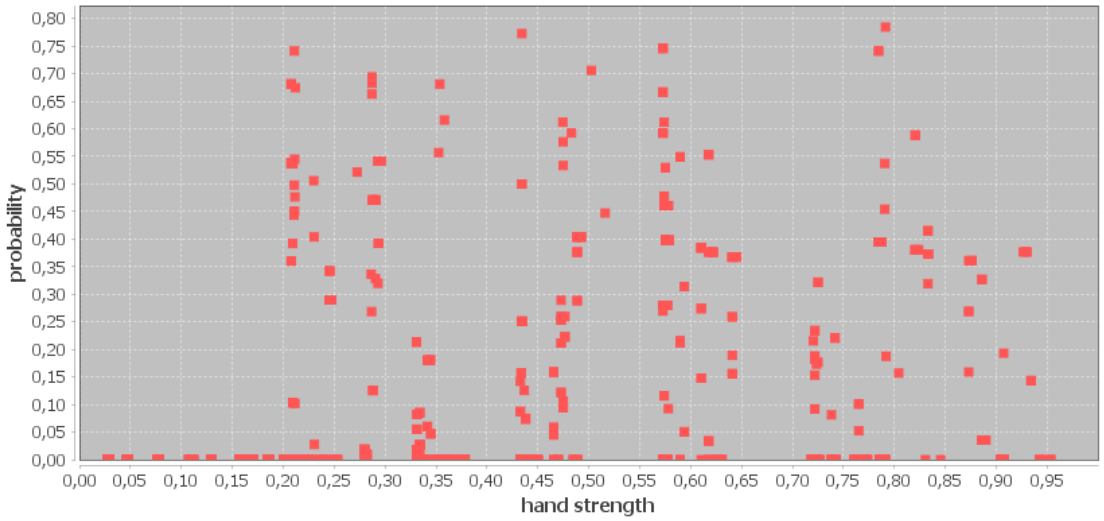

Figure 16: Example of a Nash equilibrium hand distribution at an endgame. Each point represents a potential private hand. x-axis shows the hand strength of a hand and y-axis shows its probability being played.

In our experiments, we used a simple method to adjust the hand distribution. In the case of range expansion, we sort all private hands by their respective reach probabilities (according to the Nash equilibrium) in descending order, and then greedily keep filling up the probability of each hand until our range reaches the same size as the range of the opponent. Since we want to stay as close to the Nash equilibrium as possible we add additional probabilities to hands which already have a higher probability of being played. For example, in figure 15, the filling order would be "hand_5, hand_2, hand_4, hand_3, hand_1". If our range were still not large enough after filling up the hand with the lowest positive probability (> 0), we will sort the remaining hands by their averaged $\mathbb{E}[HS]$-value over all prior betting rounds and again greedily fill up the probability starting with the hand with the highest value. The $\mathbb{E}[HS]$-value in each betting round prior to endgame has an influence on the reach probability of a hand, since $\mathbb{E}[HS]$ changes with each additional community card. Therefore, we used averaged $\mathbb{E}[HS]$-values instead of single round $\mathbb{E}[HS]$-values as the hand feature. The idea here is based on the first assumption that the opponents rather play stronger hands than weaker hands, which is the reason we first fill up probabilities of hands with higher averaged $\mathbb{E}[HS]$-value. In the case of range reduction, we instead sort hands in ascending order and keep greedily removing probability until the hand range size equals the opponent's hand range size.

But first, we need to collect some data in order to calculate the opponent's hand range deviation from the Nash equilibrium. The regular endgame solving approach is used in the first 100 games when facing an unknown opponent. Then, we measure the size of the deviation by comparing the opponent's statistics in those initial games to the statistics of the base agent. We only consider exploiting the opponent, if the deviation is significant enough, because exploitation can be risky and opens oneself up to potential counter-exploits. In other words, we only exploit opponents with obvious weaknesses, where we can gain more by exploiting. Therefore, a threshold is set, which defines the size of the deviation for us to start exploiting.

5.2 Evaluation

In order to evaluate the endgame exploitation, we first integrated exact opponent models into our endgame solver and let it compete against the same benchmark agents used in our previous experiments. Then, we compared the empirical results before and after incorporating opponent models. We ran the same amount of duplicate matches against Always-Call-Bot and Always-Raise-Bot. Due to the lack of exact opponent models, we only used these two bots in this experiment. Both of these benchmark agents could have any holdings at any time, thus we simply use a uniform distribution to represent their hand distribution. Table 6, 7 present the results of both match-ups.

Match	Sub Match	#Endgames	Endgame Solving	Endgame Exploit
1	1	-	+20,967$	+22,253$
	2	-	+29,102$	+28,977$
2	1	-	+35,267$	+40,805$
	2	-	+32,755$	+33,145$
3	1	-	+39,141$	+42,047$
	2	-	+27,387$	+28,536$
total	-	$\sim 16,200$	+184,619$	+195,763$

Table 6: Results against Always-Call-Bot.

Match	Sub Match	#Endgames	Endgame Solving	Endgame Exploit
1	1	-	+208,576$	+209,343$
	2	-	+200,396$	+208,827$
2	1	-	+202,686$	+204,896$
	2	-	+203,540$	+227,415$
3	1	-	+204,102$	+222,038$
	2	-	+199,897$	+209,564$
total	-	$\sim 18,300$	+1,219,197$	+1,282,083$

Table 7: Results against Always-Raise-Bot.

As expected, integrating opponent models improved the performance. However, it is only a minor improvement of approximately $3-6\%$ higher profits, despite the fact that the opponents were only baseline agents and we used perfect opponent models. The results are likely going to be worse against non-trivial opponents with estimated opponent models.

Next, we employed the opponent modeling approach from the previous section (5.1.1) in order to evaluate the endgame exploitation with estimated opponent models. We increased the number of initial information gathering games from 100 to 1000. The exploitation threshold was set to $1.5x$, implying that we only start exploiting when the opponent's hand range size is 1.5 times of ours which was always the case when competing against Always-Call and Always-Raise bots. The results (depicted in Table 8, 9) against these two benchmark agents were nearly identical to the prior experiments with perfect

opponent models. Since both benchmark agents always take the same action regardless of the board and history, the statistics and the deviation from Nash equilibrium could be calculated accurately.

Match	Sub Match	#Endgames	Endgame Solving	Endgame Exploit (estimated models)
1	1	-	+20,967$	+22,066$
	2	-	+29,102$	+29,177$
2	1	-	+35,267$$	+39,100$
	2	-	+32,755$	+32,472$
3	1	-	+39,141$	+41,845$
	2	-	+27,387$	+30,026$
total	-	$\sim 16,200$	+184,619$	+194,686$

Table 8: Results against Always-Call-Bot.

Match	Sub Match	#Endgames	Endgame Solving	Endgame Exploit (estimated model)
1	1	-	+208,576$	+208,110$
	2	-	+200,396$	+209,021$
2	1	-	+202,686$	+205,098$
	2	-	+203,540$	+225,118$
3	1	-	+204,102$	+222,969$
	2	-	+199,897$	+210,054$
total	-	$\sim 18,300$	+1,219,197$	+1,280,361$

Table 9: Results against Always-Raise-Bot.

However, the results against ArizonaStu (depicted in Table 10) were less satisfactory. Endgame exploit with estimated opponent model could not yield more profit than the basic endgame solving without any exploitation. Both yielded nearly the same amount of profit against ArizonaStu. Due to the inaccuracy in the statistics, the deviations from our Nash equilibrium could not always be calculated accurately which might probably be one of the reasons why the performance had not improved. Since we are using very general statistics that are averaged over all possible public boards, the actual hand range on some specific boards can be very different. For example, the statistic given a draw heavy board (such as 7♠8♠9♠) is often completely different than given a paired board with nearly no draws (such as A♠2♣2♢). Another probable reason could be that our hand distribution adjustments were incorrect.

In conclusion, we may state that the endgame exploitation approach has proven to be less practical. In order to model a non-trivial opponent, more information is required instead of completely relying on certain short-term general statistics and simple opponent assumptions. For instance, the opponent modeling could be improved by evaluating complete observations which would also include opponent's private hand information. However, only games with showdowns are fully observable, therefore more observations are required in general. In order to make online opponent modeling/exploitation perform effectively, some prior general opponent models should first be constructed to overcome the challenge of lacking samples (e.g implicit modeling). Another issue of endgame exploitation is its poor exploitative

power (only $3-6\%$ profit increase) even when competing against naive baseline bots given exact opponent models, thus making it overall a less attractive choice for exploitation than other prior approaches.

Match	Sub Match	#Endgames	Endgame Solving	Endgame Exploit (estimated model)
1	1	-	+10,067$	+19,432$
	2	-	+39,252$	+34,232$
2	1	-	+45,267$	+49,910$
	2	-	+29,056$	+35,187$
3	1	-	+29,141$	+20,544$
	2	-	+39,387$	+29,020$
total	-	$\sim 25,000$	+192,170$	+188,325$

Table 10: Results against ArizonaStu.

6 Enhancements of existing Nash Agents III: Asymmetric Action Abstraction

The standard approach for solving very large extensive-form games is to use abstraction techniques to reduce the number of game states in order to scale it to the available computational resources. An ideal abstracted game should retain as much of the information of strategic structure as possible. In the domain of no-Limit Texas hold'em, using card abstraction alone is not sufficient to make tractable abstracted games. The main part of the immense size of no-limit games results from the large number of action options available to the players. Therefore, an adequate action abstraction is always required to restrict the action options.

The granularity of an abstraction determines the size and also affects the precision of the resulting abstracted game. Typically, the finest possible granularity is selected given the computation time constraints and the computational resources to guarantee a best possible solution. However, each player in a multi-agent environment needs to be abstracted. Hence, the question arises as to how the computational resources should be distributed among the agents. For example, in a two-player game the action abstraction for the agent of interest determines the possible actions our agent could take. The action abstraction of the opponent player represents what we believe what kind of actions he could take. The sum of both abstraction sizes is the size of the entire abstracted game. This opens up several different design variations of the abstraction. We could utilize either a symmetric design where all players share the same type of abstraction, or an asymmetric design where a finer granulated abstraction is used for the agent of interest than for other agents and vice versa. Evidently, using finer granulated abstraction for one side means less fine granulated abstraction for the other side given limited computational resources. The choice of the design is very important. Most AI developers use symmetric abstractions as default plan for their agents. One obvious advantage of symmetric abstraction over asymmetric abstraction is its simplicity. While a symmetric abstraction simply requires a game to be solved once to compute all players' strategies, asymmetric abstraction requires a game to be solved twice in the case of two-player games, once for each arrangement of the players' abstractions [5, p.41]. For instance, if we would choose an asymmetric design with two raise sizes for our agent and four raise sizes for the opponent, we would have to first solve a game, where the small blind has two and the big blind has four raise sizes, and then solve another game where the small blind has four and the big blind has two raise sizes.

It has been demonstrated in prior works that the abstraction choice clearly affects the performance of the agent. Bard et al.(2014)[3] investigated asymmetric card abstractions for limit Texas hold'em. They observed a trade-off in two common performance measures of poker agents, one-on-one performance against various agent types and exploitability (performance against a worst-case opponent), when using asymmetric abstraction designs. However, they focused on card abstractions while their empirical evidence was based on smaller poker games(eg. Leduc Poker, limit Texas hold'em), where unabstracted actions can be used. It is not obvious that the same properties apply to action abstractions as well. Although there have been participants in the Annual Computer Poker Competition [1], who already implemented asymmetric action abstraction in their agents, there has been little to none empirical analysis on the effect of asymmetric action abstractions on the quality of the resulting behaviors in no-limit games.

In this chapter, we investigate asymmetric action abstractions in the domain of two-player no-limit Texas hold'em via empirical analysis. We examine how the abstraction trade-off affects an agent's performance in both one-on-one field performance and its worst-case exploitability. The performance metrics are based on similar metrics used in [3] with the exception that we compared strategies with different

action abstractions rather than card abstractions. However, computation of worst-case exploitability is a difficult task in no-limit Texas hold'em and presents a more challenging task than in the limit variant of Texas hold'em (which is normally used for experiments in prior works). Therefore we start this chapter off with a description of our own exploitability computation method. Then, we present our empirical results and provide a guidance for future action abstraction choices.

6.1 Exploitability Calculation

Given a poker strategy σ_{opp}, the strongest counter-strategy to σ_{opp} is the best response $b(\sigma_{opp})$. The average utility of the best response against σ_{opp} is the worst-case exploitability of σ_{opp}. Calculating a best response to a given strategy is very computationally expensive. A best response has the same size as an unabstracted solution of the game, which clearly does not fit into the memory of modern computers in the domain of no-limit Texas hold'em. Instead of calculating the actual best response and exploitability in the full game, developers compromise by calculating a best response strategy in an abstracted game. We call the exploitability in an abstracted game **abstracted exploitability**, which also serves as the lower bound for the real exploitability.

At every information set, the abstracted game best response chooses the action that maximizes its utility, given the probability of σ_{opp} reaching every terminal node descendant from every game state in the information set and the utility associated with that terminal node. The pseudo-code of the abstracted best response is given in Algorithm 1 [23, p.56].
Algorithm 1 describes the standard approach for calculating an exact abstracted best response. However, this approach has some drawbacks. The algorithm can only produce a best response in the same abstraction as the opponent strategy. It is not possible to use different abstractions for the best response and its opponent. In our experiments, we want to create equal testing environments for all benchmark strategies. We want to calculate a best response in the same abstraction for different benchmark strategies with different abstractions in order to compare their exploitabilities. Thus, the best response algorithm must be able to handle different abstractions.

Another drawback is that Algorithm 1 is based on the perfect recall assumption. However, most no-limit game abstractions use imperfect recall abstractions. In imperfect recall abstraction, hands are grouped differently in each betting round and forget their bucket associations in the past rounds. This means that an agent may be able to distinguish two information sets early in a game, but not distinguish their descendant information sets later in the game. They will perceive the game as a directed acyclic graph instead of a tree. This imperfect recall assumption makes the recursion in the algorithm impossible. In order to use Algorithm 1, all strategies must use perfect recall buckets, which is rarely used in state of the art poker agents.

In order to avoid the aforementioned problems, we use an alternative method to compute the abstracted best response. Instead of exactly computing it, we use an iterative approximation approach. For this purpose, we modify the standard CFR-algorithm. The standard CFR-algorithm is based on self play, where both players iteratively optimize their strategies until they converge to a Nash equilibrium. In our modified CFR-algorithm, one of the players (the opponent) is given a predefined strategy, which stays constant during the whole computation. The strategy of the best response player is iteratively improved and converges to an abstracted best response strategy against the constant strategy of the opponent. The opponent can use any abstraction, which does not have to be the same abstraction the best response player uses. The opponent strategy can be viewed as a black box, which receives the current game state as input and outputs the corresponding action. The best response player's abstraction can be chosen

Algorithm 1: Abstracted Best Response

```
1  BestResponse (A, I, S);
   Input: An abstraction of the game A
          A node I of the information set tree for game A
          A strategy σ_opp being a mapping from information sets to action probabilities
          A strategy BR(σ_opp) being a mapping from information sets to action probabilities
          An expected utility u(BR(σ_opp), σ_opp) being the expected utility of using BR(σ_opp) against
   σ_opp
2  if I is a terminal node then
3      Let u_I be the utility of reaching information set I;
4      Let p be the probability of σ_opp reaching game state g ∈ I;
5      Let u_g be the utility of reaching game state g ∈ I;
6      u_I = 1/(Σ_{g∈I} p) (Σ_{g∈I} p u_g);
7      return u_I
8  else if I is a choice node for BR(σ_opp) then
9      Let I(a) be the information set resulting from taking action a in information set I;
10     Find the action a that maximizes u(a) = BestResponse(A, I(a), σ_opp);
11     Set BR(σ_opp)(I) to select action a;
12     return u_a
13 else if I is a choice node for σ_opp then
14     for each action a available in I do
15         Let I(a) be the information set resulting from taking action a in information set I;
16         BestResponse(A, I(a), σ_opp);
```

freely. Unlike the traditional best response computation in Algorithm 1, our method does not require both players to use the same abstracted game. Since we are approximating the best response iteratively through simulation, we no longer have to recursively traverse the game tree, which also allows us to use imperfect recall buckets. The abstracted exploitability of the opponent strategy is the best response player's average utility per hand, which (like the best response strategy itself) converges after a sufficient amount of iterations. This approach can be seen as a concrete implementation of the restricted Nash response (RNR 3.4.1) with the parameter p set to zero.

6.2 Experimental Setup

Throughout our asymmetric action abstraction experiments, we use the game of no-limit Texas hold'em with ten big blinds starting stacks as our test environment. Nine approximate Nash equilibrium strategies were constructed using different pairs of action abstractions for the agent strategy and the opponent's response strategy. All strategies use the same card abstraction to highlight the performance difference between action abstraction variations. In our experiment, we used percentile equal sized bucket clustering (see 4.2.2) over the expected hand strength ($\mathbb{E}[HS]$) and expected hand strength squared ($\mathbb{E}[HS^2]$) card features except for the first betting round, which is virtually unabstracted, where each of 169 canonical hands is considered a single bucket (see 3.3.1). Each of the next three betting rounds has 1000 buckets, respectively. On each betting round, all possible hands are equally distributed into those 1000 buckets

according to their $\mathbb{E}[HS]$ or $\mathbb{E}[HS^2]$ values. The abstraction we investigate has the imperfect recall property, in which hands are reassigned to a new bucket in each betting round and forget their buckets in all earlier rounds.

Each strategy uses a unique action abstraction. While all strategies allow *fold*, *call* and *all-in* at every decision, the number of *raise* options varies. Each player's strategy has either 1, 2, 4 or 8 different bet sizes, which are shown in Table 11. The CFR-algorithm was used to generate a strategy for each pair of abstractions, which includes three symmetric and six asymmetric abstractions. The benchmark strategies are relatively small because our card abstraction is rather coarse compared to the finest possible abstraction that our hardware is able to handle. The small abstraction allows faster computation of the strategy. We solely spent twelve hours to compute the largest benchmark strategy (symmetric and 8 bet sizes).

As mentioned previously, we use exploitability and one-on-one field performance to measure the quality of a strategy. Each strategy's abstracted exploitability was computed using the modified CFR-algorithm described in Section 6.1, where the best response player always uses a symmetric action abstraction with eight bet sizes and the same card abstraction. In this category, the lower the exploitability value the better is the strategy. However, lower exploitability does not mean that the strategy is automatically superior. A better worst-case performance may lead to worse performance against normal opponents, similar to the trade-off between making oneself unexploitable and exploiting opponent with the risk of being counter-exploited. Thus, one-on-one field performance measure is also necessary, especially since worst-case opponents are almost non-existent in practice. The one-on-one performance between each pair of strategies was measured by playing one million duplicate games of poker (two millions games total). The results were measured in milli-big-blinds per game.

Action Abstration	Bet Sizes (fractions of pot)
1	1
2	0.5, 1
4	0.5, 1, 1.5, 2
8	0.25, 0.5, 0.75, 1, 1.5, 2.5, 3

Table 11: Available bet sizes as fractions of the money in pot.

6.3 Empirical Results

We computed a total of nine strategies with different action abstractions. The results of our experiments are presented in Table 12. Each strategy has a label "U-O" which indicates which action abstraction was used for *us* and for the *opponent*. For example, "8-1" is a strategy using an asymmetric action abstraction where our strategy uses 8 different bet sizes and assumes the opponent is using only one bet size. Accordingly, "8-8" is a strategy using a symmetric action abstraction where both players use the same 8 bet sizes. Column "Avg" shows the average profit of the corresponding strategy against the field. The "Exploitability" column shows the abstracted exploitabilities where the abstracted best response uses the 8-bet sizes action abstraction.

Our experiments have similar results to those in the prior work [3] about asymmetric card abstractions in the symmetric abstraction case. By increasing the action abstraction size (from 1-1 to 8-8), the

	8-8	4-4	1-1	2-1	4-1	8-1	1-2	1-4	1-8	Avg	Exploitability	Size
8-8	-	-3	52	49	31	31	32	47	37	31	1	105 MB
4-4	3	-	45	41	19	23	30	52	48	29	245	15 MB
1-1	-52	-45	-	-15	-38	-35	-5	1	10	-20	329	2 MB
2-1	-49	-41	15	-	-21	-23	1	11	16	-10	405	4 MB
4-1	-31	-19	38	21	-	0	26	43	41	13	697	5 MB
8-1	-31	-23	35	23	0	-	25	35	35	11	539	14 MB
1-2	-32	-30	5	-1	-26	-25	-	10	13	-10	272	4 MB
1-4	-47	-52	-1	-11	-43	-35	-10	-	4	-22	146	5 MB
1-8	-45	-48	-10	-16	-41	-29	-13	-4	-	-23	6	14 MB

Table 12: Cross table of approximate Nash equilibria strategies using the same card abstraction but different action abstractions. The row player's expected value is shown in milli-big-blinds per game, rounded to the nearest integer.

average utility against the field and exploitability improve as expected. However, the performance improvement seems to cap at a certain degree of granularity. While average utility against the field and exploitability improves significantly in the step from 1-1 to 4-4, the step from 4-4 to 8-8 only improves the exploitability. 8-8 even loses to 4-4 in direct comparison. This phenomenon confirms the results in Waugh et al.(2009)[36] on abstraction pathology where they demonstrated that it is not guaranteed that an increase of abstraction size would lead to a performance improvement, despite the fact that they only examined card abstractions in their work.

Now, we move on to the asymmetric cases. First, examining the asymmetric abstractions where the opponent's action abstraction is larger than our agent's action abstraction, we observe that the exploitabilities of 1-4 and 1-8 are lower than the exploitability of the symmetric 4-4. Even the smallest strategy of this category 1-2 has an only slightly higher exploitability than 4-4, even though 4-4 is more than three times of 1-2's size. However, their average utilities against the field are worse compared to the symmetric variants. If the primary objective is to minimize the worst-case exploitability, we should use a larger abstraction for the opponent than for our agent.

As we increase the size of the opponent's action abstraction, the exploitability improves. Increasing the opponent's action abstraction size implies providing the opponent with more betting options to exploit oneself. In other words, a more pessimistic assumption about the opponent is made. Therefore, the resulting strategy becomes more defensive and its worst-case performance improves. The one-on-one performance becomes worse when we increase the opponent's action abstraction size. There seems to be a trade-off between exploitability and one-on-one performance. In order to minimize the worst-case exploitability we have to sacrifice our one-on-one utility.

Continuing with the reversed case where we have a larger action abstraction than our opponent, we observe a different change of exploitability and one-on-one performance as we increase our abstraction size. The empirical results are not as obvious and clean as the ones from previous cases. A clear trade-off between exploitability and one-on-one performance is not directly observable. However, from 2-1 to 4-1 we observe an opposite trend in trade-off where the one-on-one performance improves and the exploitability gets worse as we increase our action abstraction size. From 4-1 to 8-1, we usually would

	Total Avg Utility	Total Exploitability
U>O	14	1641
U<O	-55	424

Table 13: Comparison of both asymmetric variants

expect the trend to continue which however did not occur in our experiment. Surprisingly, the one-on-one performance slightly worsened and exploitability improved. A possible explanation for this might be that we have reached a performance cap and experienced the abstraction pathology effect. Another explanation could be that the used exploitability measure is biased since we used the same 8 bet sizes for the best response player in the abstracted exploitability computation as our action abstraction in 8-1. Nonetheless, we can conclude that increasing our action abstraction size favors the one-on-one performance while the exploitability is overall worse than the other asymmetric case which is highlighted in Table 13.

Overall, our empirical results on varying action abstractions are similar to the results in [3] on asymmetric card abstractions. We observed a trend of decreasing exploitability and one-on-one field performance when increasing the action abstraction granularity of the opponent, and conversely, a opposite trend when increasing the abstraction granularity of the agent of interest. The symmetric action abstraction design has overall good results in our experiments. Comparing strategies with the same magnitude, the one with symmetric action abstraction has the strongest one-on-one field performance. However, if the best possible worst-case performance were requested, one should choose an asymmetric design where the opponent uses a larger abstraction. If worst-case opponents are unlikely and the overall competition is weak, one should choose a symmetric design or a design where the one's own agent gets a finer granulated action abstraction for more exploitative possibilities.

7 Conclusion

In this thesis, we examined the promising approach of endgame solving for improving existing poker agents. Endgame solving solves the last portion of the game that is actually reached during game-play to a greater degree of accuracy than the existing base agent strategy. It takes into account the action history and public information of the endgames and assumes a more realistic opponent private hand distribution (rather than the standard uniform random hand distribution) for the computation. Beside a detailed review of the first work [9] about endgame solving we pointed out potential pitfalls and limitations of the original approach and proposed alternatives regarding the card and action abstraction computations. Additionally, we provided a detailed description and explanation of the LP generation process based on an example of a toy poker game. We also provided our own empirical results, which have confirmed the performance improvement by applying endgame solving. In a direct comparison of a Nash equilibrium agent with endgame solving implemented against itself without endgame solving, the one with endgame solving yielded a high winrate of more than 9bb/100. In a series of experiments against various benchmark agents, the agent with endgame solving also outperformed the base agent.

Then, we proposed a modification to the standard endgame solving, where we integrated opponent models into endgame solving that allows us to exploit opponents. We named it the endgame exploitation and evaluated the performance of this modification, which has slightly improved compared to regular endgame solving against simple benchmark agents. Furthermore, we attempted to create an opponent modeling technique that we incorporated in the endgame exploitation. This opponent modeling technique measures the deviation of opponent's strategy to our base Nash equilibrium strategy based on statistics evaluation. The statistics provide information about the general opponent behavior in common situations (e.g. how often does the opponent raise preflop). The utilized statistics are quickly converging and thus allow us to quickly identify the opponent's player type. Based on the statistics difference between the opponent and our base agent, we adjusted the joint hand distribution to match the statistics of the opponent. We managed to achieve the same results against the simple benchmark agents with our estimated model as with the exact models. However, the results against a stronger agent did not improve compared to only using endgame solving. Overall, we conclude that the current endgame exploiting approach is not really practical, because even under the unrealistic premise of having a perfect opponent model, endgame exploitation still could not improve much against weak oppositions compared to normal endgame solving.

Our last contribution was to provide the first empirical analysis of asymmetric action abstraction in the domain of no-limit Texas hold'em. In addition, we proposed a technique to estimate the exploitability of no-limit Texas hold'em playing agents with imperfect recall abstractions. We modified the standard CFR algorithm to iteratively approximate the exploitability instead of exactly computing the exploitability like the commonly used best response. Our results presented a similar trade-off between worse-case and one-on-one performance to prior observations of asymmetric card abstraction in limit Texas hold'em [3]. The exploitability decreases when we increase the granularity of opponent's action abstraction. When we increase the granularity of our action abstraction, the exploitability increases while the one-on-one performance improves.

There are several areas where we can expand the scope of this work. For endgame solving, it could be interesting to include the "turn" betting round as additional part of the endgame instead of only using the "river". It would result in a massive increase of the endgame size, but also likely improve the endgame strategy. The challenge is then to abstract the endgame down to a tractable size without losing too

much relevant information. It would be interesting to utilize different endgame sizes and compare their performances and computational requirements.

Endgame exploitation can be further modified to use best response instead of Nash equilibrium, which is more effective against known static opponents. The opponent modeling based on statistics deviations can also be improved. Apart from a better statistics selection, there has to be a more intelligent way to adjust the joint hand distribution than greedily filling up/removing the probabilities of hands to match the opponent.

There are also some unanswered questions regarding asymmetric action abstraction. We have seen that an increase of action abstraction size does not necessarily improve the performance of the resulting strategy due to abstraction pathology. However, there has to be a connection between the granularity of the abstraction and the size of the stack-to-blind ratio. The idea here is that smaller stacks require less fine-grained abstractions than larger stacks. Further experiments and analyses are required to verify this theory.

References

[1] Annual Computer Poker Competition, "http://www.computerpokercompetition.org/"

[2] D. Arthur and S. Vassilvitskii, "k-means++: the advantages of careful seeding", in: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 2007

[3] N. Bard, M. Johanson and M. Bowling "Asymmetric Abstractions for Adversarial Settings", in: *Proceedings of the 13th International Conference on Autonoumous Agents and Multiagent Systems (AAAMAS 2014)*, 2014.

[4] N. Bard, M. Johanson, N. Burch and M. Bowling "Online Implicit Agent Modeling", in: *Proceedings of the 12th International Conference on Autonoumous Agents and Multiagent Systems (AAAMAS 2013)*, 2013.

[5] N. Bard, "Online Agent Modelling in Human-Scale Problems", PhD Thesis, 2016.

[6] R.E Bellman, "On the application of dynamic programming to the determination of optimal play in chess and checkers", in: *National Academy of Sciences of the United States of America*, 1965.

[7] D. Billings, N. Burch, A. Davidson, R. Holte, T. Schauenberg and D. Szafron "Approximating Game-Theoretic Optimal Strategies for Full-scale Poker", in: *Proceedings of the 18th International Joint Conference on on Artificial Intelligence(IJCAI)*, 2003.

[8] Brains vs. Artificial Intelligence No-limit Hold'em Challenge, "https://www.cs.cmu.edu/brains-vs-ai"

[9] S. Ganzfried and T. Sandholm, "Endgame Solving in Large Imperfect-Information Games", Carnegie Mellon University, in: *Proceedings of the 14th International Conference on Autonoumous Agents and Multiagent Systems (AAAMAS 2015)*, 2015.

[10] S. Ganzfried, "My Reflections on the First Man vs. Machine No-Limit Texas hold'em Competition", Carnegie Mellon University, 2015.

[11] S. Ganzfried and T. Sandolm, "Action translation in extensive-form games with large action spaces: Axioms, paradoxes, and the pseudo-harmonic mapping", Carnegie Mellon University, in: *Proceedings of the International Joint Conference on Artificial Intelligence(IJCAI)*, 2013.

[12] S. Ganzfried and T. Sandolm, "Game theory-based Opponent Modeling in Large Imperfect-Information Games", in: *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems(AAMAS 2011)*, 2011.

[13] R. Gibson "Regret Minimization in Games and the Development of Champion Multiplayer Computer Poker-Playing Agents", PhD Thesis, 2014.

[14] A. Gilpin and T. Sandolm, "Lossless abstraction of imperfect information games", in: *Journal of the ACM*, 2007.

[15] A. Gilpin, T. Sandholm, and T. B. Sørensen, "Potential-aware automated abstraction of sequential games, and holistic equilibrium analysis of Texas Hold'em poker", in: *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 2007.

[16] Gurobi Optimization Inc, Gurobi optimizer reference manual version 6.5, "http://www.gurobi.com/documentation/", 2016.

[17] E. Jackson "Slumbot NL: Solving Large Games with Counterfactual Regret Minimization Using Sampling and Distributed Processing", in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013)*, 2013

[18] T. Kanungo, D. Mount "An Efficient *k*-Means Clustering Algorithm: Analysis and Implementation", 2002

[19] D. Koller, N. Meggido and B. von Stengel, "Fast algorithms for finding randomized strategies in game trees", in: *Proceedings of the 26th ACM Symposium on Theory of Computing(STOC)*, p. 750-760, 1994.

[20] H. W. Kuhn, "A Simplified Two-Person Poker", in: *Contributions to the Theory of Games 1*, pp. 97-103, 1950.

[21] M. Johanson, N. Burch, R. Valenzano and M. Bowling, "Evaluating State-Space Abstractions in Extensive-Form Games", in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems(AAMAS 2013)*, 2013.

[22] M. Johanson, "Measuring the Size of Large No-Limit Poker Games", University of Alberta, 2013.

[23] M. Johanson, "Robust Strategies and Counter-Strategies: Building a Champion Level Computer Poker Player", MSc. Thesis, University of Alberta, 2007.

[24] M. Johanson and M. Bowling, "Data Biased Robust Counter Strategies", in: *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 2009)*, 2009.

[25] E. Loza Mencia and J. Prommer "ArizonaStu (KEmpfer)", in: *Participants of Annual Computer Poker Competition: Heads-up No-limit Texas Hold'em 2014*, "http://www.computerpokercompetition.org/", 2014

[26] M. Pfetsch, "Einführung in die Optimierung", 2013

[27] Pokerstrategy, Stats Basics, "https://www.pokerstrategy.com/strategy/various-poker/stats-basics/1/"

[28] Pokertracker, "https://www.pokertracker.com/"

[29] J. F. Nash, "Non-cooperative games" in:*Annuals of Mathematics*, 1951

[30] A. Risk and D. Szafron, "Using Counterfactual Regret Minimization to Create Competitive Multiplayer Poker Agents", in: *Proceedings of the 9th International Conference on Autonoumous Agents and Multiagent Systems (AAAMAS 2010)*, 2010.

[31] J. Rubin and I. Watson, "Computer poker: A review" in: *Elsevier Artificial Intelligence(2011)*, 2011

[32] D. Schnizlein, "State Translation in No-Limit Poker", MSc. Thesis. University of Alberta, 2009.

[33] D. Sklansky "The Theory of Poker", Two Plus Two Publishing, 4th edition, 1999

[34] D. Silver A. Huang C.J. Maddison A. Guez L. Sifre and co., "Mastering the game of Go with Deep Neural Networks and Tree Search", in: *Nature 2016*, 2016.

[35] O. Tammelin, N. Burch, M. Johanson, and M. Bowling, "Solving Heads-up Limit Texas Hold'em", in: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[36] K. Waugh, "Abstraction in Large Extensive Games", MSc. Thesis, 2009.

[37] K. Waugh, M. Zinkevich, M. Johanson, M. Kan, D. Schnizlein, and M. Bowling "A practical use of imperfect recall", in: *Proceedings of the Eighth Symposium on Abstraction, Reformulation and Approximation (SARA)*, 2009.

[38] K. Waugh, "A Fast and Optimal Hand Isomorphism Algorithm" in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013)*, 2013.

[39] K. Waugh, M. Zinkevich, M. Bowling and M. Johanson, "Accelerated Best Response Calculation in Large Extensive Games", in: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence(IJCAI)*, 2011.

[40] M. Zinkevich, M. Bowling and M. Johanson, and C. Piccione, "Regret minimization in games with incomplete information", in: *Proceedings of the Annual Conference on Neural Information Processing Systems(NIPS)*, 2007.

Appendices

A Pseudocode

Algorithm 2: Compute index of 7-card hands on a given board [9, p.5]

IndexSeven (h_1, h_2, B);
Input : Private hole cards h_1 and h_2, board B consisting of five community cards
Output: An index value between 0 and 1080
if $h_2 < h_1$ **then**
  $t \leftarrow h_1$; $h_1 \leftarrow h_2$; $h_2 \leftarrow t$;
$n_1 \leftarrow 0$; $n_2 \leftarrow 0$;
for $i = 1$ **to** 5 **do**
  for $j = 1$ **to** 2 **do**
    if $B[i] < h_j$ **then**
      n_j++;
return $\binom{h_2 - n_2}{2} + \binom{h_1 - n_1}{1}$

Algorithm 3: Compute index of 2-card hands [9, p.5]

IndexTwo (h_1, h_2);
Input : Private hole cards h_1 and h_2
Output: An index value between 0 and 1325
if $h_2 < h_1$ **then**
  $t \leftarrow h_1$; $h_1 \leftarrow h_2$; $h_2 \leftarrow t$;
return $\binom{h_2}{2} + \binom{h_1}{1}$

Algorithm 4: Compute joint distribution [9, p.5]

Input: Board B; number of possible private hands H; betting history h; index conflicts array IC; base strategy s^*

D_1,$D_2 \leftarrow$ array of dimension H of zeroes;
$D \leftarrow H \times H$ matrix of zeroes;
for $p_1 = 0$ **to** 50, $p_1 \notin B$ **do**
  for $p_2 = 1$ **to** 51, $p_2 \notin B$ **do**
    $I \leftarrow$ IndexSeven(p_1,p_2,B);
    IndexMap[I] $\leftarrow$ IndexTwo(p_1,p_2);
    $P_1 \leftarrow$ probability player$_1$ would play according to h with p_1,p_2 in s^*;
    $P_2 \leftarrow$ probability player$_2$ would play according to h with p_1,p_2 in s^*;
    $D_1[I] += P_1$;
    $D_2[I] += P_2$
Normalize D_1 and D_2 **for** $i = 0$ **to** H **do**
  for $j = 1$ **to** H **do**
    if *!IC[IndexMap[i]][IndexMap[j]]* **then**
      $D[i][j] \leftarrow D_1[i] \cdot D_2[j]$
    else
      $D[i][j] \leftarrow 0$
Normalize D so all entries sum to 1 **return** D

Algorithm 5: Compute equity arrays

Input: Joint hands distribution $D[][]$; number of possible private hands H; hand ranks $R[]$

e_1,$e_2 \leftarrow$ array of dimension H of zeroes;
$D \leftarrow H \times H$ matrix of zeroes;
for $h_1 = 0$ **to** H **do**
  $r_1 \leftarrow R[h_1]$;
  s_1, $s_2 \leftarrow 0$;
  for $h_2 = 1$ **to** H **do**
    $r_2 \leftarrow R[h_2]$, $s_2 \mathrel{+}= D[h_1][h_2]$, $s_2 \mathrel{+}= D[h_2][h_1]$;
    if $r_2 < r_1$ **then**
      $e_1[h_1] \mathrel{+}= D[h_1][h_2]$, $e_2[h_1] \mathrel{+}= D[h_2][h_1]$
    else if $r_2 == r_1$ **then**
      $e_1[h_1] \mathrel{+}= \frac{D[h_1][h_2]}{2}$, $e_2[h_1] \mathrel{+}= \frac{D[h_2][h_1]}{2}$
  if $s_1 > 0$ **then**
    $e_1[h_1] \leftarrow \frac{e_1[h_1]}{s_1}$
  else
    $e_1[h_1] \leftarrow -1$
  if $s_2 > 0$ **then**
    $e_2[h_1] \leftarrow \frac{e_2[h_1]}{s_2}$
  else
    $e_2[h_1] \leftarrow -1$

Algorithm 6: Percentile hand strength clustering

Input: Equity arrays e_i; number of possible private hands H; maximum number of buckets per agent k_i; top bucket equity threshold α

$J \leftarrow \frac{\alpha}{k_1}$;
$A_1 \leftarrow$ array of zeroes of size H;
$U_1 \leftarrow$ array of false of size H;
for $h = 0$ **to** H **do**
    if $e_1[h] \geq \alpha$ **then**
        $b \leftarrow k_1 - 1$
    else
        $b \leftarrow \lfloor \frac{e_1[h]}{J} \rfloor$
    if $U_1[b] == FALSE$ **then**
        $U_1[h] \leftarrow TRUE$
$M_1 \leftarrow$ array of zeroes of size k_1;
$g \leftarrow 0$;
for $i = 0$ **to** k_1 **do**
    $M_1[i] \leftarrow g$;
    if $U_1[i] == TRUE$ **then**
        $g = g + 1$
for $h = 1$ **to** H **do**
    if $e_1[h] < 0$ **then**
        $A_1[h] = -1$
    else
        if $e_1[h] \geq \alpha$ **then**
            $A_1[h] \leftarrow M_1[k_1 - 1]$
        else
            $A_1[h] \leftarrow M_1[\lfloor \frac{e_1[h]}{J} \rfloor]$
Compute A_2 analogously

Algorithm 7: Percentile hand strength clustering

Input: Equity arrays E_i; Hand index array $Indices$, number of possible private hands H; number of buckets per agent k_i

$A_i \leftarrow$ array of zeroes of size H;
Sort $Indices$ in ascending order according to E_i;
$HandsPerBucket \leftarrow$ array of $\lfloor \frac{H}{k_i} \rfloor$ of size k_i;
for $i = 0$ **to** $H - k_i \cdot HandsPerBucket[0]$ **do**
    $HandsPerBucket[i]++$;
$h \leftarrow 0$;
for $i = 0$ **to** k_i **do**
    for $j = 0$ **to** $HandsPerBucket[i]$ **do**
        $A_i[Indices[h]] \leftarrow i$;
        $h++$;
Compute A_2 analogously

Algorithm 8: Compute buckets joint distribution

Input : Joint hands distribution $D[][]$; number of possible private hands H; number of player_1 buckets N; number of player_2 buckets M; private hands to bucket mapping for player_1 $B_1[]$; private hands to bucket mapping for player_2 $B_2[]$;

Output: Joint buckets distribution B

$C \leftarrow N \times M$ of zeroes;
for $h_1 = 0$ **to** H **do**
    for $h_2 = 0$ **to** H **do**
        $b_1 \leftarrow B_1[h_1]$;
        $b_2 \leftarrow B_2[h_2]$;
        if $b_1 < 0$ *or* $b_2 < 0$ **then**
            continue
        $B[b_1][b_2] \leftarrow C[b_1][b_2] + D[h_1][h_2]$
return C

Algorithm 9: Compute bucket equities

Input : Joint hands distribution $D[][]$; hand ranks $R[]$; number of possible private hands H; number of player$_1$ buckets N; number of player$_2$ buckets M; private hands to bucket mapping for player$_1$ $B_1[]$; private hands to bucket mapping for player$_2$ $B_2[]$;

Output: Bucket equities T

$T \leftarrow N \times M$ of zeroes;
$Counters \leftarrow N \times M$ of zeroes;
for $h_1 = 0$ **to** H **do**
    for $h_2 = 0$ **to** H **do**
        if $D[h_1][h_2] \leq 0$ **then**
            continue
        $b_1 \leftarrow B_1[h_1]$;
        $b_2 \leftarrow B_2[h_2]$;
        $Counters[b_1][b_2]++$;
        if $R[h_1] > R[h_2]$ **then**
            $T[b_1][b_2] \leftarrow T[b_1][b_2] + 1$
        else if $R[h_1] = R[h_2]$ **then**
            $T[b_1][b_2] \leftarrow T[b_1][b_2] + 0.5$
for $b_1 = 0$ **to** N **do**
    for $b_2 = 0$ **to** M **do**
        if $Counters[b_1][b_2] = 0$ **then**
            $T[b_1][b_2] \leftarrow 0$
        else
            $T[b_1][b_2] \leftarrow \frac{T[b_1][b_2]}{Counters[b_1][b_2]}$
return T

Algorithm 10: Create the public tree given an action abstraction

createPublicTree $(s, abs, pi_1, pi_2, i_1, i_2)$;

Input : Game state s
parent sequence index pi_1 and pi_2
sequence index i_1 and i_2
global sequence counter $numSeq_1$ and $numSeq_2$

Output: A node

if *player*$_1$ *is acting* **then**
  $numSeq_1 \leftarrow numSeq_1 + 1$;
else
  $numSeq_2 \leftarrow numSeq_2 + 1$;
if *s is terminal* **then**
  if *player*$_1$ *is acting* **then**
    return $node(i_1, pi_1)$
  else
    return $node(i_2, pi_2)$
if *player*$_1$ *is acting* **then**
  $n \leftarrow node(i_1, pi_1)$
else
  $n \leftarrow node(i_2, pi_2)$
for *each a in* $A(s)$ **do**
  $s' \leftarrow doAction(s, a)$;
  if *player*$_1$ *is acting* **then**
    $child \leftarrow createPublicTree(s', abs, i_0, pi_1, numSeq_1, numSeq_2)$
  else
    $child \leftarrow createPublicTree(s', abs, pi_0, i_1, numSeq_1, numSeq_2)$
  $n.addChild(child)$
return n

Algorithm 11: Extract matrix information from public tree

```
1  extractMatrices (n, i);
   Input: public tree node n
          Parent node index i
          Showdown matrix SD[][]
          Terminal pot size matrix P[][]
          Public tree constraint matrix of player_1 E[][]
          Public tree constraint matrix of player_2 F[][]
          Information set node counter c_1, c_2 = 0
2  if n is a terminal node then
3      if player_1 was last to act then
4          if player_1 has folded then
5              P[n.index][i] ← −player_1.spent;
6          else
7              P[n.index][i] ← n.pot;
8              SD[n.index][i] ← true;
9      else
10         if player_2 has folded then
11             P[i][n.index] ← player_2.spent;
12         else
13             P[i][n.index] ← n.pot;
14             SD[i][n.index] ← true;
15     return
16 if player_1 is acting then
17     E[c_1][n.first_child.pi] ← −1;
18     for each child of n do
19         E[c_1][child.index] ← 1;
20     c_1 ← c_1 + 1
21 else
22     E[c_2][n.first_child.pi] ← −1;
23     for each child of n do
24         E[c_2][child.index] ← 1;
25     c_2 ← c_2 + 1
26 for each child of n do
27     extractMatrices(child,n.index);
```

Algorithm 12: Create payoff matrix of the game tree

createFullPayoffMatrix;
Input: Showdown matrix $SD[][]$
Terminal pot size matrix $P[][]$
Joint bucket distribution $Pr[][]$
Buckts equity table $Eq[][]$
Number of buckets of each player b_1, b_2
Number of action sequences in public tree a_1, a_2
totalRows $\leftarrow b_1 \cdot a_1 + 1$;
totalCols $\leftarrow b_2 \cdot a_2 + 1$;
$A \leftarrow$ matrix of totalRows $\times$ totalCols of zeros;
for $i \leftarrow 0$; i **to** b_1; **do**
  for $j \leftarrow 0$; j **to** b_2; **do**
    $row \leftarrow a_1 \cdot i + 1$;
    $col \leftarrow a_2 \cdot j + 1$;
    $pr \leftarrow Pr[i][j]$;
    $eq \leftarrow Eq[i][j]$;
    for $k \leftarrow 0$; k **to** a_1; **do**
      for $l \leftarrow 0$; l **to** a_2; **do**
        if $SD[k][l] == true$ **then**
          $A[row+k][col+l] \leftarrow pr \cdot (eq \cdot P[k][l] - \frac{P[k][l]}{2})$;
        else
          $A[row+k][col+l] \leftarrow pr \cdot P[k][l]$;
return A

Algorithm 13: Create constraint matrix of player$_i$ for the game tree

createFullPayoffMatrix;

Input: Constaint matrix of player$_i$ in public tree $C_i[][]$
Number of buckets of each player$_i$ b_i
Number of action sequences of player$_i$ in public tree a_i
Number of information sets of player$_i$ in public tree s_i

totalRows $\leftarrow b_i \cdot a_i + 1$;
totalCols $\leftarrow b_i \cdot s_i + 1$;
$E \leftarrow$ matrix of totalRows $\times$ totalCols of zeros;
for $k \leftarrow 0$; j **to** b_i; **do**
    $row \leftarrow s_i \cdot k + 1$;
    $col \leftarrow a_i \cdot k + 1$;
    for $l \leftarrow 0$; l **to** s_i; **do**
        for $m \leftarrow 0$; m **to** $a_i + 1$; **do**
            if $m == 0$ **then**
                $E[row + l][0] \leftarrow C_i[l][m]$;
            else
                $E[row + l][col + m - 1] \leftarrow C_i[l][m]$;
return E