Solving Pasur Using GPU-Accelerated Counterfactual Regret Minimization*

Sina Baghal
siinabaghal@gmail.com

August 7, 2025

Abstract

Pasur is a fishing card game played over six rounds and is played similarly to games such as Cassino and Scopa, and Bastra [1]. This paper introduces a CUDA-accelerated computational framework for simulating Pasur, emphasizing efficient memory management. We use our framework to compute near-Nash equilibria via Counterfactual Regret Minimization (CFR), a well-known algorithm for solving large imperfect-information games.

Solving Pasur presents unique challenges due to its intricate rules and the large size of its game tree. We handle rule complexity using PyTorch CUDA tensors and to address the memory-intensive nature of the game, we decompose the game tree into two key components: (1) actual game states, and (2) inherited scores from previous rounds. We construct the *Full Game Tree* by pairing card states with accumulated scores in the *Unfolding Process*. This design reduces memory overhead by storing only essential strategy values and node connections. To further manage computational complexity, we apply a round-by-round backward training strategy, starting from the final round and recursively propagating average utilities to earlier stages. Our approach constructs the complete game tree, which on average consists of over 10^9 nodes. We provide detailed implementation snippets to illustrate the structure and training logic of our framework and the CFR algorithm.

After computing a near-Nash equilibrium strategy, we train a tree-based model to predict these strategies for use during gameplay. We then estimate the fair value of each deck through large-scale self-play between equilibrium strategies by simulating, for instance, 10,000 games per matchup, executed in parallel using GPU acceleration. Our analysis reveals that the distribution of high-value cards heavily influences match outcomes, accounting for much of the variation in deck fairness. Finally, each trained model is lightweight, making it feasible to be used as a real-time AI agent for Pasur.

Similar frameworks can be extended to other reinforcement learning algorithms in settings where the action tree naturally decomposes into multiple rounds such as turn-based strategy games or sequential trading decisions in financial markets.

*This preprint is available at URL: https://sinabaghal.github.io/files/SolvePasurviaCFR.pdf
[1]Wikipedia: Pasur (card game)

1 Introduction and Background

Imperfect-information games pose significant challenges due to hidden information and strategic uncertainty. Counterfactual Regret Minimization (CFR) has emerged as a powerful algorithmic framework for approximating Nash equilibria in such settings, notably achieving success in poker variants and other complex card games [7, 3, 1, 2, 6]. By iteratively minimizing regret over possible strategies, CFR converges towards equilibrium policies without exhaustive search, making it an attractive method for large-scale sequential games.

In this work, we focus on *Pasur*, a traditional fishing card game popular in Middle Eastern cultures, with a six-round structure and unique scoring rules. Each round, players receive new cards while aiming to capture valuable cards from the table through matching and summation mechanics. Unlike many extensively studied games, Pasur has not been rigorously solved, in part due to its combinatorial complexity and memory-intensive game tree.

In this paper, we present a CUDA-accelerated implementation of CFR in PyTorch tailored to Pasur, addressing its computational challenges by decomposing the game tree into game states and accumulated scores. Our approach computes the complete game tree, enabling near-Nash equilibrium calculations rather than relying on sampling methods. To further optimize GPU memory usage, we train the CFR models round-by-round in reverse order, propagating average utilities to ensure tractability. At any given time, only the tensors for the current round are kept on the GPU; all others are stored in CPU memory. Our carefully designed data structures made it possible to train and develop the entire CFR algorithm on a game tree with an average size of 10^9, all within the constraints of a system with 32 GB of RAM and 10 GB of VRAM. After computing a near-Nash equilibrium strategy, we train a tree-based model to calculate the strategies for use during gameplay. These tree-model also facilites the estimation of deck fair values through self-play.

This framework not only advances the strategic understanding of Pasur and fishing card games more broadly, but also enables the development of practical AI agents capable of playing at near-optimal levels. Moreover, this document details the construction and update procedures of every step in the framework through CodeSnippets, including the implementation of the CFR algorithm.

Several clarifications about the CodeSnippets are in order. The CodeSnippets aim to convey as much detail as possible; however, certain operations are omitted for clarity. These include routine tasks such as data type conversions and the use of `.clone()` to avoid in-place modifications, if necessary. Additionally, we omit the `torch.` prefix in all code expressions. For example, `torch.any(t_act[:,1,:], dim=1)` is written as `any(t_act[:,1,:], dim=1)`. It is also worth noting that not all lines of code are shown in the CodeSnippets. For complete implementations, we refer the reader to the GitHub repository[2]. Finally, we use symbolic notation to simplify certain expressions. For instance, instead of `t_1.repeat_interleave(t_2)`, we write `t_1`$\otimes$`t_2`. Similarly, `t_1`$\otimes_1$`t_2` denotes the dimension-specific form

```
t_1.repeat_interleave(t_2, dim=1).
```

[2]Paper Homepage

1.1 Extensive Games and CFR

In *Imperfect-Information Extensive-Form Games* there is a finite set of players ($\mathcal{P}$). A *node* h at time T is defined by all the information revealed at a , particularly the actions taken by players $\mathcal{P}$. *Terminal nodes*, denoted by $\mathcal{Z}$, are defined as those nodes where no further action is available. For each player $p \in \mathcal{P}$, there is a payoff function $u_p : \mathcal{Z} \to \mathbb{R}$. In this paper, we focus on the *zero-sum* two-player setting *i.e.,* $\mathcal{P} = \{0, 1\}$ and $u_0 = -u_1$.

Imperfect information is represented by *information sets* (infosets) for each player $p \in \mathcal{P}$. It is emphasized that at player p's turn at infoset I, all nodes $h, h' \in I$ are identical from p's perspective. In this situation, we say infoset I belongs to p and we denote the set of all such infosets by $\mathcal{I}_p$. Actions available at infoset I are also denoted by $A(I)$. A *strategy* $\sigma_p(I) : A(I) \to \mathbb{R}^{\geq 0}$ is a distribution over actions in $A(I)$. The strategy of other players is denoted by σ_{-p}. We denote by $u_p(\sigma_p, \sigma_{-p})$ the expected payoff for p if players' actions are governed by strategy profile $\sigma := \{\sigma_p\}_{p \in \mathcal{P}}$.

Reach probability $\pi^\sigma(h)$ is defined as the probability of arriving at node h, if all players play according to σ. For $I \in \mathcal{I}_p$ and $h \in A(I)$, we denote by $\pi^\sigma_{-p}(h)$ the probability of arriving at h in the event where p chooses to reach h and other players follow σ. Define $\pi^\sigma_p(I) := \sum_{h \in I} \pi^\sigma_p(h)$ and $\pi^\sigma_{-p}(I) := \sum_{h \in I} \pi^\sigma_{-p}(h)$. *Counterfactual utility* at infoset $I \in \mathcal{I}_p$ is defined as

$$u^\sigma(I) := \sum_{h \in I, h' \in \mathcal{Z}} \pi^\sigma_{-p}(h) \pi^\sigma(h, h') u_p(h') \tag{1}$$

Similarly, counterfactual utility for $a \in A(I)$, $u^\sigma(I, a)$ is defined as in (1), except that p chooses a with probability 1 once it reaches I. Formally, if $h.a$ denotes the node wherein action a is chosen at node h, then

$$u^\sigma_p(I, a) := \sum_{h \in I, h' \in \mathcal{Z}} \pi^\sigma_{-p}(h) \pi^\sigma(h.a, h') u_p(h') \tag{2}$$

Finally, in a two-player extensive game a *Nash equilibrium* [4] is a strategy profile $\boldsymbol{\sigma}^*$ for which the following holds

$$u_p(\sigma^*_p, \sigma^*_{-p}) = \max_{\sigma'_p} u_p(\sigma'_p, \sigma^*_{-p}).$$

In other words, σ_p is the *best response* to σ_{-p} for each $p \in \mathcal{P}$. An ϵ-Nash equilibrium (in a two-player game, for example) is also defined as

$$u_p(\sigma^*_p, \sigma^*_{-p}) + \epsilon \geq \max_{\sigma'_p} u_p(\sigma'_p, \sigma^*_{-p}), \quad \forall p \in \{0, 1\}.$$

We are now ready to provide an overview of CFR next; for a complete discussion, see Zinkevich et al. (2007).

CFR is an iterative algorithm that converges to a Nash equilibrium in any finite two-player zero-sum game with a theoretical convergence bound of $O\left(\frac{1}{\sqrt{T}}\right)$. At the

heart of CFR lies the concept of *regret*. For a strategy profile σ, *instantaneous regret* of playing a vs. σ at $I \in \mathcal{I}_p$ is denoted by

$$r_p(I,a) := u_p^{\sigma}(I,a) - u_p^{\sigma}(I) \tag{3}$$

In CFR, a *regret matching* (RM) is performed at each iteration. According to RM, at iteration T, $\sigma_p^{T+1}(I)$ is determined using regrets (3) accumulated up to time T. Formally,

$$\sigma_p^{T+1}(I) \propto R_+^T(I,a) := \left(\sum_{t=1}^{T} r^t(I,a)\right)_+ \tag{4}$$

$R_+^T(I,a)$ is called *counterfactual regret* for infoset I action a. Under update rule (4), average strategy is then defined as follows

$$\bar{\sigma}_p^T(I)(a) \propto \sum_{t=1}^{T} \pi_p^{\sigma^t}(I)\, \sigma_p^t(I)(a). \tag{5}$$

The following two well-established results show that under RM (4), the average strategy (5) converges to a Nash equilibrium in zero-sum two-player games.

Theorem 1 *In a zero-sum game, average strategy* (5) *is a* 2ϵ*-Nash equilibrium provided that the total regret which is defined below is less than* ϵ *for* $p \in \{0,1\}$*.*

$$R_p^T := \max_{\sigma_p'} \frac{1}{T} \sum_{t=1}^{T} \left(u_p(\sigma_p', \sigma_{-p}^t) - u_p(\sigma_p^t, \sigma_{-p}^t)\right)$$

The following theorem states that $R_p^T \to 0$ as $T \to +\infty$ under RM (4).

Theorem 2 (Theorem 3 & 4, [7]) *Under RM* (4)*, the following bound holds*

$$R_p^T = \mathcal{O}\left(\frac{1}{\sqrt{T}}\right).$$

Moreover, total regret is lower bounded by counterfactual regrets defined in (4)*.*

There are different variants of CFR where the difference is in the way it updates counterfactual regrets. For example, in CFR$^+$ [5], regrets, initialized at zero, are updated according to the following rule:

$$R^{+,t}(I,a) = \left(R^{+,t-1}(I,a) + u^{\sigma^t}(I,a) - u^{\sigma^t}(I)\right)_+$$

Another example that is also used in this paper is the Discounted CFR (DCFR) [2], where prior iterations when determining both regrets and the average strategy are discounted. The update rule for DCFR is as follows:

$$R_p^t(I,a) = R_p^{t-1}(I,a) \cdot d_p^{t-1}(I,a) + r_p^t(I,a) \tag{6}$$

where

$$d_p^t(I,a) = \begin{cases} \dfrac{t^\alpha}{t^\alpha + 1} & \text{if } R_p^t(I,a) > 0 \\ \dfrac{t^\beta}{t^\beta + 1} & \text{otherwise} \end{cases} \tag{7}$$

Moreover, the average strategies are updated according to the following rule:

$$\bar{\sigma}_p^t(I)(a) \propto \left(1 - \tfrac{1}{t}\right)^\gamma \cdot \bar{\sigma}_p^{t-1}(I)(a) + \pi_p^{\sigma^t}(I)\sigma_p^t(I)(a) \tag{8}$$

1.2 Pasur

Pasur is a traditional card game played with a standard 52-card deck (excluding jokers), and it supports 2 to 4 players. In this paper, we focus on the two-player variant, where the players are referred to as **Alex** and **Bob**, along with a **Dealer** who manages the game.

The game begins with four cards placed face-up on the table to form the initial pool. This pool must not contain any Jacks. If a Jack appears among the initial four cards, it is returned to the deck and replaced. If multiple Jacks are dealt, or if a replacement card is also a Jack, the dealer reshuffles and redeals.

Once the pool is valid and face-up, the dealer deals four cards to each player, starting with the player on their left (assumed to be Alex). Players then take turns beginning with Alex. On each turn, a player must play one card from their hand. The played card will either: Be added to the pool of face-up cards, or Capture one or more cards from the pool, following rules described in Table 1. If a capture is possible, the player *must* capture; they cannot simply add a card to the pool. Captured cards are retained and used to calculate each player's score at the end.

Card Type	**Capture Rule**
Numeric	One or more numeric cards from the pool if their total sum equals 11
Jack	All cards in the pool, except Kings and Queens (can also capture other Jacks)
Queen	A single Queen
King	A single King

Table 1: Capture rules for each card type

A **Sur** occurs when a player captures all the cards from the pool in a single move. There are two important exceptions: (1) a Sur cannot be made using a Jack, and (2) Surs are not permitted during the final round of play. Table 2 outlines Pasur's scoring system, and Figure 1 provides a gameplay example.

Rule	**Points**
Most Clubs	7
Each Jack	1
Each Ace	1
Each Sur	5
10♦	3
2♣	2

Table 2: Pasur Scoring System

Pasur: A Step-by-Step Gameplay Snapshot

Stage	Alex	Bob	Pool	Lay	Pick	Acl	Bcl	Apt	Bpt	Asr	Bsr	Δ	L	CL
0_0_0	4♣ 4♢ 7♡ Q♣	3♢ 3♡ 5♣ K♠	A♣ A♠ 9♢ K♢	A♡		0	0	0	0	0	0	0	0	-
0_0_1	4♣ 7♢ Q♣	3♢ 3♡ 5♣ K♠	A♣ A♠ 4♢ 9♢ K♢	K♠	K♢	0	0	0	0	0	0	0	B	-
0_1_0	4♣ 7♢ Q♣	3♢ 3♡ 5♣	A♣ A♠ 4♢ 9♢	Q♣		0	0	0	0	0	0	0	B	-
0_1_1	4♣ 7♢	3♢ 3♡ 5♣	A♣ A♠ 4♢ 9♢ Q♣	5♣	A♣ A♠ 4♢	0	2	0	2	0	0	0	B	-
0_2_0	4♣ 7♢	3♢ 3♡	9♢ Q♣	4♣		0	2	0	2	0	0	0	B	-
0_2_1	7♢	3♢ 3♡	4♣ 9♢ Q♣	3♢		0	2	0	2	0	0	0	B	-
0_3_0	7♢	3♡	3♢ 4♣ 9♢ Q♣	7♢	4♣	1	2	0	2	0	0	0	A	-
0_3_1		3♡	3♢ 9♢ Q♣	3♡		1	2	0	2	0	0	0	A	-
1_0_0	6♣ 6♢ 9♡ J♣	6♡ 7♣ J♢ K♡	3♢ 3♡ 9♢ Q♣	J♣	3♢ 3♡ 9♢	2	2	1	0	0	0	-2	A	-
1_0_1	6♣ 6♢ 9♡	6♡ 7♣ J♢ K♡	Q♣	J♢		2	2	1	0	0	0	-2	A	-
1_1_0	6♣ 6♢ 9♡	6♡ 7♣ K♡	J♢ Q♣	6♣		2	2	1	0	0	0	-2	A	-
1_1_1	6♢ 9♡	6♡ 7♣ K♡	6♣ J♢ Q♣	K♡		2	2	1	0	0	0	-2	A	-
1_2_0	6♢ 9♡	6♡ 7♣	6♣ J♢ Q♣ K♡	6♢		2	2	1	0	0	0	-2	A	-
1_2_1	9♡	6♡ 7♣	6♣ 6♢ J♢ Q♣ K♡	7♣		2	2	1	0	0	0	-2	A	-
1_3_0	9♡	6♡	6♣ 6♢ 7♣ J♢ Q♣ K♡	9♡		2	2	1	0	0	0	-2	A	-
1_3_1		6♡	6♣ 6♢ 7♣ 9♡ J♢ Q♣ K♡	6♡		2	2	1	0	0	0	-2	A	-
2_0_0	5♢ 9♣ 10♡ K♣	A♢ 6♠ 8♣ 8♠	6♣ 6♢ 6♡ 7♣ 9♡ J♢ Q♣ K♡	10♡		2	2	0	0	0	0	-1	0	-
2_0_1	5♢ 9♣ K♣	A♢ 6♠ 8♣ 8♠	6♣ 6♢ 6♡ 7♣ 9♡ 10♡ J♢ Q♣ K♡	8♣		2	2	0	0	0	0	-1	0	-
2_1_0	5♢ 9♣ K♣	A♢ 6♠ 8♠	6♣ 6♢ 6♡ 7♣ 8♣ 9♡ 10♡ J♢ Q♣ K♡	K♣	K♡	3	2	0	0	0	0	-1	A	-
2_1_1	5♢ 9♣	A♢ 6♠ 8♠	6♣ 6♢ 6♡ 7♣ 8♣ 9♡ 10♡ J♢ Q♣	A♢	10♡	3	2	0	1	0	0	-1	B	-
2_2_0	5♢ 9♣	6♣ 8♠	6♣ 6♢ 6♡ 7♣ 8♣ 9♡ J♢ Q♣	5♢	6♡	3	2	0	1	0	0	-1	A	-
2_2_1	9♣	6♣ 8♠	6♣ 6♢ 7♣ 8♣ 9♡ J♢ Q♣	8♠		3	2	0	1	0	0	-1	A	-
2_3_0	9♣	6♣	6♣ 6♢ 7♣ 8♣ 8♠ 9♡ J♢ Q♣	9♣		3	2	0	1	0	0	-1	A	-
2_3_1		6♣	6♣ 6♢ 7♣ 8♣ 8♠ 9♣ 9♡ J♢ Q♣	6♣		3	2	0	1	0	0	-1	A	-
3_0_0	2♣ 3♣ J♣ Q♣	7♠ 8♢ 8♡ Q♡	6♣ 6♢ 6♠ 7♣ 8♣ 8♠ 9♣ 9♡ J♢ Q♣	2♣	9♣	5	2	2	0	0	0	-2	A	-
3_0_1	3♣ J♣ Q♣	7♠ 8♢ 8♡ Q♡	6♣ 6♢ 6♠ 7♣ 8♣ 8♠ 9♡ J♢ Q♣	8♢		5	2	2	0	0	0	-2	A	-
3_1_0	3♣ J♣ Q♣	7♠ 8♡ Q♡	6♣ 6♢ 6♠ 7♣ 8♣ 8♢ 8♠ 9♡ J♢ Q♣	J♣	6♣ 6♢ 6♠ 7♣ 8♣ 8♢ 8♠ 9♡ J♢	8	2	4	0	0	0	-2	A	-
3_1_1	3♣ Q♣	7♠ 8♡ Q♡	7♠ Q♣	7♠		8	2	4	0	0	0	-2	A	-
3_2_0	3♣ Q♣	8♡ Q♡	7♠ Q♣	3♣		8	2	4	0	0	0	-2	A	-
3_2_1	Q♣	8♡ Q♡	3♣ 7♠ Q♣	Q♡	Q♣	8	3	4	0	0	0	-2	B	-
3_3_0	Q♣	8♡	3♣ 7♠	Q♣		8	3	4	0	0	0	-2	B	-
3_3_1		8♡	3♣ 7♠ Q♣	8♡	3♣	8	4	4	0	0	0	-2	B	-
4_0_0	4♡ 4♠ 7♡ J♡	2♢ 5♡ 9♠ 10♠	7♠ Q♣	4♡	7♠	0	0	0	0	0	0	2	A	A
4_0_1	4♠ 7♡ J♡	2♢ 5♡ 9♠ 10♠	Q♣	5♡		0	0	0	0	0	0	2	A	A
4_1_0	4♠ 7♡ J♡	2♢ 9♠ 10♠	5♡ Q♣	7♡		0	0	0	0	0	0	2	A	A
4_1_1	4♠ J♡	2♢ 9♠ 10♠	5♡ 7♡ Q♣	10♠		0	0	0	0	0	0	2	A	A
4_2_0	4♠ J♡	2♢ 9♠	5♡ 7♡ 10♠ Q♣	J♡	5♡ 7♡ 10♠	0	0	1	0	0	0	2	A	A
4_2_1	4♠	2♢ 9♠	Q♣	2♢		0	0	1	0	0	0	2	A	A
4_3_0	4♠	9♠	2♢ Q♣	4♠		0	0	1	0	0	0	2	A	A
4_3_1		9♠	2♢ 4♠ Q♣	9♠	2♢	0	0	1	0	0	0	2	B	A
5_0_0	A♡ 3♠ 10♣ Q♢	2♡ 2♠ 8♣ 10♢	4♠ Q♣	A♡		0	0	0	0	0	0	3	0	A
5_0_1	3♠ 10♣ Q♢	2♡ 2♠ 8♣ 10♢	A♡ 4♠ Q♣	5♠		0	0	0	0	0	0	3	0	A
5_1_0	3♠ 10♣ Q♢	2♡ 2♠ 10♢	A♡ 4♠ 5♠ Q♣	10♣	A♡	1	0	1	0	0	0	3	A	A
5_1_1	3♠ Q♢	2♡ 2♠ 10♢	4♠ 5♠ Q♣	10♢		1	0	1	0	0	0	3	A	A
5_2_0	3♠ Q♢	2♡ 2♠	4♠ 5♠ 10♢ Q♣	Q♢	Q♣	1	0	1	0	0	0	3	A	A
5_2_1	3♠	2♡ 2♠	4♠ 5♠ 10♢	2♠	4♠ 5♠	1	0	1	0	0	0	3	B	A
5_3_0	3♠	2♡	10♢	3♠		1	0	1	0	0	0	3	B	A
5_3_1		2♡	3♠ 10♢	2♡		1	0	1	0	0	0	3	B	A
CleanUp						1	0	1	3	0	0	2	B	A

Figure 1: Columns `Acl`, `Apt`, and `Asr` represent the number of clubs, points, and surs collected or earned by Alex so far. `Bcl`, `Bpt`, and `Bsr` are defined similarly for Bob. Once, at the end of any round, `Acl` exceeds 7, both `Acl` and `Bcl` reset to zero, and the column `CL` indicates which player collected at least 7 clubs. According to Pasur rules, this player earns 7 points. Collecting more than 7 clubs yields no additional points unless the card is also a point card (i.e., *e.g.*, $A\clubsuit$ or $2\clubsuit$). The column Δ shows the cumulative point difference up to the end of the previous round. Δ updates at the end of each round to reflect the points earned in that round. Finally, in the *CleanUp* phase, the player who made the last pick (as shown in column `L`) collects all remaining cards from the pool. If any point cards are present, the corresponding point columns will be updated. If there are club cards in the CleanUp phase and neither player has yet reached 7 clubs, then the remaining clubs in the pool determine who earns the 7-club bonus. In such situations, the identity of the last player to pick becomes critical.

In the displayed game above, Alex earns the 7-club bonus. Bob gains 3 points from collecting $10\diamondsuit$, but he is trailing by 3 points from the rounds preceding the last. Additionally, Alex earns 1 point in the final round from capturing the $A\heartsuit$ card. This results in a final score with Alex leading by 8 points. See Table 2 for the Pasur scoring system.

2 PyTorch-Based Framework

In this section, we present our computational framework for implementing for simulating Pasur using PyTorch. The game consists of 6 rounds, and in each round, both players take 4 turns, giving rise to a game tree of depth 48. We begin by describing how a set of cards is represented as a PyTorch tensor. As illustrated in Figure 2, each card is mapped to a unique tensor index according to a fixed, natural order. The interpretation of the tensor values at these positions varies depending on the specific tensor in which they appear: it may indicate card ownership (e.g., who holds the card) or an action involving that card (e.g., laid or picked), depending on whether the tensor encodes game state, actions, or other game-related structures. This indexing convention serves as the foundation for constructing all tensors listed in Table 4.

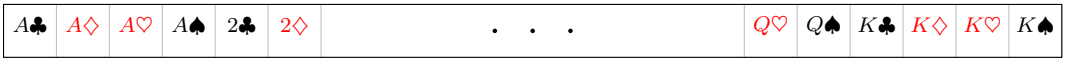


Figure 2: Mapping Cards to Indices

To manage memory efficiently during game tree generation, we maintain two boolean tensors that track card availability throughout the game. Let `m` denote the number of *in-play cards*—those currently in play, either held by a player or present in the pool. For instance, in the first round, `m = 12`, since each player is dealt four cards and four cards are placed in the pool. At the end of each round, we update the set of in-play cards: we remove any cards that have been picked across all nodes and add new cards that are about to be dealt.

This information is captured using the `t_inp` tensor. This binary tensor marks which cards are currently involved in the game—either held by players or present in the pool—and serves as the basis for constructing and updating all relevant tensors during game tree generation. It is emphasized that as a result of this, the shapes of certain tensors—particularly those that represent actions or action history, such as `t_act` and `t_gme`—may vary across rounds to reflect the current number of in-play cards. To illustrate this point, consider the initial configuration of the game shown in Table 3, with the corresponding `t_inp` tensor depicted in Figure 3.

Table 3: Initial setup for a single game instance

Alex	4♣	4♢	7♢	Q♣
Bob	3♢	3♡	5♣	K♠
Pool	A♣	A♠	9♢	K♢

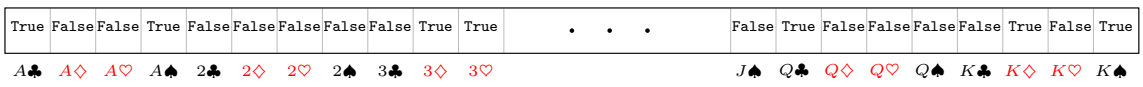


Figure 3: `t_inp` tensor at the beginning of the game for initial setup in Table 3

We next provide an overview how the game tree is generated and recorded using tensor operations. During the tree construction process, we distinguish between

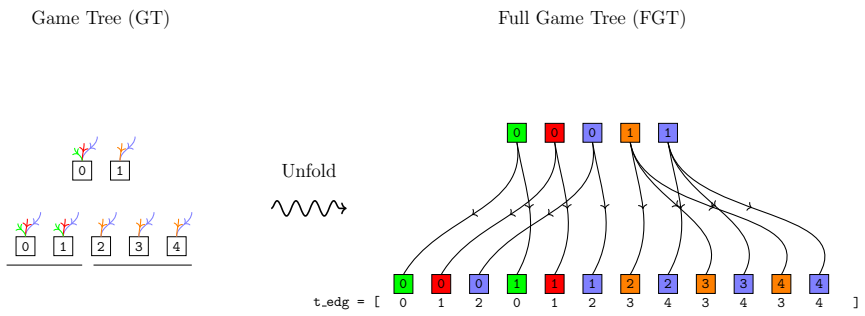


Figure 4: Unfolding Process

the *card state*—including the cards held by Alex, the cards held by Bob, and the cards in the pool—and the *score information*, which is updated independently. A given card state may correspond to multiple incoming edges, each representing a different score inherited from earlier rounds. Figure 5 illustrates this structural design. Alongside the *Game Tree* (GT) illustrated in Figure 5, we construct a *Full Game Tree* (FGT) via an *unfolding process* (Figure 4), which systematically expands the tree by combining each card state with all compatible incoming scores.

To ensure memory efficiency, the only parameters stored for FGT are the strategy values at each node and the edges linking nodes between successive layers.

Thus, each node in FGT is represented as a pair: a GT node and its corresponding incoming score. We explicitly maintain the edge structure between FGT nodes, which is crucial for updating strategies during the CFR iterations. Figure 4 provides a visual representation of the unfolding process. Further details on this mechanism and the edge-tracking procedure are discussed in the subsections that follow.

This section is organized as follows. We begin by describing the mask and padding tensors used during game tree generation in Subsection 2.1. These tensors help track the indices of cards within the current tensor as the size of these tensors may vary from round to round due to the fact that the set of in-play cards changes dynamically. Next, in Subsection 2.2, we describe the construction of the *Game Tensors*. These tensors encode the state and action history. Subsection 2.3 explains how the *Action Tensors* are built and how they are used to update the Game Tensors. Once the construction of the Action Tensors is understood, we proceed to explain the construction of the Compressed Tensor in Subsection 2.4, where all game information up to the current point is encoded in a tensor of shape `[58]`. Next, Subsection 2.5 explains the two types of score tensors used: the *Running-Score Tensor*, which accumulates scores across a single round, and the score tensors that are passed along the edges of GT, containing scores inherited from previous rounds. Then, in Subsection 2.6, we detail how *FGT Tensors* are updated within each round; it also introduces the construction of *Edge Tensors*. Following that, Subsection 2.7 describes the Between-Hand updates, including how the Score Tensor and FGT tensors are updated at the end of each round, along with the *Linkage Tensor*. Finally, Subsection 2.8 explains how *Infoset Tensors* are constructed.

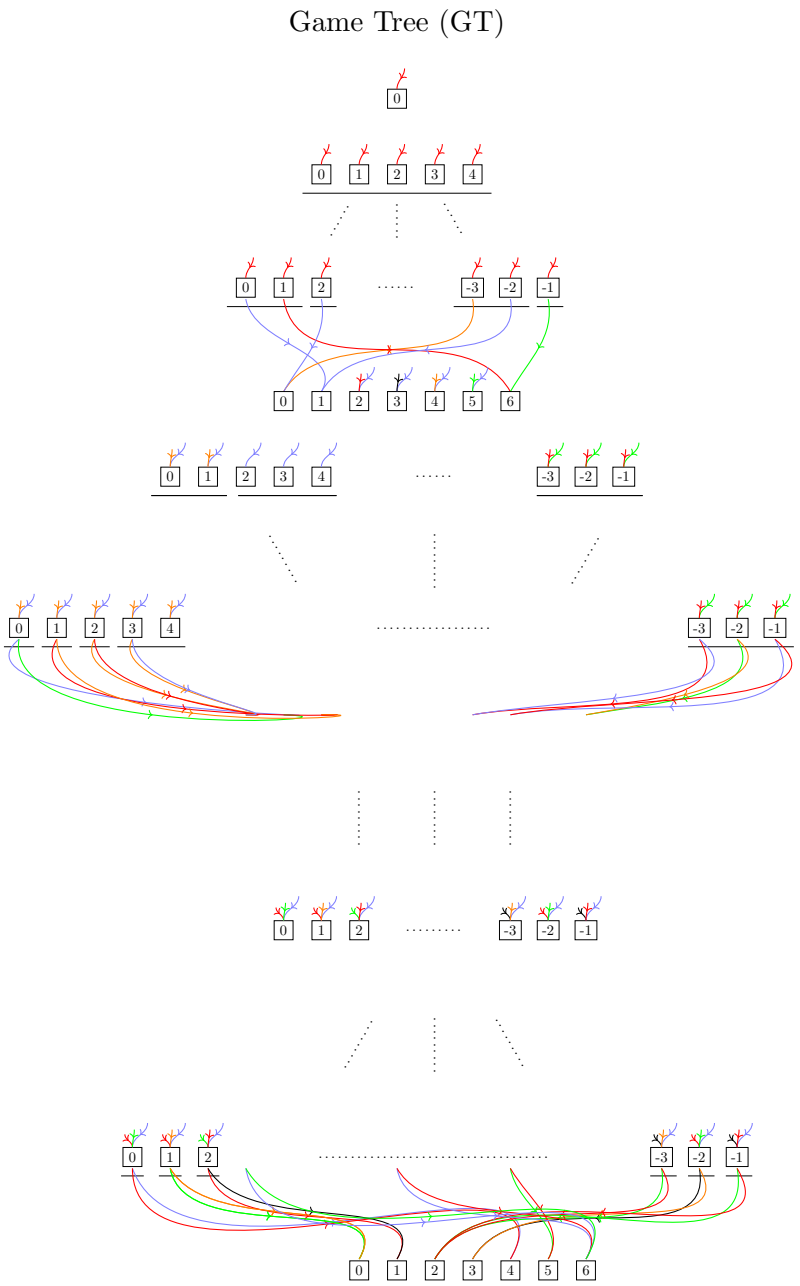


Figure 5: We distinguish between the *card state* and the *score information*, which are updated independently. A given card state may correspond to multiple incoming edges, each representing a different score inherited from earlier rounds. Underlines indicate the number of available actions for each parent node.

Before proceeding to the next section, we summarize the key components of our framework in Table 4. Padding-related tensors are described separately in Table 5.

Table 4: Summary of Tensors Used in Our Framework

Component	Tensor	Shape	Type	Update	Description
Game	`t_gme`	`[M,3,m]`	`int8`	Per turn	Encodes state and action history per node. Each slice represents a singleton node in the tree layer.
In-Play Cards	`t_inp`	`[52]`	`int8`	Per round	Boolean indicator for whether each of the 52 cards is still in play (held or in pool).
Dealt Cards	`t_dlt`	`[52]`	`int8`	Per round	Boolean indicator marking cards that have been dealt in previous rounds.
Full Game	`t_fgm`	`[Q, 2]`	`int32`	Per turn/round	Each row `[g, s]` in `t_fgm` indicates that GT's node `t_gme[g,:]` inherits the score with ID `s` from `t_scr`. Here, `Q` denotes FGT's current layer's number of nodes.
Scores	`t_scr`	`[Q, 4]`	`int8`	Per turn/round	Unique score in form (Alex Clubs, Bob Clubs, Point Difference, 7-Clubs Bonus). Scaled each turn using $\otimes$`t_brf`. See Section 2.6 for in-between hands updates.
Runnsing Scores	`t_rus`	`[M,7]`	`int8`	Per turn	Unique score in form (Alex Club, Bob Club, Last Picker, Alex Points, Alex Sur, Bob Points, Bob Sur).
Action	`t_act`	`[M',2,m]`	`int8`	Per turn	Action representation per node. `[0,:]` encodes the lay card; `[1,:]` encodes picked cards. Here, `M'` denotes the number of nodes in GT's next layer.
Branch Factor	`t_brf`	`[M]`	`int8`	Per turn	Number of valid actions available from each node; used to replicate game states before applying actions. Indicated using underlines in Figure 5.
Linkage	`t_lnk`	`[Q]`	`int32`	Per round	Connects FGT nodes between consecutive hands to identify how scores and states map across hands.
Edge	`t_edg`	`[Q']`	`int32`	Per turn	Records edges between FGT nodes to track the structure of the overall graph. Here, `Q'` denotes the number of nodes in the FGT's next layer.
Strategy	`t_sgm`	`[Q]`	`float32`	Per turn	Stores strategy values (e.g., probabilities) associated with each FGT's node.
Compressed Game	`t_cmp`	`[M, m]`	`int8`	Per turn	Stores the entire game information up to the current turn in a compressed format.
Infoset	`t_inf`	`[Q,58]`	`int8`	Per turn	Encodes information available to player whose turn it is. `t_inf` hides information that is not observable by the acting player. Includes metadata such as round index, turn counter, and cumulative score up to the previous round.

2.1 Mask and Padding Tensors

We now describe how masks and padding tensors are constructed from the in-play card tensor `t_inp`, which has shape `[M, 52]` and indicates which cards are active during a round.

Table 5: Mask and Padding Variables

Variable	Description
`t_inn, t_inq, t_ink, t_inj, t_inc, t_ins`	Masks for Numerical, Queen, King, Jack, Club, and Point cards resp.
`i_pdn, i_pdj, i_pdq, i_pdk`	Padding sizes for Numerical, Jack, Queen, and King actions resp.
`t_pdq, t_pdk`	Permutations to restore Queen and King actions to original order, resp.

Table 6: Index Sets for Card Categories and Scoring

Index Set	Description
`l_n = range(40)`	Numerical cards (2 to 10 of all suits including Aces).
`l_j = [40, 41, 42, 43]`	Jacks.
`l_q = [44, 45, 46, 47]`	Queens.
`l_k = [48, 49, 50, 51]`	Kings.
`l_c = range(0, 52, 4)`	Clubs (every 4th card).
`l_p = [0,1,2,3,4,37,40,41,42,43]`	Scoring cards: all Aces, 2♣, 10♢, all Jacks.
`l_s = [1,1,1,1,2,3,1,1,1,1]`	Scores corresponding to entries in `l_p`.

To compute masks in Table 5, we use the index sets from Table 6, which are computed as follows:

```
t_inn, t_inq, t_ink, t_inj, t_inc, t_ins =

            (f(index_set) for index_set in [l_n,l_q,l_k,l_j,l_c,l_s])
```

Here the helper function `f` is defined below:

```
f = lambda lm: tensor([id in lm for id,card in enumerate(t_inp) if card])
```

We compute the counts of each category:

```
i_n, i_j, i_q, i_k = (t.sum() for t in [t_inn, t_inj, t_inq, t_ink])
```

We also define intermediate sums:

```
i_kq = i_k + i_q
i_jqk = i_j + i_q + i_k
i_njk = i_n + i_j + i_k
i_njq = i_n + i_j + i_q
i_njqk = i_n + i_j + i_q + i_k
```

Using these, we define:

```
i_pdn, i_pdj, i_pdk, i_pdq = i_jqk, i_kq, i_njq, i_njk
```

And the corresponding padding permutations:

```
t_pdk = hstack((arange(i_k, i_njqk), arange(i_k)))
t_pdq = hstack((arange(i_q, i_njq), arange(i_q), arange(i_njq, i_njqk)))
```

2.2 Game Tensor

Each row in the game tree (Figure 5) is represented using an `[M,3,m]` GT tensor `t_gme`, where `M` corresponds to the number of nodes in the current GT's layer. The associated `[3,m]` tensor is constructed for each node: Each slice `t_gme[g,:,:]` encodes a single game state. The first row of the tensor, `t_gme[g,0,:]`, represents the current card holdings and pool status. To encode the game state numerically, we assign integer identifiers to the entities involved, as shown in Table 7.

Table 7: State Encoding in `t_gme[g,0,:]`

Element	Encoding
Alex	1
Bob	2
Pool	3

The remaining two rows of `t_gme`, namely `t_gme[g,1,:]` and `t_gme[g,2,:]`, record the action histories of Alex and Bob, resp. We describe the construction of `t_gme[g,1,:]` in detail below; the construction of `t_gme[g,2,:]` is analogous.

Each round consists of four turns, meaning that Alex plays a card four times per round. During each turn `i_trn = i`, for `i = 0, 1, 2, 3`, he may or may not collect cards from the pool.

Suppose Alex plays the card $A\heartsuit$ on his first turn and collects $10\heartsuit$. We update `t_gme[g, 1,:]` by recording the value `1` at the position corresponding to $A\heartsuit$, and adding the value `10` at the position corresponding to $10\heartsuit$. If Alex collects multiple cards from the pool on his first turn, the value `10` is added to all corresponding positions in `t_gme[g, 1,:]` for each collected card. For subsequent turns, similar updates are performed, except we use the value pairs `(2,20)`, `(3,30)`, and `(4,40)` instead of `(1,10)` for the second, third, and fourth turns, respectively.

Importantly, `t_gme[g,0,:]` is updated after each move to reflect the current state of the game. For cards that are played but not collected, the corresponding position in `t_gme[g,0,:]` is set to `3`. If a played card is used to collect one or more cards from the pool, then the corresponding positions are updated as follows: hand cards that were played or picked change from `1` to `0`, and pool cards that were collected change from `3` to `0`. See CodeSnippet 1.

2.3 Actions

At each turn, we first compute two key tensors: the *Branch Factor Tensor* `t_brf` and the *Action Tensor* `t_act`: `t_brf` is a tensor of length `M`, where `M` is the number of nodes in the current round. Each entry `t_brf[g]` records the number of valid actions available from node `g`. Correspondingly, `t_act` is a binary tensor of shape `[M',2,m]`, where `m` is the number of in-play cards. Here, `M'=t_brf.sum()` is the total number of resulting nodes in the next round, created by enumerating all valid actions from the current nodes. Each slice `t_act[j, :, :]` encodes a single action, consisting of two rows as described in Table 8.

Table 8: Structure of the Action Tensor `t_act`

Tensor Slice	Description
`t_act[j, 0, :]`	Encodes the *lay card*. Contains exactly one `1` at the index of the played card; all other entries are `0`.
`t_act[j, 1, :]`	Encodes the *picked cards* from the pool. May contain zero or more `1`s depending on the number of picked cards.

To update the game state tensor `t_gme` for the next round, we first replicate each current node `g` according to its corresponding `t_brf[g]` count, effectively expanding `t_gme` to match the total number of action slices `M'`. This is done using the `repeat_interleave` operator, denoted by the Kronecker product symbol: $\otimes$.

$$\texttt{t\_gme} \leftarrow \texttt{t\_gme} \otimes \texttt{t\_brf}$$

This expansion ensures that each valid action is paired with its own copy of the corresponding game state. Then, each slice of `t_act` is applied to the corresponding replicated game state to produce the updated states. Finally, we apply the encoded actions by updating `t_gme` using `t_act`. See CodeSnippet 1.

CodeSnippet 1 ApplyActions

```
Input t_gme, t_act, i_ply, i_trn
# t_act.shape[0] = t_gme.shape[0] after expansion t_gme ← t_gme ⊗ t_brf
t_mpk ← any(t_act[:,1,:], dim=1)
# Whether a pick action occurs
t_gme[t_mpk,0,:]+=(2-i_ply)*t_act[t_mpk,0,:]
# Add lay card to pool (only if no pick)
t_gme[t_mpk,0,:]-=(1+i_ply)*t_act[t_mpk,0,:]
# Remove lay card from player's hand
t_gme[t_mpk,0,:]-=3*t_act[t_mpk,1,:]
# Remove picked cards from the pool
t_gme[:,i_ply+1,:]+=(i_trn+1)*t_act[:,0,:]+10*(i_trn+1)*t_act[:,1,:]
# Update player record: lay (weighted by i_trn+1) + pick (weighted by 10 × i_trn+1)
```

We now present how the Action Tensor `t_act` is constructed using the GT tensor `t_gme`. There are four types of actions: Numerical, Jack, King, and Queen. The construction process is explained below. To derive actions from a given `t_gme` tensor, we first construct an `int8` tensor of shape `[2, 52]` denoted as `t_2x52`, along

with various views such as t_2x40, t_2x44, and two t_2x4 tensors corresponding to different action types. The construction of t_2x52 is explained in CodeSnippet 2. Once t_2x52 is obtained, we define the inputs to each action operator as:

CodeSnippet 2 t_2x52

```
Input t_gme
t_2x52 ← zeros((t_gme.shape[0],2,t_gme.shape[2]))
t_2x52[:,0,:][t_gme[:,0,:]==i_ply+1]←1
t_2x52[:,1,:][t_gme[:,0,:]==3]←1
```

```
t_2x52[:,:,t_msk] for t_msk in [t_inn, t_inn+t_inj, t_ink, t_inq]
```

These serve as inputs to the Numerical, Jack, King, and Queen action routines, respectively. For memory efficiency, we first apply `unique` to these tensors and pass the result to `n_act`, `j_act`, `k_act`, and `q_act` to compute the action and branch factor tensors. We next pass each resulting action and branch tensor pair `t_act, t_brft` to CodeSnippet 3 to map them back to the full tensor.

CodeSnippet 3 Mapping Back Actions from Unique Tensor to Full Tensor

```
Input t, f_act # t: input tensor, f_act: function to find actions (e.g., numerical, jack, etc.)
tu, t_inx ← unique(t, sorted=Fasle, return_inverse=True)
tu_act, tu_brf ← f_act(tu)
t_brf ← tu_brf[t_inx]
t_act ← tu_act[inverseunique(tu_brf,t_inx)]
return t_brf, t_act

function INVERSEUNIQUE(t_cnt,t_inx) # Ex: t_cnt = [2,3], t_inx = [1,0,0]
    t_cms ← zeros(t_cnt.shape[0]+1, dtype=int32) # = [0,2,5]
    t_cms[1:]  ← t_cnt.cumsum(0)
    t_bgn ← t_cms[t_inx] # = [2,0,0]
    t_len ← t_cnt[t_inx] # = [3,2,2]
    t_lns ← t_len.cumsum(0) # = [3,5,7]
    t_ofs ← arange(t_lns[-1])-(t_lns-t_len)⊗t_len
# arange(t_lns[-1])=[0,1,2,3,4,5,6], t_lns-t_len=[0,3,5]
# (t_lns - t_len)⊗t_len = [0,0,0,3,3,5,5], t_ofs = [0,1,2,0,1,0,1]
    return t_bgn⊗t_len + t_ofs # = [2,2,2,0,0,0,0]+[0,1,2,0,1,0,1] = [2,3,4,0,1,0,1]
end function
```

We denote the resulting action tensors as `t_pck, t_lay, t_kng, t_jck, t_qun` and the corresponding branch factor tensors as `c_pck, c_lay, c_kng, c_jck, c_qun`. It is emphasized that `n_act` produces two pairs of action and branch factor tensors: Pick and Lay pairs. Pick actions correspond to cases where at least one card (along with the card in hand) is picked from the pool, while Lay actions correspond to cases where no card is picked and only one card is laid down and added to the pool.

We pad these tensors to match the original shape of tensor `t_2x52` as follows:

```
t_pck = pad(t_pck, (0, i_pdn))
t_lay = pad(t_lay, (0, i_pdn))
t_kng = pad(t_kng, (0, i_pdk))[:,:,t_pdk]
t_qun = pad(t_qun, (0, i_pdq))[:,:,t_pdq]
t_jck = pad(t_jck, (0, i_pdj))
```

The branch factor is also easily calculated as below

```
t_brf = sum(stack([c_pck, c_lay, c_kng, c_qun, c_jck]), dim=0)
```

We perform one final step to construct the action tensor `t_act`. This step ensures that all actions corresponding to each row of the game tensor are grouped together. To achieve this, we repeat each row index of the game tensor according to its corresponding branch factor, and then use the sorted indices to reorder the concatenated action tensors.

```
_, t_inx = sort(cat([arange(M)⊗t for t in [c_pck,c_lay,c_kng,c_qun,c_jck]]))
```

Once `t_inx` is obtained, the final action tensor is constructed by concatenating all action components and reordering them using `t_inx`:

```
t_act = cat([t_pck, t_lay, t_kng, t_qun, t_jck])[t_inx]
```

2.3.1 Numerical Actions

In this section, we explain how Numerical Actions are constructed. Recall that the input tensor to `n_act` is given by `t_2x40 = t_2x52[:,:,t_inn]`.

We begin by identifying all possible combinations of numerical cards such that the sum of each subset equals `11`. This is a classical instance of the *Subset Sum Problem* (SSP), which we solve using dynamic programming. The solutions are stored in a tensor `t_40x2764` of shape `[40, 2764]`, where each row corresponds to one of the `40` numerical cards, and each column represents a valid subset whose sum equals `11`. We also define an auxiliary tensor `t_tpl` as in Table 9.

Table 9: Tuple Tensor `t_tpl` of shape `[2, 2764]`

Row	Description
`t_tpl[0,:]`	All entries are 1, representing a fixed lay card for each action.
`t_tpl[1,:]`	Stores the number of cards from the pool that are involved in each action.

Since we mask only for in-play cards, we need to exclude those that are not in-play. To this end, we apply the mask `t_inp[:40]` to `t_40x2764` by computing `t_40x2764[:, t_inp[:40]]`, and pass the resulting tensor to `n_act`. We still denote `t_40x2764` as the masked tensor for convenience. Moreover, denote by `t_2764x40` the transpose of `t_40x2764`. We now describe how tensors `t_pck` and `t_lay` are constructed. The process begins with `t_pck`, from which `t_lay` is subsequently derived. Consider the matrix product

```
matmul(t_2x40, t_40x2764)
```

which results in a tensor of shape `[M,2,2764]`, where `M` is the number of rows in `t_2x40`. This tensor evaluates how each input pair of player hand and pool (i.e., each row of `t_2x40`) overlaps with the precomputed subset-sum solutions in `t_40x2764`. Notice masking out `t_40x2764` does not affect the outcome of the matrix multiplication, as masked entries are `0` and thus do not contribute to the result.

A pair of player hand and pool (i.e., row `i` of `t_2x40`) is said to have the action `j` available if and only if the count of matching cards exactly matches the template `t_tpl[:, j]`. The following code yields all indices `[i, j]` such that action `j` is valid for row `i` of `t_2x40`.

```
t_inx = nonzero(((t_tpl - matmul(t_2x40, t_40x2764)) == 0).all(dim=1))
```

We use the index tensor `t_inx` to construct the pick action tensor `t_pck`. Computing the corresponding `t_brf` is straightforward from `t_inx`; it is computed as follows:

```
c_pck = t_inx[:,0].bincount(minlength=M)
```

Now that we have the number of actions availble for each row and also the indices of the corresponding actions in `t_inx`, we could obtain `t_pck` as follows: We repeat each row of `t_2x40` based on the number of available actions for that row. Simultaneously, we repeat each available action twice along `dim=1`. Then, we apply the `logical_and` operator to identify matching cards. The following performs this computation:

```
t_pck = logical_and(t_2x40⊗c_pck, t_2764x40[t_inx[:,1]].unsqueeze(1)⊗1 2)
```

Here each row of `t_2764x40[t_inx[:,1]]` corresponds to a valid available action.

We are now ready to construct the Lay Action tensor `t_lay`. The main idea is to identify which cards in the player's hand have not been used in any Pick Action. To achieve this, we first build a tensor `t_hnd` that records how many times each card in the hand has been selected for a Pick Action. This is accomplished using the `scatter_add_` operation. The index tensor used for this purpose has the same shape as `t_pck[:,0,:]`, and rows corresponding to the same row in `t_2x40` are grouped together through this index tensor. Once `t_hnd` is constructed, we subtract it from the player's original hand tensor `t_2x40[:,0,:]` and apply `relu` to isolate the remaining cards that were not picked. The result is a binary tensor indicating candidate cards for Lay Actions. The remainder of the construction is straightforward and presented in CodeSnippet 4.

CodeSnippet 4 Constructing Lay Action Tensor from Pick Tensor

```
m_pck ← c_pck > 0
t_cnt ← c_pck[m_pck]
i_cnt ← t_cnt.shape[0]

# Construct hand tensor by summing picked cards
t_hnd ← zeros((i_cnt, m))
t_inx ← (arange(i_cnt)⊗t_cnt).view(-1,1).expand(-1, m)
t_src ← t_pck[:,0,:]
t_hnd.scatter_add_(0, t_inx, t_src)

# Remove picked cards from hand to construct lay
t_cln ← t_2x40[:,0,:].clone()
t_cln[m_pck,:]  ← relu(t_2x40[m_pck,0,:]-t_hnd)

# Find positions to lay the remaining card
t_inx ← nonzero(t_cln == 1)
c_lay ← bincount(t_inx[:,0], minlength=M)
t_lay ← zeros((t_inx.shape[0], 2, m))
t_lay[arange(t_lay.shape[0]), 0, t_inx[:,1]] ← 1
```

2.3.2 Jack Actions

As discussed earlier, `t_2x44 = t_2x52[:,:,t_inn+t_inj]` serves as the input tensor to the `j_act` function. The first step is to identify all rows in `t_2x44` that contain Jack cards. This is accomplished by:

```
t_inx ← nonzero(t_2x44[:, 0, i_pdn:])
```

Recall from Table 5 that `i_pdn` denotes the number of numerical cards among the current in-play cards. The corresponding branch factor tensor is obtained via:

```
c_jck ← t_inx[:, 0].bincount(minlength=M)
```

To construct the Jack Action tensor `t_jck`, we first let `i_cnt = t_inx.shape[0]`. Since each Jack card corresponds to exactly one action, the tensor `t_jck` is initialized with shape `[i_cnt, 2, m]`. This tensor is then populated as follows:

```
t_rng ← arange(i_cnt)
t_jck[t_rng, 0, i_pdn + t_inx[:,1]] ← 1
t_jck[t_rng, 1, :]  ← t_2x44[:,1,:]⊗c_jck
```

2.3.3 King Actions

In this section, we describe how King Action is constructed; the construction of Queen Action follows similarly. Recall that the input to `k_act` is given by `t_2x4 = t_2x52[:, :, t_ink]`. Since we apply `unique` before passing the tensor to `k_act` (as explained earlier), the input tensor `t_2x4` has a limited number of unique possibilities. Specifically, the number of rows in `t_2x4` cannot exceed `3**4 = 81`. Each

row of t_2x4 is first converted into a string of length 4. To generate this string representation, we first apply the transformation

$$\texttt{t\_2x4[:, 1, :][t\_2x4[:, 1, :] == 1] = 2},$$

and then compute t_2x4.sum(dim=1), which is subsequently converted into a string key. For example, '0201' indicates that the player holds a $K\spadesuit$, while the pool contains a $K\diamondsuit$. This key is then used to index into the dictionary to retrieve the corresponding action tensor. Finally, we use this key to look up the corresponding action tensor in our dictionary. CodeSnippet 5 summarizes this discussion. Details regarding the lookup table are omitted here; please refer to the GitHub code repository for full implementation details.

CodeSnippet 5 Constructing King Action Tensor

```
t_2x4[:, 1, :][t_2x4[:, 1, :] == 1] ← 2
M0 ← t_2x4.shape[0]
Build lookup_tbl using t_2x4.sum(dim=1).
c_k ← tensor([lookup_tbl[i].shape[0] for i in range(M0)])
t_k ← cat([lookup_tbl[i] for i in range(M0)], dim=0)
```

2.4 Compressed Game Tensor

In this subsection, we represent a game tensor t_gme of shape [M, 3, m] using a compressed form t_cmp of shape [M, m]. This compressed tensor is later used to build the Infoset representation in Section 2.8. During GT construction, all resulting t_cmp tensors are stored, and it becomes straightforward to mask information not observable by the acting player through simple masking operations. Next, we describe how t_cmp is constructed and explain why this construction uniquely encodes a GT layer, given the current In-Play tensor t_inp. See CodeSnippet 6.

CodeSnippet 6 Compressed Game Tensor

```
Input: t_gme
t_cmp ← t_gme[:,1,:] - t_gme[:,2,:]
t_lpm ← logical_and(t_gme[:,0,:] == 0, (t_cmp != 0) & (t_cmp.abs()
< 5))
t_cmp[t_lpm] ← 110 + t_cmp[t_lpm]
t_cmp[logical_and(t_gme[:,0,:] == 3, t_cmp == 0)] ← 110
t_cmp[logical_and(t_gme[:,0,:] == 1, t_cmp == 0)] ← 100
t_cmp[logical_and(t_gme[:,0,:] == 2, t_cmp == 0)] ← 105
```

In Table 11, we summarize all possible values that can appear in the compressed tensor t_cmp. Each card can have up to two *legs* in a round. The first leg corresponds to when the card is *laid* into the pool, and the second leg corresponds to when it is *picked* from the pool. A card may have only one leg in the current layer of the game tensor—this occurs when a player lays the card into the pool, but it has not yet been picked by any player. Alternatively, a card may have no legs in the current round; this happens when the card was already in the pool at the start of the round

or is still held by one of the players. By design of the game tensor `t_gme`, we can characterize these situations precisely:

Table 10: Leg Status Conditions for Cards

No Legs	One Leg	Both Legs in Same Turn
`t_cmp==0`	`t_gme[:,0,:]==3 & t_cmp!=0`	`t_gme[:,0,:]==0 & (t_cmp != 0) & (t_cmp.abs() < 5)`

Table 11: Compressed Game Tensor Values in `t_cmp`

Value	First Leg	Second Leg	Description
100	—	—	Held by Alex
105	—	—	Held by Bob
110	—	—	Was already in the pool
111,112,113,114	Alex laid at `i_trn=0:3`	Alex picked at same turn	Both legs in same turn by Alex
109,108,107,106	Bob laid at `i_trn=0:3`	Bob picked at same turn	Both legs in same turn by Bob
1,2,3,4	Alex laid at `i_trn=0:3`	—	Not yet picked
-1,-2,-3,-4	Bob laid at `i_trn=0:3`	—	Not yet picked
-9,-19,-29,-39	Alex laid at `i_trn=0`	Bob picked at `i_trn=0:3`	
-18,-28,-38	Alex laid at `i_trn=1`	Bob picked at `i_trn=1:3`	
-27,-37	Alex laid at `i_trn=2`	Bob picked at `i_trn=2:3`	
21,31,41	Alex laid at `i_trn=0`	Alex picked at `i_trn=1:3`	
32,42	Alex laid at `i_trn=1`	Alex picked at `i_trn=2:3`	
43	Alex laid at `i_trn=2`	Alex picked at `i_trn=3`	
19,29,39	Bob laid at `i_trn=0`	Alex picked at `i_trn=1:3`	
28,38	Bob laid at `i_trn=1`	Alex picked at `i_trn=2:3`	
37	Bob laid at `i_trn=2`	Alex picked at `i_trn=3`	
-21,-31,-41	Bob laid at `i_trn=0`	Bob picked at `i_trn=1:3`	
-32,-42	Bob laid at `i_trn=1`	Bob picked at `i_trn=2:3`	
-43	Bob laid at `i_trn=2`	Bob picked at `i_trn=3`	

2.5 Score Tensors

There are two types of score tensors: those that store the score accumulated within a single round, called the *running-score tensor* `t_rus`, and those that are passed along edges in GT, referred to as the *score tensor* `t_scr`. The tensor `t_rus` is initialized to zero at the beginning of each round and is updated after every turn. At the end of the round, the final value of `t_rus` is used to update `t_scr`. Tables 12 and 13 outline the meaning of each index in the score tensors `t_scr` and `t_rus`, respectively. Note that the running-score tensor `t_rus` includes additional components relevant to a single round.

Table 12: Column definitions for the score tensor

Alex Club	Bob Club	Point Difference	7-Clubs Bonus
0	1	2	3

Table 13: Column definitions for the running-score tensor

Alex Club	Bob Club	Last Picker	Alex Points	Alex Sur	Bob Points	Bob Sur
0	1	2	3	4	5	6

We need to clarify an important point about the score tensor `t_scr`. After each round, once the number of Clubs collected by either Alex or Bob reaches 7, we update the last column—the 7-Clubs Bonus—to reflect that one of the players has achieved the bonus. Specifically, the value becomes 1 if Alex reaches 7 Clubs, or 2 if Bob does. At the same time, the counts in the first two columns, which track Clubs for Alex and Bob respectively, are reset to 0.

For example, the score vector `[3, 8, 2, 0]` is not a valid state; it will be converted to `[0, 0, 2, 2]`, indicating that Bob has earned the 7-Clubs bonus. At this point, accumulating additional Clubs is irrelevant to the 7-Clubs bonus and is therefore not tracked. The primary advantage of this design choice is that it reduces the number of possible edges (and hence the size of the FGT), resulting in significant savings in both computation and memory.

We omit the procedure for updating the running-score tensor and refer the reader to the Github repository.

2.6 In-Hand Updates

In this section, we explain how in-hand updates are performed. A key operation required throughout is a generalized version of `torch.repeat_interleave`, which we refer to as **RepeatBlocks**. Unlike `torch.repeat_interleave`, which repeats individual elements, RepeatBlocks operates on entire contiguous blocks of elements and repeats them as a whole. This operation is detailed in Appendix 5.

We now explain how FGT is updated within each round. Two tensors are updated during this process: the FGT tensor `t_fgm` and the Edge tensor `t_edg`, which captures the connections between nodes across successive layers of FGT.

The FGT tensor `t_fgm` is used to track which scores accumulated from previous rounds (stored in `t_scr`) are linked to each node of the GT tensor `t_gme`. A row entry `[g, s]` in `t_fgm` indicates that node `t_gme[g]` inherits the score `t_scr[s]` from a previous round. See Figure 6 for an example consistent with the setting illustrated on the left-hand side of Figure 4.

Figure 4 illustrates how `t_edg` is constructed. On the left-hand side, we show two parent nodes—one with 2 child nodes and 3 inherited scores, and the other with 3 child nodes and 2 inherited scores. The right-hand side of the figure illustrates the corresponding structure within FGT. Here, colors denote inherited scores from the previous round, and numbers represent the row indices from the current GT tensor `t_gme`. An edge is drawn when the score colors match and the nodes are connected in the folded GT. In this example, the tensor

```
t_edg = [0,1,2,0,1,2,3,4,3,4,3,4]
```

records the parent indices in FGT. The counts of inherited scores and available

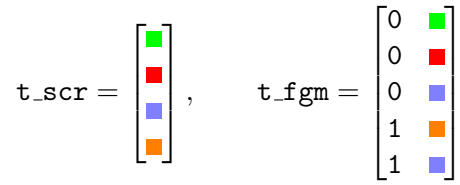


Figure 6: An example illustrating how the FGT tensor `t_fgm` associates nodes in the GT with entries in the Score Tensor `t_scr`. Colors are used to indicate rows of `t_scr`, consistent with Figure 5. Each row of `t_scr` is represented by a tensor of size 4, as described in Table 12.

actions per parent are given by `c_scr = [3,2]` and `t_brf = [2,3]`, respectively. This structure can be generated compactly by the RepeatBlocks operator as follows:

$$\texttt{t\_edg} \leftarrow \texttt{arange(t\_fgm.shape[0])} \otimes_{\texttt{c\_scr}} \texttt{t\_brf}.$$

We will next explain how the FGT tensor `t_fgm` is updated. Figure 7 shows the `t_fgm` tensors before and after the update, corresponding to the setting of Figure 4. The update to `t_fgm` is performed in two phases: first, we update `t_fgm[:,1]`; then, we update `t_gme` and `c_scr` before proceeding to update `t_fgm[:,0]`. We have

$$\texttt{t\_fgm[:,1]} \leftarrow [\blacksquare\ \blacksquare\ \blacksquare\ \blacksquare\ \blacksquare] \otimes_{\texttt{[3,2]}} \texttt{[2,3]}$$

where the first tensor on the right-hand side is `t_fgm[:,1]`. The first three colors are repeated twice on the right-hand side of Figure 7 because `t_brf[0] = 2`, and the last two are repeated three times because `t_brf[1] = 3`. Therefore

$$\texttt{t\_fgm[:,1]} \leftarrow \texttt{t\_fgm[:,1]} \otimes_{\texttt{c\_scr}} \texttt{t\_brf}$$

Furthermore,

$$\texttt{t\_fgm[:,0]} \leftarrow \texttt{[0,1,2,3,4]} \otimes \texttt{[3,3,2,2,2]}$$

In words, each node in GT's second layer in the left-hand side of Figure 4 is repeated as many times as the number of scores it inherited from the previous round. Namely,

$$\texttt{t\_fgm[:,0]} \leftarrow \texttt{arange(t\_gme.shape[0])} \otimes \texttt{c\_scr}$$

Both `t_gme` and `c_scr` are updated before the equation above.

2.7 Between-Hand Updates

In this section, we introduce the processing framework used at the beginning of each hand. Figure 8 illustrates this process. By the end of each hand, we obtain a set of GT nodes, each associated with one or more inherited scores from previous rounds. Our goal is to construct the root-level nodes of GT's next round, along with their inherited scores. To begin, we identify the unique game states and running-score

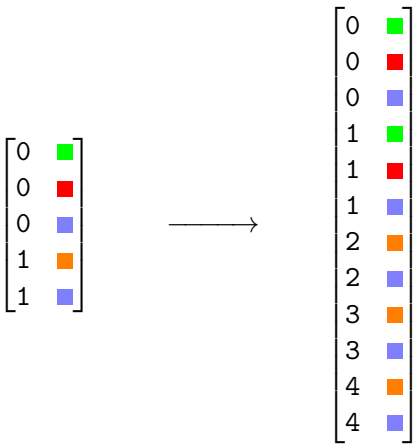


Figure 7: An illustration of how `t_fgm` is updated.

tensors at the terminal level of the current round. This is achieved via the following operation:

$$\texttt{t\_gme}, \texttt{t\_gnk} = \texttt{unique(t\_gme, dim=0, sorted=False, return\_inverse=True)}$$
$$\texttt{t\_rus}, \texttt{t\_rnk} = \texttt{unique(t\_rus, dim=0, sorted=False, return\_inverse=True)}$$

We first update the second column of FGT tensor `t_fgm`, namely `t_fgm[:,1]`, which encodes the inherited scores. To propagate scores correctly, we take the score components from the previous round, stored in `t_fgm[:,1]`, and add the scores earned during the current hand, recorded in `t_rus`. To this end, we represent each player's total score—comprising both the inherited and running components—as follows:

$$\texttt{t\_prs} \leftarrow \texttt{cat([t\_fgm[:,1], t\_rnk[t\_fgm[:,0]]], dim=1)}$$

Concatenate these two components to identify the unique score combinations:

$$\texttt{t\_prs, t\_pid} \leftarrow \texttt{unique(t\_prs, dim=0, sorted = False, return\_inverse=True)}$$

Each unique pair in `t_prs` represents a total score passed to the next round, and is computed by summing the individual components:

$$\texttt{t\_scr} \leftarrow \texttt{t\_scr[t\_prs[:,0]]+t\_rus[t\_prs[:,1]]}$$

We next eliminate duplicate score values using another call to `unique`:

$$\texttt{t\_scr}, \texttt{t\_fid} \leftarrow \texttt{unique(t\_scr, dim=0, soretd=False, return\_inverse=True)}$$

This gives us enough ingredients to update `t_fgm[:,1]` as follows:

$$\texttt{t\_fgm[:,1]} \leftarrow \texttt{t\_fid[t\_pid]}$$

Now we update the first column of the FGT `t_fgm[:,0]` as follows:

$$\texttt{t\_fgm[:,0]} \leftarrow \texttt{t\_gnk[t\_fgm[:,0]]}$$

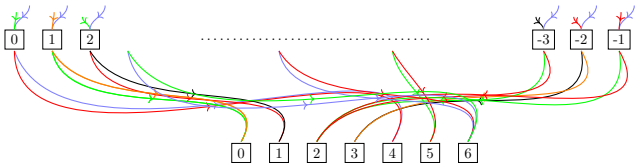


Figure 8: An illustration of between-hand processing. Note that the number of incoming edges to each terminal node in the last round is equal to the number of edges going from that node to the root-level node in the next round. However, the color may change, as the corresponding running-score tensor is added to each score inherited from previous rounds.

Note that this is an intermediate step: the final version of `t_fgm` will be updated later in the process. Next, we determine how scores are passed from one hand to the next. That is, we construct the new `t_fgm[:,1]` tensor for the upcoming round. Finally, we find unique rows in `t_fgm` as follows:

```
t_fgm, t_lnk ← unique(t_fgm,dim=0, sorted=False, return_inverse=True)
```

CodeSnippet 7 summarizes the steps discussed above.

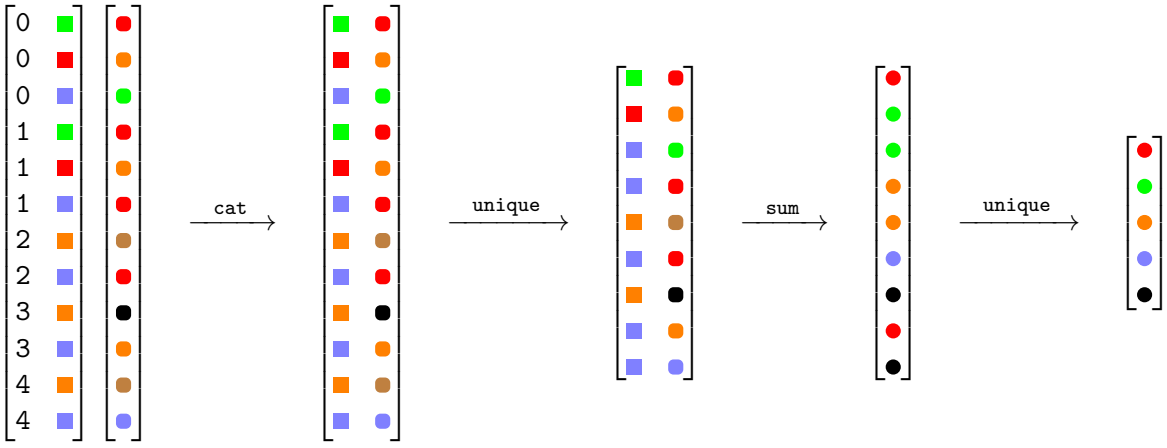


Figure 9: An illustration of between-hand updates for `t_fgm`.

2.8 Infoset Tensors

In this section, we represent FGT's nodes using the `int8` Infoset Tensor `t_inf`, where each row has shape `[58]`. This tensor encodes all information available at that point in the game tree and can be easily adapted to mask any information not observable by the acting player. This design serves two primary purposes: first, to ensure memory efficiency; and second, to provide a compact representation of both the game state and associated action information. These representations are later used to train a tree-based model to approximate the strategy profile, which is

CodeSnippet 7 BetweenHands

```
Input t_gme,t_fgm,t_scr,t_dck
t_gme, t_gnk ← unique(t_gme,dim=0)
t_rus, t_rnk ← unique(t_rus,dim=0)

t_prs ← cat([t_fgm[:,1],t_rnk[t_fgm[:,0]]], dim=1)
t_prs,t_pid ← unique(t_prs,dim=0)

t_scr ← t_scr[t_prs[:,0]]+t_rus[t_prs[:,1]]
t_scr, t_fid ← unique(t_scr, dim=0)

t_fgm[:,1] ← t_fid[t_pid]
t_fgm[:,0] ← t_gnk[t_fgm[:,0]]

t_fgm, t_lnk ← unique(t_fgm,dim=0)
# Note:  All the unique operations here consider sorted=False,return_inverse=True
```

subsequently used in self-play simulations, as described in Section 3. It is important to emphasize that these infoset tensors are used only after the Nash equilibrium has been learned using the CFR algorithm. Tensor `t_inf` consists of three distinct parts, summarized in Table 14 and CodeSnippet 8 explains its construction process.

Table 14: Index structure of the `t_inf` tensor

Index Range	Section	Description
0:52	Cards	Status/history of each card
52:55	Score context	Inherited scores such as club points and point differential
55:58	Metadata	Round index, turn counter, and current player indicator

CodeSnippet 8 Infoset Tensor

```
t_inf ← zeros((Q, 58)) # Q=t_fgm.shape[0]

t_inf[:, t_inp]←t_cmp[t_fgm[:,0]]
t_inf[:,52:55]←t_scr[t_fgm[:,1]]

t_inf[logical_and(t_dlt==1,t_inf==0)]←-127# for any dealt card represented
in t_dlt & not present in t_gme, -127 is assigned i.e., card is collected in past rounds.
t_inf[:,55:58]←tensor([i_hnd,i_trn,i_ply])
```

2.9 External Sampling

In this section, we describe the procedure for external sampling used during the unfolding process. Although the present work does not rely on or develop specific sampling techniques, we include this explanation for the sake of completeness and future reference. The external sampling method outlined here will play a key role in subsequent extensions of this research, where sampling-based approaches will be more actively explored and integrated into the modeling framework.

In external sampling, a single action is selected for the opponent based on their

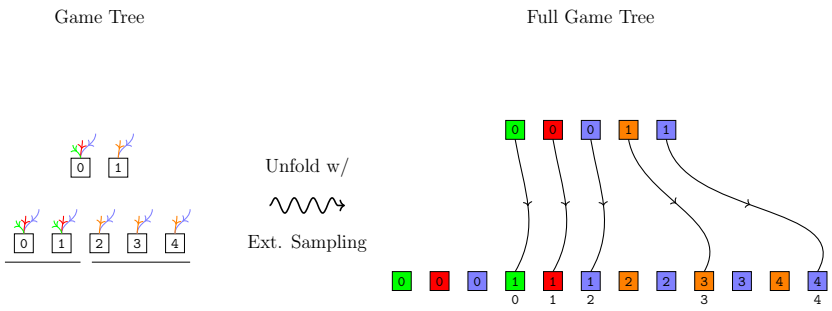


Figure 10: An illustration of how `t_edg` is constructed under ex. sampling regime.

current strategy. This selection is performed within FGT, as illustrated in Figure 10. To further aid understanding, Figure 11 illustrates how the FGT tensor `t_fgm` is updated/sampled. CodeSnippet 9 details the procedure used in external sampling.

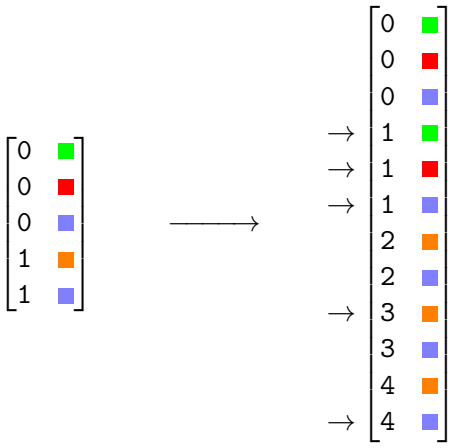


Figure 11: An illustration of how `t_fgm` is updated under ext. sampling regime.

CodeSnippet 9 ExternalSampling

```
Input t_edg,c_edg,t_sgm
  #t_edg links lower to upper nodes, while c_edg specifies the degree of each upper node.
  #Ex:  t_edg = [0,1,2,0,1,2,3,4,3,4,3,4], c_edg = t_brf⊗c_scr = [2,2,2,3,3]
t_inv = argsort(t_edg, stable=True)
  #t_inv = [0,3,1,4,2,5,6,8,10,7,9,11]
  #use t_inv when sorting edges.1st green maps to index 0, 2nd green maps to index 3, etc
t_idx = empty_like(t_inv)
t_idx[t_inv] ← arange(len(t_inv))
  #t_idx =[ 0, 2, 4, 1, 3, 5, 6, 9, 7, 10, 8, 11]
  #use t_idx when mapping back sorted edges
i_max = c_edg.max()
t_msk = arange(i_max).unsqueeze(0) < c_edg.unsqueeze(1)
  #t_msk = tensor([[ T, T, F], [ T, T, F], [ T, T, F], [ T, T, T], [ T, T, T]])
t_mtx = zeros_like(t_msk)
t_mtx[t_msk] ← t_sgm[t_inv]
t_smp = multinomial(t_mtx).squeeze(1)
  #sampled indices for each row
t_gps = cat([0, c_edg.cumsum()[:-1]])
t_res = zeros_like(t_sgm, dtype=bool)
t_res[t_gps + t_smp] ← True
t_res ← t_res[t_idx]
```

2.10 CFR Algorithm

In this section, we provide details on the PyTorch implementation of CFR for Pasur, following the framework outlined in the previous sections.

Our approach is as follows: for each round, we compute the average utility at the root-level nodes of the FGT. This utility is then propagated backward to the preceding round, where CFR is applied to compute the utilities for that round's root-level nodes. While this process is underway within a round, we simultaneously compute the average strategy profile (Equation (5)) for that round.

It is important to emphasize that when working within a round, the reach probabilities for the root-level nodes are all set to 1. Since Pasur consists of six rounds, we apply the CFR algorithm six times—once per round. The utility for the final round is computed directly by calculating the final score. Notably, the running-score tensor `t_rus` is added to the utility of the final-layer nodes before applying CFR.

We treat utility values with equal weights, based on the logic that each unit of utility represents a point earned under the Nash Equilibrium at that terminal node within the round. Because `t_rus` reflects the points accumulated during that round, it is appropriate to add it linearly to the computed utility.

This round-by-round implementation is both more accurate and memory-efficient. First, by computing utilities one round at a time rather than propagating utilities from terminal nodes all the way to the top of the full game tree, we improve numerical stability. This localized computation reduces the accumulation of floating-point

errors that can arise when backpropagating through long sequences. Second, by isolating the computation to a single round at a time, we only need to move the relevant tensors for that round to the GPU, while keeping the rest on the CPU. Importantly, all tensors involved in our framework are stored on the CPU for use in the CFR algorithm. This design choice significantly reduces GPU memory pressure during training. As a result, we require significantly less GPU VRAM to compute the Nash Equilibrium. On the other hand, the main downside is the increased complexity in implementation and debugging, as computations must now be carefully coordinated across multiple rounds.

We begin by obtaining the final scores from FGT's terminal nodes. This can be achieved using the following:

$$\begin{aligned} &\texttt{t\_fsc} = \texttt{7*(2*(t\_scr[:,-1] \% 2) - 1)} \\ &\texttt{t\_utl} = \texttt{t\_fsc[t\_fgm[:,1]]} \end{aligned} \tag{9}$$

Within CodeSnippet 7, we need to reverse Line (10) to recover `t_utl` at the terminal nodes. In other words, the `t_utl` tensor computed above can be interpreted as the root-level utility of the next round, which must be passed back to the terminal level of the current round. Once `t_utl` is obtained at the root level for each round, it should be propagated upward to the terminal nodes of the previous round and combined with the corresponding running-score tensor for each FGT node at that level. The following code accomplishes this:

$$\begin{aligned} &\texttt{t\_rus} = \texttt{t\_rus} \otimes \texttt{c\_scr} \\ &\texttt{t\_utl} = \texttt{t\_utl[t\_lnk]} + \texttt{t\_rus} \end{aligned} \tag{10}$$

Here `t_lnk` is defined in the final step of CodeSnippet 7 and `t_rus` appears in the same CodeSnippet, prior to being passed through `unique` in line 3.

The next three CodeSnippets present the CFR algorithm. There are two main components: the `FindUtility` and `FindReachProbability` functions. The overall CFR algorithm is shown in CodeSnippet 10.

CodeSnippet 10 CFR

```
function RUNCFR
    Compute g_utl # via Equation (9)
    for round in reversed(rounds) do
        g_utl ← Pass back up g_utl via Equation (10)
        Initialize g_reg, g_sgm, g_UTL
# g_reg (init. to 0): regrets for all the availble actions, r^t in Eq. (6),
# g_sgm (init. to uniform): strategy profile, σ^t in Eq. (4)
# g_UTL (init. to 0): root-node utilities for current round, R^t in Eq. (6)
        for t in range(NUM_ITER) do
            g_rpr ← FINDRPR(g_sgm)
            t_utl, g_sgm, a_sgm, g_reg ← FINDUTILITY(g_rpr, g_utl)
            g_UTL ← g_UTL + t_utl
        end for
        g_utl ← g_UTL/NUM_ITER
        g_sgm ← NORMALIZE(a_sgm) # Each FGT node's children's values sum to 1
    end for
end function
```

CodeSnippet 11 FindReachProbability

```
function FINDRPR(g_sgm, i_cfr) # g_sgm:current round's strategy. σ^t in Eq. (4)
    i_h ← # nodes in current round's root level
    t_rpr ← ones(i_h)
    d_rpr ← {}
    for i_trn in range(4) do
        for i_ply in range(2) do
            i_htp ← i_hnd_i_trn_i_ply
            t_sgm ← g_sgm[i_htp]
            t_sgm ← ones_like(t_sgm) if i_ply == i_cfr
            t_edg ← gt_edg[i_htp] # CUDA tensors gt_edg contains FGT edges
            d_rpr[i_htp] ← t_sgm * t_rpr[t_edg]
            t_rpr ← d_rpr[i_htp]
        end for
    end for
    return d_rpr
end function
```

CodeSnippet 12 FindUtility

```
function FindUtility(t_utl, g_sgm, a_sgm, g_reg, g_rp0, g_rp1)
    # Input:  gt_edg:  FGT edges, gc_edg:  FGT branch factor
    # TOL = 1e-5, a_sgm:  average strategy, g_rp0, g_rp1:  reach probs for Alex & Bob resp.
    # g_sgm:  current strategy
    for i_trn in reversed(range(4)) do
        for i_ply in reversed(range(2)) do

            i_htp ← i_hnd_i_trn_i_ply
            i_h ← # nodes in current layer's root-level

            t_ply, t_opp ← g_rp0[i_htp], g_rp1[i_htp] if i_ply == 1
            t_ply, t_opp ← g_rp1[i_htp], g_rp0[i_htp] if i_ply == 0

            c_edg,t_edg,t_sgm←gc_edg[i_htp], gt_edg[i_htp],g_sgm[i_htp]


            t_pt2 ← zeros(i_h)
            t_pt2.scatter_add_(0,t_edg,t_sgm*t_utl)

            t_reg ← (1-2*i_ply)*t_opp*(t_utl-t_pt2[t_edg])
            t_utl ← t_pt2

            t_msk ← g_reg[i_htp] >= TOL
            t_fct ← zeros_like(g_reg[i_htp])
            t_fct[t_msk] ← i_cnt**1.5/(1+i_cnt**1.5)
            t_fct[~t_msk] ← 0.5
            g_reg[i_htp] ← t_fct*g_reg[i_htp]+t_reg
            t_rts ← clamp(1000*g_reg[i_htp], min=TOL)

            t_sum ← zeros(i_h)
            t_sum.scatter_add_(0, t_edg, t_rts)

            t_msk ← t_sum[t_edg] <= TOL
            t_sgm[~t_msk] ← t_rts[~t_msk]/t_sum[t_edg][~t_msk]
            t_sgm[t_msk] ← (1/c_edg[t_edg])[t_msk]

            a_sgm[i_htp]←(1-1/(1+i_cnt)**2)*a_sgm[i_htp] + t_sgm*t_ply
            g_sgm[i_htp] ← t_sgm

        end for
    end for
    return t_utl, a_sgm, g_sgm, g_reg
end function
```

3 Experimental Evaluation

We trained more than 500 randomly generated decks using the CFR algorithm and obtain strategy profiles similar (stored in `.parquete`) to the one shown in Table 15. Figure 12 shows the distribution of the sizes of the resulting full game trees corresponding to the shapes of the strategy profile tables.

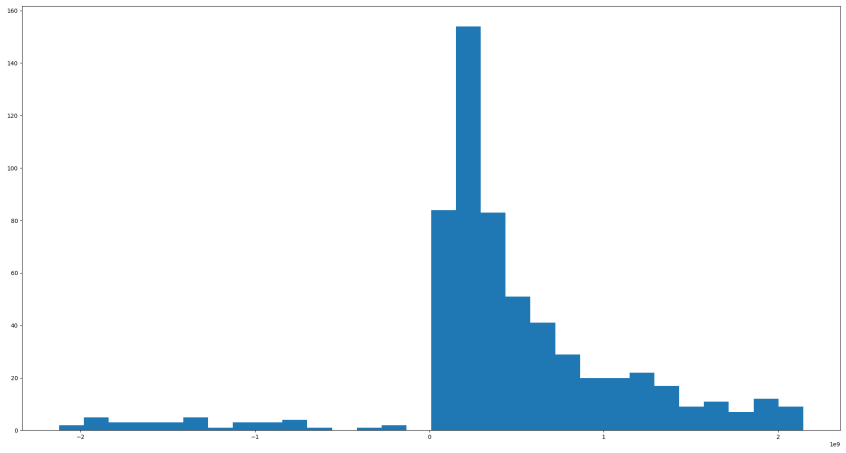


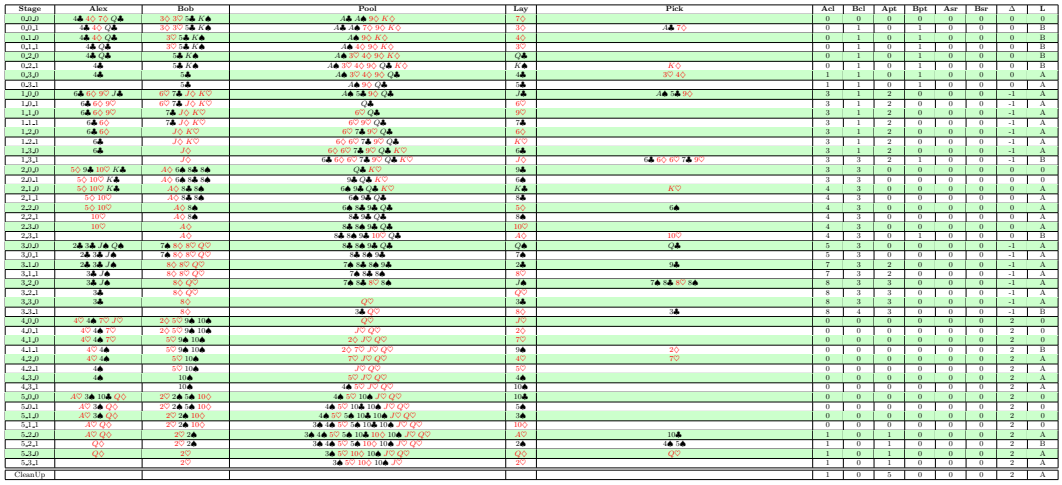
Figure 12: Distribution of game tree sizes (i.e., strategy profile table shapes) across 500 randomly generated decks.

We also illustrate a near-Nash equilibrium strategy, computed via CFR, in Figure 13 and Table 15. Figure 13 shows a sample game between two near-Nash strategies, while Table 15 lists their first-turn strategies in the first round.

Once we have the strategy profiles at our disposal, we fit an XGBoost model to predict $100 \times \sigma$, using root mean squared error (RMSE) as the evaluation metric and setting `gamma=1` as the regularization parameter. This allows us to pass the current infoset to the model in order to obtain a near-Nash strategy for the active player. The main advantage of this approach is that it enables the simulation of multiple self-play games in parallel on the GPU. Specifically, if `N` denotes the number of self-play instances, then the infoset tensor at each turn has shape `[N, 58]`. Sampling based on the learned strategy profile is performed using the same method introduced in CodeSnippet 9.

With the ability to run multiple self-play games in parallel, we can now estimate the fair value of each deck composition. To do so, we let two near-Nash equilibrium strategies compete against each other using a given deck. The percentage of games won by Alex, divided by 100, is what we define as the fair value of that deck. As a first sanity check during development, we evaluated our model against a random strategy. Figure 14 presents the results: the left panel shows near-Nash vs. near-Nash, the middle panel shows near-Nash vs. Random, and the right panel shows Random vs. near-Nash.

Sample Game Played Between Near-Nash Strategies



Stage	Alex	Bob	Pool	Lay	Pick	Acl	Bcl	Apt	Bpt	Asr	Bsr	Δ	L
0,0,0	4♣ 4♢ 7♢ Q♣	3♢ 3♡ 5♣ K♣	A♣ A♠ 9♢ K♢	7♢		0	0	0	0	0	0	0	0
0,0,1	4♣ 7♢ Q♣	3♢ 3♡ 5♣ K♣	A♣ A♠ 7♢ 9♢ K♢	3♢	A♣ 7♢	0	1	0	1	0	0	0	B
0,1,0	4♣ 4♢ Q♣	3♢ 5♣ K♣	A♣ 9♢ K♢	4♢		0	1	0	1	0	0	0	B
0,1,1	4♣ Q♣	3♢ 5♣ K♣	A♣ 4♢ 9♢ K♢	3♢		0	1	0	1	0	0	0	B
0,2,0	4♣ Q♣	5♣ K♣	A♣ 3♡ 4♢ 9♢ K♢	Q♣		0	1	0	1	0	0	0	B
0,2,1	4♣	5♣ K♣	A♣ 3♡ 4♢ 9♢ Q♣ K♢	K♣	K♢	0	1	0	1	0	0	0	B
0,3,0	4♣	5♣	A♣ 3♡ 4♢ 9♢ Q♣	4♣	3♡ 4♢	1	1	0	1	0	0	0	A
0,3,1		5♣	A♣ 9♢ Q♣	5♣		1	1	0	1	0	0	0	A
1,0,0	6♣ 6♢ 9♡ J♣	6♡ 7♣ J♢ K♡	A♣ 5♣ 9♢ Q♣	J♣	A♣ 5♣ 9♢	3	1	2	0	0	0	-1	A
1,0,1	6♣ 6♢ 9♡	6♡ 7♣ J♢ K♡	Q♣	6♡		3	1	2	0	0	0	-1	A
1,1,0	6♣ 6♢ 9♡	7♣ J♢ K♡	6♡ Q♣	9♡		3	1	2	0	0	0	-1	A
1,1,1	6♣ 6♢	7♣ J♢ K♡	6♡ 9♡ Q♣	7♣		3	1	2	0	0	0	-1	A
1,2,0	6♣ 6♢	J♢ K♡	6♡ 7♣ 9♡ Q♣	6♢		3	1	2	0	0	0	-1	A
1,2,1	6♣	J♢ K♡	6♢ 6♡ 7♣ 9♡ Q♣	K♡		3	1	2	0	0	0	-1	A
1,3,0	6♣	J♢	6♢ 6♡ 7♣ 9♡ Q♣ K♡	6♣		3	1	2	0	0	0	-1	A
1,3,1		J♢	7♣ 6♢ 6♡ 7♣ 9♡ Q♣ K♡	J♢	7♣ 6♢ 6♡ 7♣ 9♡	3	3	2	1	0	0	-1	B
2,0,0	5♢ 9♣ 10♡ K♣	A♢ 6♣ 8♣ 8♣	Q♣ K♡	9♣		3	3	0	0	0	0	0	0
2,0,1	5♢ 10♡ K♣	A♢ 6♣ 8♣ 8♣	8♣ Q♣ K♡	6♣		3	3	0	0	0	0	0	0
2,1,0	5♢ 10♡ K♣	A♢ 8♣ 8♣	6♣ 8♣ Q♣ K♡	K♣	K♡	4	3	0	0	0	0	0	A
2,1,1	5♢ 10♡	A♢ 8♣ 8♣	6♣ 8♣ Q♣	8♣		4	3	0	0	0	0	0	A
2,2,0	5♢ 10♡	A♢ 8♣	6♣ 8♣ 8♣ Q♣	5♢	6♣	4	3	0	0	0	0	0	A
2,2,1	10♡	A♢ 8♣	8♣ 8♣ Q♣	8♣		4	3	0	0	0	0	0	A
2,3,0	10♡	A♢	8♣ 8♣ 8♣ Q♣	10♡		4	3	0	0	0	0	0	A
2,3,1		A♢	8♣ 8♣ 8♣ 10♡ Q♣	A♢	10♡	4	3	0	1	0	0	0	B
3,0,0	2♣ 3♣ J♣ Q♣	7♣ 8♢ 8♡ Q♡	8♣ 8♣ 8♣ Q♣	Q♣	Q♣	5	3	0	0	0	0	-1	A
3,0,1	2♣ 3♣ J♣	7♣ 8♢ 8♡ Q♡	8♣ 8♣ 8♣	7♣		5	3	0	0	0	0	-1	A
3,1,0	2♣ 3♣ J♣	8♢ 8♡ Q♡	7♣ 8♣ 8♣ 8♣	2♣	9♣	7	3	2	0	0	0	-1	A
3,1,1	3♣ J♣	8♢ 8♡ Q♡	7♣ 8♣ 8♣	8♡		7	3	2	0	0	0	-1	A
3,2,0	3♣ J♣	8♢ Q♡	7♣ 8♣ 8♡ 8♣	J♣	7♣ 8♣ 8♡ 8♣	8	3	3	0	0	0	-1	A
3,2,1	3♣	8♢ Q♡		Q♡		8	3	3	0	0	0	-1	A
3,3,0	3♣	8♢	Q♡	3♣		8	3	3	0	0	0	-1	A
3,3,1		8♢	3♣ Q♡	8♢	3♣	8	4	3	0	0	0	-1	B
4,0,0	4♢ 4♣ 7♡ J♡	2♢ 5♡ 9♣ 10♣	Q♣	J♡		0	0	0	0	0	0	2	0
4,0,1	4♢ 4♣ 7♡	2♢ 5♡ 9♣ 10♣	J♡ Q♣	2♢		0	0	0	0	0	0	2	0
4,1,0	4♢ 4♣ 7♡	5♡ 9♣ 10♣	2♢ J♡ Q♣	7♡		0	0	0	0	0	0	2	0
4,1,1	4♢ 7♣	5♡ 9♣ 10♣	2♢ 7♡ J♡ Q♣	9♣	2♢	0	0	0	0	0	0	2	B
4,2,0	4♢ 4♣	5♡ 10♣	7♡ J♡ Q♣	4♢	7♡	0	0	0	0	0	0	2	A
4,2,1	4♣	5♡ 10♣	J♡ Q♣	5♡		0	0	0	0	0	0	2	A
4,3,0	4♣	10♣	5♡ J♡ Q♣	4♣		0	0	0	0	0	0	2	A
4,3,1		10♣	4♣ 5♡ J♡ Q♣	10♣		0	0	0	0	0	0	2	A
5,0,0	A♡ 3♣ 10♣ Q♡	2♡ 2♣ 5♣ 10♡	4♣ 8♡ 10♣ J♡ Q♡	10♣		0	0	0	0	0	0	2	0
5,0,1	A♡ 3♣ Q♡	2♡ 2♣ 5♣ 10♡	4♣ 8♡ 10♣ 10♣ J♡ Q♡	5♣		0	0	0	0	0	0	2	0
5,1,0	A♡ 3♣ Q♡	2♡ 2♣ 10♡	4♣ 8♡ 5♣ 10♣ 10♣ J♡ Q♡	3♣		0	0	0	0	0	0	2	0
5,1,1	A♡ Q♡	2♡ 2♣ 10♡	3♣ 4♣ 8♡ 5♣ 10♣ 10♣ J♡ Q♡	10♡		0	0	0	0	0	0	2	0
5,2,0	A♡ Q♡	2♡ 2♣	3♣ 4♣ 8♡ 5♣ 10♣ 10♣ J♡ Q♡	A♡	10♣	1	0	1	0	0	0	2	A
5,2,1	Q♡	2♡ 2♣	3♣ 4♣ 8♡ 5♣ 10♡ 10♣ J♡ Q♡	2♣	4♣ 5♣	1	0	1	0	0	0	2	B
5,3,0	Q♡	3♡	3♣ 5♡ 10♡ 10♣ J♡ Q♡	Q♡	Q♡	1	0	1	0	0	0	2	A
5,3,1		3♡	3♣ 8♡ 10♡ 10♣ J♡	3♡		1	0	1	0	0	0	2	A
CleanUp						1	0	5	0	0	0	2	A

Figure 13: Some highlight of sound decisions made by both players: Since both players have full knowledge of each other's hands, Alex should avoid playing either 4♣ or 4♢. Doing so would allow Bob to respond immediately by collecting A♣ and A♠ with his 5♣, thereby securing 2 points and at least two clubs. Assuming instead that Alex plays A♣ and A♠ on his first move, Bob's optimal response is to collect using his 5♣. Finally, if Alex plays 7♢, Bob should respond by picking up one Ace using either his 3♢ or 3♡. It is emphasized that the strategy profile in Table 15 reflects all of these observations.

A♣	A♠	3♢	3♡	4♣	4♢	5♣	7♢	9♢	Q♣	K♢	K♠	H	T	P	σ
110	110	105	105	100	1	105	100	110	100	110	105	0	0	0	0.00
110	110	105	105	1	100	105	100	110	100	110	105	0	0	0	0.00
110	110	105	105	100	100	105	1	110	100	110	105	0	0	0	1.00
110	110	105	105	100	100	105	100	110	1	110	105	0	0	0	1.00
110	110	105	-1	100	1	105	100	110	100	110	105	0	0	1	0.00
110	110	-1	105	100	1	105	100	110	100	110	105	0	0	1	0.00
-10	-10	105	105	100	-9	109	100	110	100	110	105	0	0	1	1.00
110	110	105	105	100	1	105	100	110	100	-10	109	0	0	1	0.00
110	110	-1	105	1	100	105	100	110	100	110	105	0	0	1	0.00
110	110	105	-1	1	100	105	100	110	100	110	105	0	0	1	0.00
110	110	105	105	1	100	105	100	110	100	-10	109	0	0	1	0.00
-10	-10	105	105	-9	100	109	100	110	100	110	105	0	0	1	1.00
110	110	105	105	100	100	105	1	110	100	-10	109	0	0	1	0.00
110	-10	109	105	100	100	105	-9	110	100	110	105	0	0	1	0.25
-10	110	105	109	100	100	105	-9	110	100	110	105	0	0	1	0.25
110	110	105	105	100	100	-1	1	110	100	110	105	0	0	1	0.00
-10	110	109	105	100	100	105	-9	110	100	110	105	0	0	1	0.25
110	-10	105	109	100	100	105	-9	110	100	110	105	0	0	1	0.25
110	110	-1	105	100	100	105	100	110	1	110	105	0	0	1	0.00
110	110	105	105	100	100	105	100	110	1	-10	109	0	0	1	1.00
110	110	105	105	100	100	-1	100	110	1	110	105	0	0	1	0.00
110	110	105	-1	100	100	105	100	110	1	110	105	0	0	1	0.00

Table 15: Near-Nash strategies for first turn of Figure 13's deck

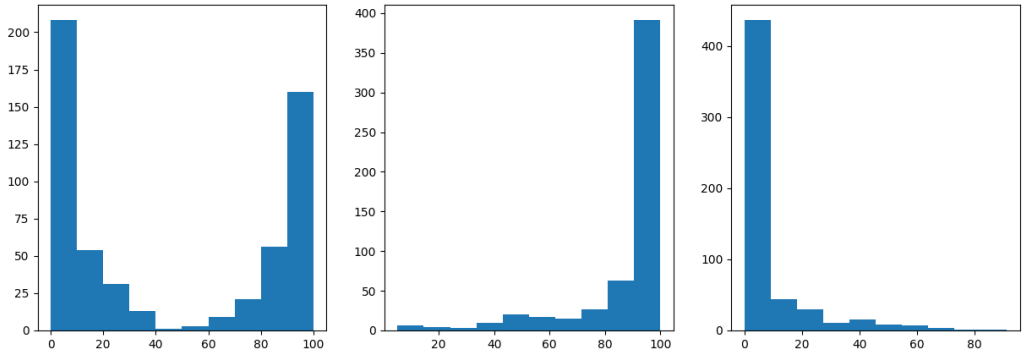


Figure 14: Left panel shows near-Nash vs. near-Nash, middle panel shows near-Nash vs. Random, and the right panel shows Random vs. near-Nash

# J	# ♣	v	std
$= 0$	≤ 7	0.06	0.03
$= 0$	> 7	0.22	0.11
$= 1$	≤ 5	0.06	0.02
$= 1$	> 5	0.25	0.12
$= 2$	≤ 6	0.36	0.15
$= 2$	> 6	0.54	0.17
$= 3$	≤ 5	0.58	0.15
$= 3$	> 5	0.76	0.10
$= 4$	≤ 4	0.66	0.11
$= 4$	> 4	0.85	0.09

Table 16: Deck fair values vs. Jacks and Clubs held by Alex.

Another step in our analysis is to relate the fair value of a deck to key elements of its composition. For example, it is clear that holding Jack cards or Club cards provides a strategic advantage to the player. To investigate this relationship, we fit decision trees to subsets of decks, each categorized by the number of Jack cards held by Alex. We then examine how the decision tree branches based on the number of Clubs held by Alex. Table 16 illustrates this relationship: the more Jacks or Clubs Alex holds, the higher the fair value of the deck tends to be; providing supporting evidence that our CFR algorithm has converged to near-Nash equilibrium strategies.

We next examine the convergence behavior of CFR. To this end, we measure the mean squared error (MSE) between each intermediate strategy profile $\bar{\sigma}^t$ and the final CFR strategy vector. Since our approach computes strategies on a round-by-round basis, we provide separate plots for each round to visualize the convergence. It is important to note that, as shown in CodeSnippet 12, we use the DCFR algorithm from [2] with the following parameters:

$$\gamma = 2, \quad \alpha = 1.5, \quad \beta = 0.$$

Figure 15 shows the CFR convergence behavior. Finally, Figure 16 shows the σ value for the first turn of Alex for Deck of Figure 13.

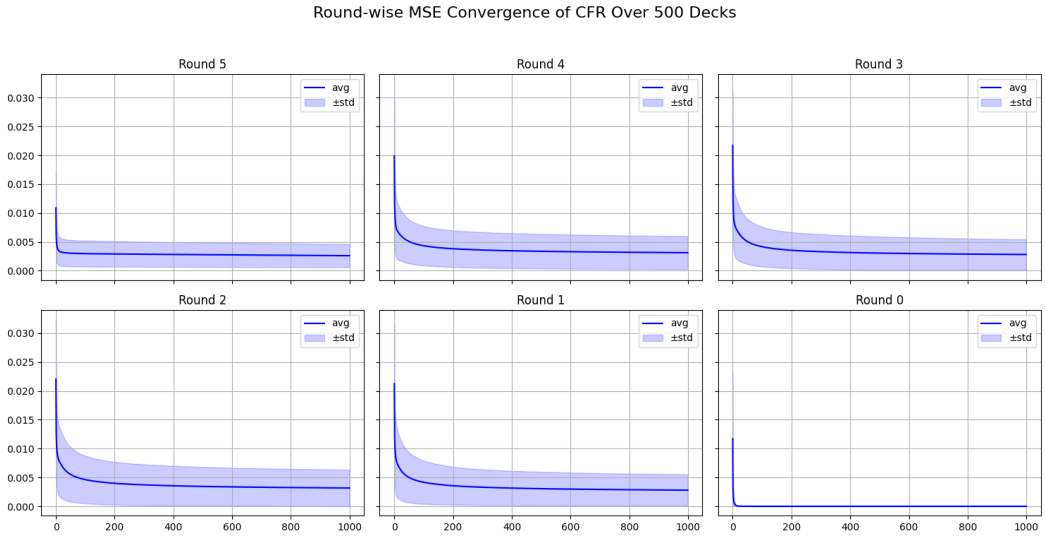


Figure 15: Convergence of CFR for 500 random decks. The plot shows the average ± standard deviation bands of the mean squared error (MSE) to the final CFR vector, measured separately for each round. As observed, the last round converges the fastest due to having the fewest nodes in the FT representation.

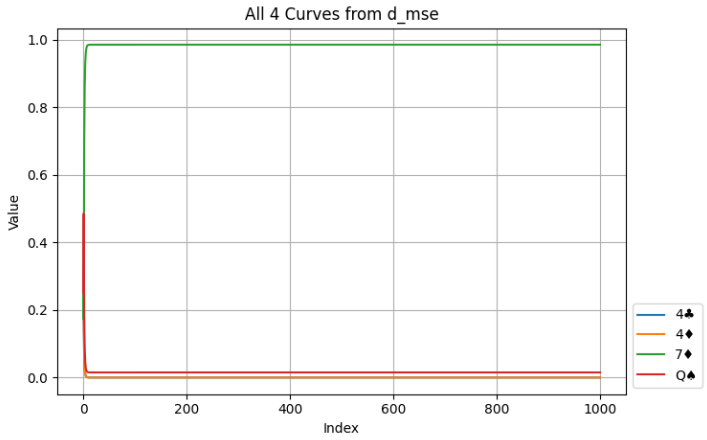


Figure 16: σ values for Alex's first turn using the deck from Figure 13

4 Future Work

We conclude the paper by outlining two promising directions for future research.

First, we need to extend the current framework to the more realistic and challenging setting where players do not have access to the opponent's hand information. This extension could proceed in two stages. First, the possible opponent deck compositions can be narrowed down to a small set of plausible candidates. Then, leveraging the external sampling technique introduced in Section 2.9, one could apply methods similar to those proposed in [1] to train general tree-based models and use CFR to solve Pasur in the most general setting. The development and evaluation of this more general framework are left to future work.

Second, it would be valuable to investigate whether a single XGBoost model can be trained to generalize across multiple strategy profiles. This investigation remains within the setting where both players have full knowledge of each other's hands, with the added complexity that multiple deck decompositions may be possible and gradually revealed as the game progresses. A key challenge in this context arises when decks share partial similarity—for example, two decks may have identical first-round compositions but differ in later rounds. Such differences can influence strategies even in the early stages, causing the same information sets to map to different actions. To address this, strategy training must account for this additional source of randomness.

Finally, another important direction for future work is to approximate the exploitability of the strategies trained in this paper. This would help validate convergence toward a true Nash equilibrium. Exploitability quantifies how far a strategy profile deviates from equilibrium, lower values indicate closer approximation to optimal play. A natural starting point would be the simplified setting where both players have full knowledge of each other's hands. Once exploitability is well-understood in this restricted scenario, approximation techniques can be applied to the general case to estimate the exploitability of the learned strategies.

References

[1] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. In *International conference on machine learning*, pages 793–802. PMLR, 2019.

[2] Noam Brown and Tuomas Sandholm. Solving imperfect-information games via discounted regret minimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1829–1836, 2019.

[3] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. *Advances in neural information processing systems*, 22, 2009.

[4] John F Nash Jr. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950.

[5] Oskari Tammelin. Solving large imperfect information games using cfr+. *arXiv preprint arXiv:1407.5042*, 2014.

[6] Hang Xu, Kai Li, Bingyun Liu, Haobo Fu, Qiang Fu, Junliang Xing, and Jian Cheng. Minimizing weighted counterfactual regret with optimistic online mirror descent. *arXiv preprint arXiv:2404.13891*, 2024.

[7] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. *Advances in neural information processing systems*, 20, 2007.

5 Appendix: RepeatBlocks

RepeatBlocks is a generalization of `torch.repeat_interleave` in which, instead of repeating individual elements, entire blocks of elements are repeated. Each block has a custom size and a custom repetition count. The idea is best explained through an example. Consider the following

$$\texttt{t\_org = [1,2,3,4,5,6], t\_bsz = [2,3,1],t\_rpt = [1,3,2]}$$

Here `t_org,t_bsz,t_rpt` are the input tensor, block sizes, and repeat counts respectively. The desired output denoted by $\texttt{t\_org} \otimes_{\texttt{t\_bsz}} \texttt{t\_rpt}$ is

$$\texttt{t\_org} \otimes_{\texttt{t\_bsz}} \texttt{t\_rpt = [1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 6, 6]}$$

For convenience, let us denote $\texttt{t\_rbk} = \texttt{t\_org} \otimes_{\texttt{t\_bsz}} \texttt{t\_rpt}$. To construct `t_rbk`, it suffices to build the corresponding index tensor `t_idx`.

$$\texttt{t\_idx = [0, 1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 5, 5]}$$

Note that `t_rbk = t_org[t_idx]`. Next, consider the corresponding start indices of each repeated block

$$\texttt{t\_blk = [0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 5, 5],}$$

It is emphasized that using input tensors, `t_blk` is given as below.

$$\texttt{t\_blk = (cat([0,t\_bsz[:-1]])} \otimes \texttt{t\_rpt)} \otimes \underbrace{(\texttt{t\_bsz} \otimes \texttt{t\_rpt})}_{:=\texttt{t\_bls}}$$

Deducting `t_blk` from `t_idx`, we arrive at the following tensor.

$$\texttt{t\_pos = [0, 1, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 0]}$$

Next,

$$\texttt{arange(t\_blk.shape[0])-t\_pos = [0, 0, 2, 2, 2, 5, 5, 5, 8, 8, 8, 11, 12]}$$

where a simple examination shows that the RHS is equal

$$\texttt{cumsum(cat([0, t\_bls[:-1]]))} \otimes \texttt{t\_bls}.$$

Putting pieces together, we arrive at Algorithm 13.

CodeSnippet 13 RepeatBlocks

Input `t_org, t_bsz, t_rpt`

`# t_org:input tensor, t_bsz:blocks sizes, t_rpt:repeat counts`

`# Ex: t_org = [1,2,3,4,5,6], t_bsz = [2,3,1],t_rpt = [1,3,2]`

`t_bgn` ← `cat([0,t_bsz[:-1]])`⊗`t_rpt`

Compute beginning indices of each output's block

`# Ex: t_bgn = [0, 2, 2, 2, 5, 5]`

`t_bls` ← `t_bsz`⊗`t_rpt`

Compute sizes of each output's block

`# Ex: t_bls = [2, 3, 3, 3, 1, 1]`

`t_blk` ← `t_bgn`⊗`t_bls`

t_blk[i] = j i.e., output[i] belongs to block j

`# Ex: t_blk = [0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 5, 5]`

`t_csm` ← `cumsum(0,t_bls[:-1])`⊗`t_bls`

`# Ex: t_csm=[ 0, 2, 5, 8, 11, 12]`⊗`t_bls=[ 0, 0, 2, 2, 2, 5, 5, 5, 8, 8, 8, 11, 12]`

`t_pos` ← `arange(i_blk)-t_csm`

`# Ex: t_pos=[0, 1, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 0]`

`t_idx` ← `t_pos+t_blk`

`# Ex: t_idx=[0, 1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 5, 5]`

`t_rbk` ← `t_org[t_idx]`

`# Ex: t_rbk=[1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 6, 6]`

Output `t_rbk`

denoted by $\texttt{t\_org}\otimes_{\texttt{t\_bsz}}\texttt{t\_rpt}$