Solving Pasur Using GPU-Accelerated CFR

July 3, 2025

Contents

1 Introduction

2 Notation and Background

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 $u_p \in \mathcal{P}$, there is a payoff function $p : \mathcal{Z} \to \mathbb{R}$. 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 $h \in A(I)$ and $I \in \mathcal{I}_p$, 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

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

Similarly, counterfactual utility for $a \in A(I)$, $u^\sigma(I, a)$ is defined as in (2), 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

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

Finally, in a two-player extensive game a *Nash equilibrium* [**?**] 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).

2.1 Counterfactual Regret Minimization (CFR)

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 $O\left(\sqrt{\frac{1}{T}}\right)$. At the heart of CFR lies the concept of *regret*. For a strategy profile σ, *instantaneous regret* of playing a vs. σ at I is denoted by

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

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

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

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

$$\bar{\sigma}^T_i(I)(a) \propto \sum_{t=1}^{T} \pi^{\sigma^t}_i(I)\, \sigma^t(I)(a). \tag{5}$$

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, [?]) *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).

In this paper, we also use a variant of CFR known as CFR$^+$ [**?**], which differs from vanilla CFR only in the way it updates counterfactual regrets. These regrets, initialized at zero, are updated according to the following rule:

$$R^{+,T}(I, a) = \left(R^{+,T-1}(I, a) + u^{\sigma^T}(I, a) - u^{\sigma^T}(I) \right)^+ \tag{6}$$

2.2 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 these rules:
  - A numeric card can capture a combination of numeric cards from the pool if their total sum equals 11.
  - A Queen can only capture another single Queen. The same rule applies to Kings.
  - A Jack captures all cards in the pool (including other Jacks), except for Kings and Queens.

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 of the game. Table 1 outlines the scoring system used in Pasur. Notably, a **Sur**—defined below—grants a 5-point bonus.

A **Sur** occurs when a player captures all the cards from the pool in a single move. There are two important exceptions:

- A Sur cannot be made using a Jack.
- Surs are not permitted during the final round of play.

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

Table 1: 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♢	4♢		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♣	5♠		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♣ 8♣ 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♠ 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♡	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♠ 5♠ 10♢	4♣ Q♣	A♡		0	0	0	0	0	0	3	0	A
5_0_1	3♠ 10♣ Q♢	2♡ 2♠ 5♠ 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 1 for the Pasur scoring system.

3 PyTorch-Based Framework

In this section, we present our framework for implementing the CFR algorithm for *Pasur* using PyTorch. We begin by describing how the game tree is generated and recorded using tensor operations. The game consists of 5 rounds, and in each round, both players take 4 turns. Therefore, the full game comprises 40 steps, giving rise to a game tree of depth 40.

During the tree construction process, we distinguish between 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 4 illustrates this structural design.

Alongside the Game Tree illustrated in Figure 4, we construct a *Full Game Tree* via an *unfolding process*, which systematically expands the tree by combining each card state with all compatible incoming scores.

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

Each node in the Full Game Tree is thus represented as a pair: a Game Tree node and its corresponding incoming score. We also explicitly maintain the edge structure between nodes, which is crucial for updating strategies during the CFR iterations. Figure 10 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.

Table 3 summarizes the key components involved in our framework. Before proceeding to the next section, we first describe 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 3.

$A\clubsuit$	$A\diamondsuit$	$A\heartsuit$	$A\spadesuit$	$2\clubsuit$	$2\diamondsuit$	. . .	$Q\heartsuit$	$Q\spadesuit$	$K\clubsuit$	$K\diamondsuit$	$K\heartsuit$	$K\spadesuit$

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 `N` 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, `N = 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 following initial configuration of the game, shown in Table 2.

Table 2: Initial setup for a single game instance

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

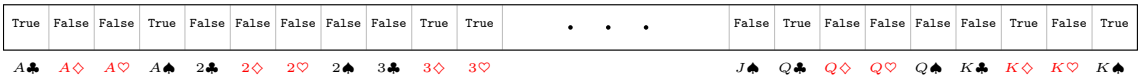

Figure 3: `t_inp` tensor at the beginning of the game for the initial setup shown in Table 2

Table 3: Summary of Tensors Used in Game Tree Construction

Component	Tensor	Shape	Type	Update	Description
Game	`t_gme`	`[M,3,N]`	`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 `[i, s]` in `t_fgm` indicates that node `i` in the game tree `t_gme` inherits the score with ID `s` from `t_scr`. Here, `Q` denotes the number of nodes in the current layer of the Full Game Tree.
Scores	`t_scr`	`[S,4]`	`int8`	Per round	Unique score triples in the form (Alex clubs, Bob clubs, Point Difference).
Action	`t_act`	`[M',2,N]`	`int8`	Per turn	Action representation per node. `[0,:]` encodes the lay card; `[1,:]` encodes picked cards. Here, `M'` denotes the number of nodes in the next layer of the Game Tree.
Branch Factor	`t_brf`	`[M]`	`int8`	Per turn	Number of valid actions available from each node; used to replicate game states before applying actions.
Linkage	`t_lnk`	`[M,2]`	`int8`	Per round	Connects nodes between consecutive hands to identify how scores and states map across hands. Used during in-between-hand updates.
Edge	`t_edg`	`[Q',2]`	`int32`	Per turn	Records edges between nodes in the full game tree to track the structure of the overall graph. Here, `Q'` denotes the number of nodes in the next layer of the Full Game Tree.
Strategy	`t_sgm`	`[Q]`	`float32`	Per turn	Stores the strategy values (e.g., probabilities) associated with each node in the Full Game Tree.
Infoset	`t_inf`	`[Q,58]`	`int8`	Per turn	Encodes information available to the player whose turn it is. `t_inf` hides information that is not observable by the acting player. It also includes metadata such as round index, turn counter, and cumulative score up to the previous round.

This section is organized as follows. First, in Subsection 3.1, we describe the construction of the *Game Tensors*. Next, Subsection 3.2 explains how the *Action Tensors* are built and how they are used to update the Game Tensors. Then, in Subsection 3.4, we detail how the `t_fgm` tensor is updated within each round; this subsection also introduces the construction of the Edge Tensor `t_edg`. Following that, Subsection 3.5 describes the Between-Hand updates, including how the Score Tensor and Full Game Tensor are updated at the end of each

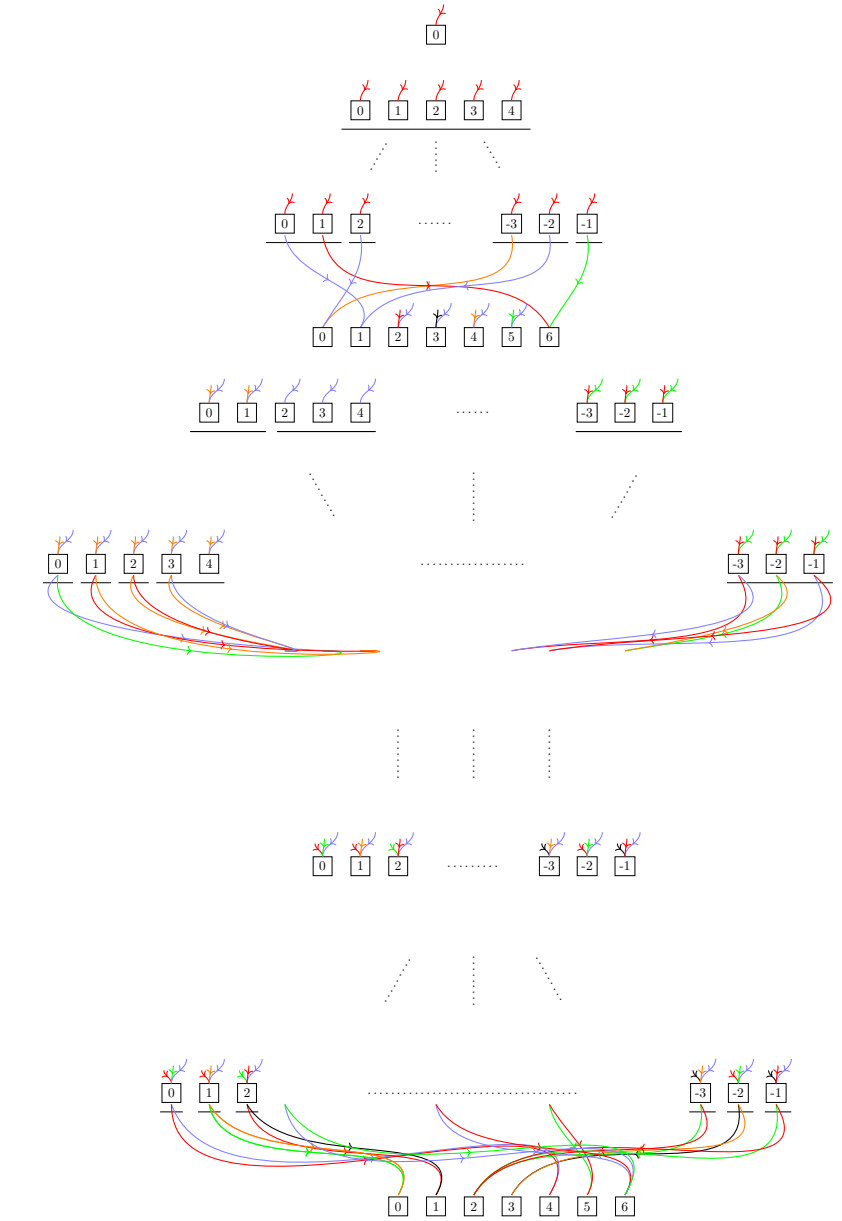

Figure 4: Game Tree

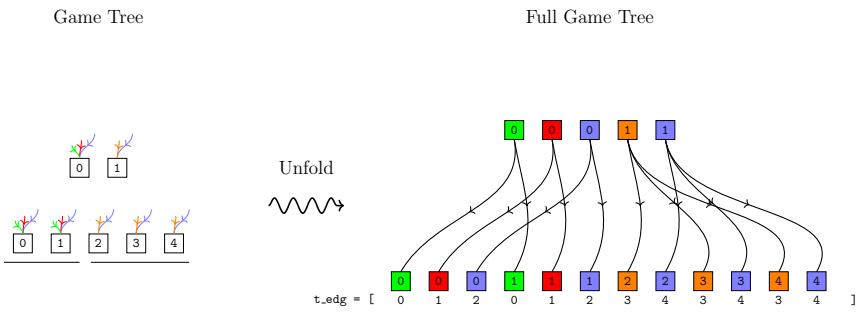

Figure 5: Unfolding Process & In-Hand Update

round, along with the Linkage Tensor `t_lnk`. Subsection 3.3 discusses the update process for the In-Play Cards and Out-of-Play Cards tensors. Finally, Subsection 3.6 explains how the Infoset Tensors are constructed.

3.1 Game Tensor

Each row in the game tree (Figure 4) is represented using an `M*3*N` Game Tensor `t_gme`, where `M` corresponds to the number of nodes in that layer of the tree. The the associated `3*N` tensor is constructed for each node:

Each slice `t_gme[i,:,:]` encodes a single game state. The first row of the tensor, `t_gme[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 4.

Table 4: State Encoding in `t_gme[0,:]`

Element	Encoding
Alex	1
Bob	2
Pool	3

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

3.1.1 Compressed Game Tensor

In this subsection, we represent a game tensor `t_gme` of shape `[M, 3, N]` using a compressed form `t_cmp` of shape `[M, N]`. This compressed tensor is later used to build the Infoset representation in Section 3.6. During Game Tree 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 Game Tree layer, given the current In-Play tensor `t_inp`. See CodeSnippet 1.

In Table 6, we summarize all possible values that can appear in the compressed tensor `t_cmp` (see CodeSnippet 1). Each card can have up to two *legs* in a round. The first leg

CodeSnippet 1 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[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
```

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 5: Leg Status Conditions for Cards

No Legs	One Leg	Both Legs in Same Turn
`t_cmp==0`	`t_gme[:,0,:]==3` and `t_cmp!=0`	`t_gme[:,0,:]==0` and `t_cmp!=0`

Table 6: 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`	

3.2 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[i]` records the number of valid actions available from node `i`. Correspondingly, `t_act` is a binary tensor of shape `[M',2,N]`, where `N` 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 and consists of two rows:

- The first row `t_act[j,0,:]` encodes the *lay card*. It contains exactly one `1` indicating

the played card, with zeros elsewhere.

- The second row `t_act[j,1,:]` encodes the *picked cards* from the pool. This row may contain zero or more `1`s depending on the number of cards picked in that action.

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

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

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, as shown in CodeSnippet 2. Finally, we apply the encoded actions by updating `t_gme` using `t_act`, as detailed in CodeSnippet 2.

3.2.1 Find Actions

In this subsection, we present how Action Tensor `t_act` is constructed using Game Tensor `t_gme`.

CodeSnippet 2 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)
```

3.3 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 the Game Tree, 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`. Table **??** outlines the meaning of each index in the score tensor `t_scr`. In contrast, the running-score tensor `t_rus` contains additional components relevant during a single round, which are described in Table **??**.

Table 7: Column definitions for the score tensor

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

Table 8: 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

The Full Game Tensor `t_fgm` is used to track which scores accumulated from previous rounds (stored in `t_scr`) are linked to each node of the Game Tree tensor `t_gme`. A row entry `[i, s]` in `t_fgm` indicates that node `t_gme[i]` inherits the score `t_scr[s]` from a previous round. For example, during the first round (*i.e.*, `i_rnd = 0`), the second column of `t_fgm` is identically `0`, since `t_scr` has shape `[1, 4]` at this point. In other words, the only available score from previous rounds is `[0, 0, 0, 0]`. Figure 4 illustrates this point, where all edges corresponding to the first round are highlighted in red.

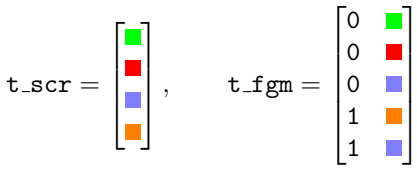

Figure 6: An example illustrating how the Full Game Tensor `t_fgm` associates nodes in the Game Tree with entries in the Score Tensor `t_scr`. Colors are used to indicate rows of `t_scr`, consistent with Figure 4. Each row of `t_scr` is represented by a tensor of size 4, as described in Table 7.

3.4 In-Hand Updates

In this section, we explain how in-hand updates are performed. Before proceeding, we introduce a generalized version of `torch.repeat_interleave`, referred to as **RepeatBlocks**, which will be used throughout this section. Unlike `torch.repeat_interleave`, which repeats individual elements, RepeatBlocks operates on and repeats entire contiguous blocks of elements. This operation is described in detail in the next subsection.

3.4.1 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 3.

CodeSnippet 3 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, 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 t_org⊗_{t_bsz}t_rpt
```

3.4.2 Full Game Tree

In this section, we explain how the Full Game Tree is updated within each round. Two tensors are updated during this process: the Full Game Tensor `t_fgm` and the Edge Tensor `t_edg`, which captures the connections between nodes across successive layers of the Full Game Tree.

Figure 5 illustrates how the Edge Tensor `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 the Full Game Tree. Here, colors denote inherited scores from the previous round, and numbers represent the row indices from the current game tensor `t_gm`. An edge is drawn when the score colors match and the nodes are connected in the folded Game Tree. In this example, the tensor

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

records the parent indices in the Complete Tree. The counts of inherited scores and available 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 full game tensor `t_fgm` is updated with an example provided in Figure 7. Consider the example in Figure 5 again with their `t_fgm` tensors shown in Figure 7. We update `t_fgm` in two phases: first, we update `t_fgm[:,1]`; then, we update `t_gme` and `c_scr` before continuing to update `t_fgm[:,0]`. We have

$$\texttt{t\_fgm[:,1]} \leftarrow \left[\begin{smallmatrix}\blacksquare\\ \blacksquare\\ \blacksquare\\ \blacksquare\\ \blacksquare\end{smallmatrix}\right] \otimes_{\texttt{[3,2]}} \texttt{[2,3]}$$

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 values are repeated three times because `t_brf[1] = 3`. Therefore, the following is true.

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

Furthermore,

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

In words, each node in the second layer of the Game Tree 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 in the equation above.

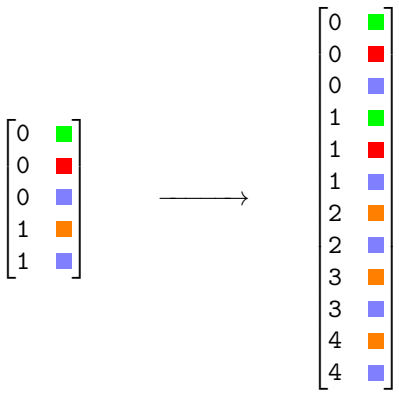

Figure 7: An illustration of how `t_fgm` is updated.

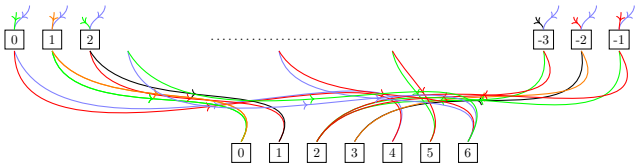

Figure 8: An illustration of between-hand processing

3.5 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 hyper-nodes, each associated with one or more inherited scores from previous rounds.

To begin, we identify the unique game states among all generated hyper-nodes. This is achieved via the following operation:

```
t_gme, t_edg = unique(t_gme, dim=0, return_inverse=True)
```

This assigns a unique index to each distinct game state (card configuration), and we update the first column of the Full Game tensor accordingly:

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

Note that this is an intermediate step: the final version of `t_fgm` will be updated and sorted 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` tensor for the upcoming round. As shown in Figure 8, a hyper-node (e.g., node 0) may map to another node (e.g., node 6) after applying the `unique` operation, with multiple distinct scores inherited from different incoming paths.

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`. We then concatenate these two components and identify the unique score combinations:

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

We apply the `unique` operation to extract distinct score transitions:

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

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

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

Finally, we eliminate duplicate score values using another call to `unique`, and update both the `t_scr` (score tensor) and the second column of `t_fgm` accordingly:

$$\begin{aligned}\texttt{t\_scr}, \texttt{t\_fid} &\leftarrow \texttt{unique(t\_scr, dim=0, return\_inverse=True)} \\ \texttt{t\_fgm[:,1]} &\leftarrow \texttt{t\_fid[t\_pid]}\end{aligned}$$

Figure 9 provides a visual example of this score propagation procedure.

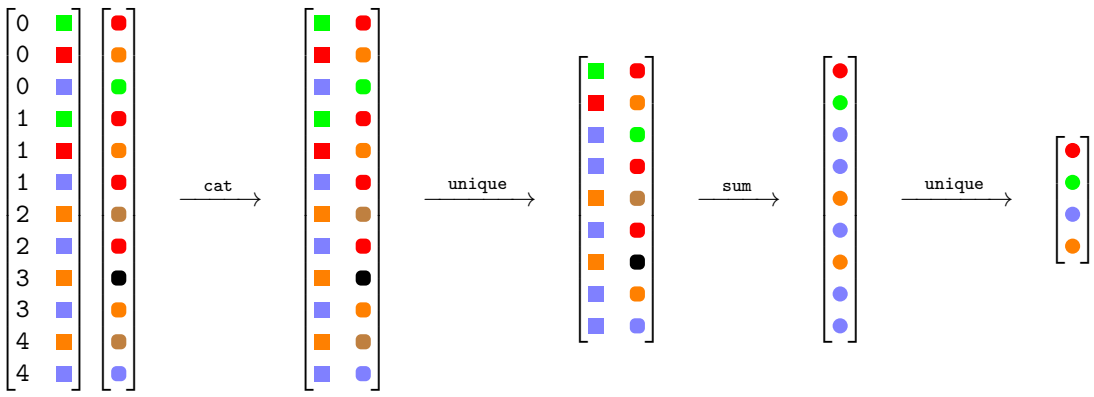

Figure 9: An illustration of between-hand updates for `t_fgm`.

3.6 Infoset Tensors

In this subsection, we represent each node of the Full Game Tree using the Infoset Tensor `t_inf`, which has shape `[56]`. 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, together with the corresponding strategy profile. These representations are later used to train a tree-based model to approximate the strategy profile, as described in Section 4, and are subsequently used in self-play simulations, as detailed in Section 5. It is important to emphasize that these infoset tensors are used only after the Nash equilibrium has been learned using the CFR algorithm. CodeSnippet 4 explains how `t_inf` is constructed.

CodeSnippet 4 Infoset Tensor

```
t_inf ← zeros((Q, 58), dtype=int8) # 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
t_inf[:,55:58]←tensor([i_hnd,i_trn,i_ply])
```

Suppose the current layer of the Game Tree and the Full Game Tree contains `M` and `Q` nodes, respectively, as defined in Table 3. That is, the tensors `t_gme` and `t_fgm` have `M` and `Q` rows, respectively. `t_inf` consists of three distinct parts:

- The first `52` indices are reserved for cards. These entries encode the current status or history of each card, depending on the information available at the node.
- The next three indices (`t_inf[52:55]`) store score-related context inherited from the previous round, such as accumulated club points or point differential.
- The final three indices (`t_inf[55:58]`) encode metadata, including the current round index, the turn counter, and an indicator specifying whose turn it is.

For any dealt card represented in `t_dlt`, if that card is not present in the current game tensor `t_gme`, we assign it a value of `-127` in `t_inf` to indicate that the card has already been collected in one of the previous rounds. We need to clarify why no two rows of `t_gme`

will be mapped to the same compressed tensor. To this end, we summerize in Table ???, all the possible values inside the `t_cmp`.

3.7 Algorithms

CodeSnippet 5 PlayStep

Input `t_inf`, `c_scr`, `t_snw`, `t_sid` # input tensors
`t_inf.shape`= $[d_0, 3, d_2]$, `c_scr`= $[s_1, \cdots, s_{d_0}]$, `t_snw.shape`= $[d_0, 8]$
$v_k := |V_k| = s_1 + \cdots + s_{d_0}$, `t_sid.shape`= $[v_k, 2]$
$[i, j] \in$ `t_sid` *i.e.,* `t_scr[j]` $\in \mathcal{H}$(`t_inf[i]`)
`t_act,c_act = find_moves(t_inf)` # find available moves
`c_act` $= [a_1, \cdots, a_{d_0}]$, `t_act.shape` $= [\tilde{d}_0, 3, d_2]$
$\tilde{d}_0 = a_1 + \cdots + a_{d_0}$
#$v_{k+1} = |V_{k+1}| = s_1 a_1 + \cdots + s_{d_0} a_{d_0}$
`t_inf` $\leftarrow$ `t_inf` $\otimes$ `c_act` # `t_inf.shape`= $[\tilde{d}_0, 3, d_2]$
`t_inf`, `t_snw` $\leftarrow$ `apply_moves(t_inf, t_act, t_snw)`
`t_inf` retains shape. `t_snw.shape`= $[\tilde{d}_0, 8]$
`t_cl1` $\leftarrow$ `t_sid[:,1]`$\otimes_{\texttt{c\_scr}}$`c_act` # See Section ??
`t_edg` $\leftarrow$ `arange(i_sid)`$\otimes_{\texttt{c\_scr}}$`c_act` # `i_sid=sum(c_scr)`
`c_scr` $\leftarrow$ `c_scr`$\otimes$`c_act` # `c_scr`= $[s_1, \cdots, s_1, \cdots, s_{d_0}, \cdots, s_{d_0}]$
`t_cl0` $\leftarrow$ `arange(i_inf)`$\otimes$`c_scr`
`t_sid` $\leftarrow$ `cat([t_cl0,t_cl1])` #`t_sid.shape`= $[v_{k+1}, 2]$
Output `t_inf, c_scr, t_snw, t_sid, t_edg`

CodeSnippet 6 StrategyEvalStep

#The strategy profile at the current information set is evaluated using the neural network v_pnn.
The input to this network, denoted by t_nn, consists of two components: t_nnl and t_nnr.
Input `t_inf,t_scr,t_sid,t_edg,v_pnn`
`t_inf.shape`= $[\tilde{d}_0, 3, d_2]$
`t_sum` $\leftarrow$ `zeros(v_k)` # `t_sid.shape[0]`= v_k
`t_nnl` $\leftarrow$ `t_inf[t_sid[:,0]]` # `t_nnl.shape`=$[v_{k+1}, 3, d_2]$
`t_nnr` $\leftarrow$ `t_scr[t_sid[:,1]]`
`t_nn = cat([t_nnl,t_nnr])`
`t_adv` $\leftarrow$ `v_pnn(t_nn)`
`t_sum.scatter_add_(0,t_edg,t_adv)`
`t_sig` $\leftarrow$ `t_reg/t_sum[t_edge]`
Output `t_sig,t_nn`

CodeSnippet 7 SimulateGameRun

```
Initialize t_dck,t_inf,t_sid,t_scr,t_m52,t_mdd, V_ann,V_bnn,M_snn
  # ← random deck, zeros(1,3,4), zeros(1,2), zeros(1,4), zerbo(52), zerbo(52),3*zeros nn
t_m52[t_dck[:4]] ← True, t_inf[0,0,:]  ← 3, t_dck ← t_dck[4:]
  # deal pool cards
for i_rnd in range(6):
  t_inf, t_dck,t_m52,t_mdd ← DealandUpdate()
    # DealandUpdate(t_dck,t_inf,t_m52,t_mdd)
  c_snw ← zeros(i_inf,8), c_scr ← v_count(t_sid[:,0])
  for i_trn in range(4):
    for i_ply in range(2):
      # update c_scr after StrategyEvalStep
      t_inf,c_scr,t_snw,t_sid,t_edg ←PlayStep()
      # PlayStep(t_inf,c_scr,t_snw,t_sid)
      t_sig,t_nn ← StrategyEvalStep()
      #t_inf[:,0,:][t_inf[:,0,:]==i_ply+1] = 0 # hide info.  NB: t_inf from disk
      #StrategyEvalStep(t_inf, t_scr, t_sid, t_edg, V_pnn)
  if i_rnd == 5:  t_snw ← CleanUpPool()
    #CleanUpPool(t_inf,t_snw)
  t_inf,t_scr,c_scr,t_sid,t_edg ← BetweenHands()
```

CodeSnippet 8 BetweenHands

```
Input t_inf,t_sid,t_scr,t_dck
t_inf, t_lnk = unique_con(t_inf,dim=0)
t_snw, t_wid = unique_con(t_snw,dim=0)
t_prs ← cat([t_sid[:,1],t_wid[t_sid[:,0]]])
t_prs, t_pid ← unique_con(t_prs,dim=0)
t_scr ← t_scr[t_prs[:,0]]+t_snw[t_prs[:,1]]
t_scr,t_fid ← unique_con(t_scr,dim=0)
t_sid[:,1] ← t_fid[t_pid]
t_sid[:,0] ← t_lnk[t_sid[:,0]]
t_sid ← unique(t_sid)
```

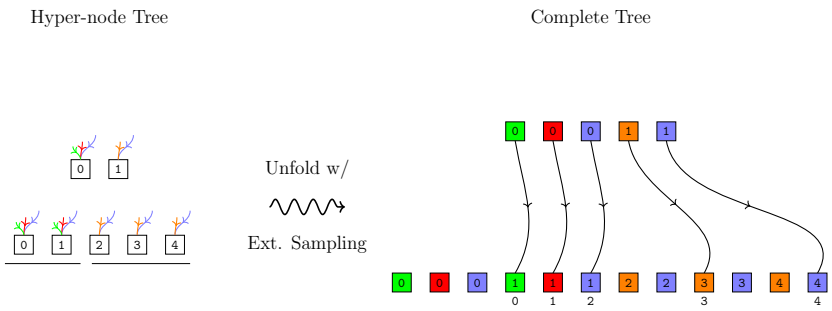

Figure 10: An illustration of how `t_edg` is constructed under ex. sampling regime.

3.8 External Sampling

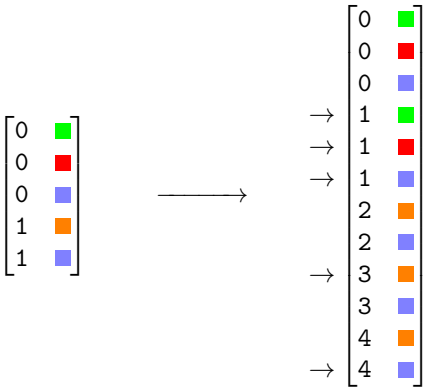

Figure 11: An illustration of how `t_sid` is updated under ext. sampling regime.

4 Tree-Based Fitting of Game Strategies

5 Experimental Results

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 = c_act⊗c_scr = [2,2,2,3,3]
t_inv = stable_argsort(t_edg)
#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]
```